А.Я. Архангельский

C++Builder 6

СПРАВОЧНОЕ ПОСОБИЕ

Книга 1
Язык C++

Москва
ЗАО «Издательство БИНОМ»
2002

УДК 004.43
ББК 32.973.26-018.1
А87

Архангельский А.Я.

C++Builder 6. Справочное пособие. Книга 1. Язык C++. -- М.: Бином-Пресс, 2002 г. — 544 с.: ил.

В книге даются исчерпывающие справочные сведения по языку C++ в C++Builder 6: синтаксис языка, все операции и операторы, все типы данных. Подробно рассматривается работа с исключениями, с текстовыми и двоичными файлами, со строками разных типов, массивами, множествами, структурами, классами. Обсуждается обработка и генерация сообщений Windows. Рассматривается около 650 функций C, C++, API Windows, из них более 300 с подробными описаниями и примерами.

Рассматривается стандартная библиотека шаблонов STL: все типы контейнеров, итераторов, все алгоритмы и функции-объекты.

Представленный в книге справочный материал снабжен подробными комментариями и примерами, что позволяет читателю изучать его практически с нуля.

Как справочник книга полезна пользователям любой квалификации: от начинающих до опытных разработчиков.

ISBN 5-9518-0007-2

Содержание

От автора

Прежде всего, почему эта книга названа справочным пособием? Книга не является учебником. В ней нет последовательного, методически выверенного изложения материала по языку C++. Однако это и не справочник в чистом виде. Сухое изложение справочных сведений полезно, но требует от читателя большого труда по их осмыслению, тестированию, определению области использования того или иного понятия, той или иной функции. Поэтому я остановился на форме справочного пособия. Книга содержит большой объем справочного материала, но он снабжен подробными комментариями, примерами, методическими советами. И стиль изложения большинства разделов достаточно повествовательный. Все это позволяет, по-моему, читателю, знакомому с программированием, но не знакомому, с C++, самостоятельно освоить этот язык.

Книга получилась достаточно объемной, несмотря на усилия автора и издательство сделать все возможное, чтобы в нее вместилось побольше материала: шрифт книги минимально возможный, верстка предельно компактная, изложение тоже, насколько возможно, компактное. Тем не менее, много уже почти готового материала в книгу не вместилось, поскольку я не рискнул увеличивать ее объем и стоимость. В книге подробно рассмотрены далеко не все функции C++ и C++Builder, хотя перечень и краткие характеристики функций охватывают почти весь их перечень. Фрагментарно рассмотрена стандартная библиотека STL — мощный инструмент программирования. Полное изложение всего этого материала увеличило бы книгу примерно вдвое.

Я пока нашел два выход из конфликта между полнотой материала и объемом книги. Во-первых, в недалеком будущем я надеюсь выпустить отдельные книги, вдвое меньшего объема, чем данная, посвященные стандартным библиотекам C и C++. Это позволит полностью описать все функции, контейнеры, алгоритмы этих библиотек. Информацию об этих намеченных книгах вы найдете по ссылкам [3] и [4] в конце книги в разделе «Дополнительные источники информации о C++ и C++Builder 6». Во-вторых, справочные сведения из данной книги плюс немало дополнительного материала включены в справочные файлы [2], сведения о которых вы найдете в разделе «Дополнительные источники информации о C++ и C++Builder 6». Там, по крайней мере, нет ограничений на объем материала. Так что можно постоянно пополнять эти справки, что регулярно и делается. В частности, в обозримом будущем описания стандартных библиотек в них станут полными. Да и стоимость справок заметно отличается от стоимости этой книги. Конечно, справки не могут заменить книгу. Но в них есть и свои преимущества (см. в [2]). Так что я думаю, что справки могут служить хорошим и постоянно развивающимся дополнением к данной книге. По книге, конечно, удобнее изучать ту или иную тему. А справки обеспечивают оперативную помощь в работе, простой способ воспроизведения содержащихся в них примеров и значительно больший объем справочных сведений.

Для читателей, знакомых с моей книгой «Программирование в C++Builder 6» [1], вероятно, полезно представлять, чем различаются материалы данной книги. По сравнению с той книгой [1] справочный материал по C++ в данной книге в несколько раз расширен и переработан. Дано то новое, что введено в последний стандарт C++, существенно расширено описание исключений, классов, шаблонов, дано описание работы с сигналами и сообщениями Windows, рассмотрены битовые поля, объединения и многое другое. В целом, описание языка C++ в данной книге полное, а в [1] — сокращенное.

Существенно расширен в данной книге материал по функциям C++ и API Windows. Помимо краткого изложения функций (их число также значительно увеличено), введена глава, содержащая подробное описание и примеры использования основных функций (около 300).

Введена глава 5, содержащая описание стандартной библиотеки шаблонов C++. Этот мощный инструмент в [1] даже не упоминался.

Так что, если сравнивать данную книгу с материалом по C++ в книге [1], то эта книга более чем на 2/3 состоит из совершенно нового материала, а включенный в нее прежний материал существенно переработан. Так что думаю, что данная книга может служить хорошим дополнением к прежней.

Глава 1

Справочные данные по языку C++

В настоящей главе приводятся основные справочные сведения по той версии языка C++, которая используется в C++Builder. Впрочем, некоторые конструкции, применяемые в C++Builder, характерны скорее для языка C, а не C++. А некоторые особенности языка, связанные с библиотечными компонентами, относятся к языку Object Pascal. Так что сведения, приводимые в этой и последующих главах, относятся ко всем языкам, используемым в C++Builder. Однако в случаях, когда возможно применить несколько альтернативных подходов, предпочтение все-таки отдается C++.

В этой и следующей главах практически не затрагиваются вопросы, связанные с STL — стандартной библиотекой шаблонов C++. Этой библиотеке и, соответственно, таким базовым для C++ понятиям, как контейнер, итератор, алгоритм и т.д., посвящена гл. 5.

1.1 Язык C++ и его синтаксис

Язык программирования C++ был создан на основе языка C и сохраняет до сих пор язык C как свое подмножество. Синтаксис языка, основные операторы и операции, многие встроенные типы данных заимствованы в C++ из C. Поэтому большинство программ, написанных на C, будут аналогично функционировать и в среде C++. А основные достижения C++ по сравнению с C — объектная ориентация, поддержка абстракции данных, наследования, полиморфизма, возможность перегрузки операций, поддержка обработки ошибок с помощью исключений.

Многие из этих возможностей — абстракция данных, наследование, полиморфизм связаны с базовым понятием языка C++ — классами. Классы рассматриваются в гл. 2 в разд. 2.14. Шаблоны классов и функций, существенно обогатившие язык, рассмотрены, соответственно, в разд. 2.14.8 и 1.7.8. Перегрузке операций посвящен разд. 1.9.17. Исключения рассмотрены в разд. 1.12. А в данном разделе коротко излагается синтаксис языка C++, во многом тождественный языку C.

Основные синтаксические правила записи программ на языке C++ сводятся к следующему:

Прописные и строчные буквы считаются разными символами. Поэтому, например, идентификаторы **DATABASE, DataBase, Database и database** относятся к совершенно разным переменным, константам или объектам. При записи идентификаторов могут использоваться латинские буквы, цифры, символ подчеркивания "_". Идентификатор не может начинаться с цифры и не может содержать пробельных символов. Длина идентификатора не ограничена, но ради удобства чтения программы надо стремиться использовать короткие и осмысленные идентификаторы.

Пробельные символы (пробелы, знаки табуляции, символ новой строки, комментарий) могут размещаться в любом месте текста, но не внутри идентификатора.

Комментарии в тексте заключаются в скобки вида /* текст комментария */. Такие комментарии могут вводиться в любом месте текста, в частности, внутри операторов, и занимать любое количество строк. Вложенные комментарии обычно не допускаются. Считается, что комментарий закончился, как только в тексте встретились первые символы "*/". Впрочем, в C++Builder 6 можно обеспечить использование вложенных комментариев. Для этого надо включить опцию Nested Comments на странице Advanced Compiler окна опций проекта. Однако в стандарте С вложенные комментарии не допускаются, так что их использование делает код непереносимым на другие платформы. Именно поэтому данная опция по умолчанию выключена. Еще один способ введение комментария — размещение его после двух символов слэш "//". Этот комментарий должен занимать конец строки, в которой он введен, и не может переходить на следующую строку. Любой текст в строке, помещенный после символов "//", воспринимается как комментарий.

Каждое предложение языка кончается символом точка с запятой ";". Немногие исключения из этого правила будут оговорены особо.

В строке может размещаться несколько операторов. Однако с точки зрения простоты чтения текста этим не надо злоупотреблять. Вообще, надо писать программу так, чтобы ее было легко читать и вам, и постороннему человеку, которому, может быть, придется ее сопровождать. Надо выделять объединенные смыслом операторы в группы, широко используя для этого отступы и комментарии.

Фигурные скобки { } выделяют составной оператор. Все операторы, помещенные между ними, воспринимаются синтаксически как один оператор.

Все используемые типы, константы, переменные, функции должны быть объявлены или описаны до их первого использования. Объявления могут встречаться в любом месте текста.

1.2 Программы на C++

1.2.1 Общие сведения

Программа на C++ состоит из *объявлений* (переменных, констант, типов, классов, функций) и *описаний функций*. Среди функций всегда имеется главная – **main** для консольных приложений (работающих с WIN32) или **WinMain** для приложений Windows. Именно эта главная функция выполняется после начала работы программы. Обычно в C++Builder эта функция очень короткая и выполняет только некоторые подготовительные операции, необходимые для начала работы. А далее при объектно-ориентированном подходе работа приложения определяется происходящими событиями и реакцией на них объектов.

Как правило, программы строятся по модульному принципу и состоят из множества *модулей*. Принцип модульности очень важен для создания надежных и относительно легко модифицируемых и сопровождаемых приложений. Четкое соблюдение принципов модульности в сочетании с принципом скрытия информации позволяет внутри любого модуля проводить какие-то модификации, не затрагивая при этом остальных модулей и головную программу.

В C++Builder все объекты компонентов размещаются в объектах — формах. Для каждой формы, которую вы проектируете в своем приложении, C++Builder создает отдельный модуль. Именно в модулях и осуществляется программирование задачи. В обработчиках событий объектов — форм и компонентов, вы помещаете все свои алгоритмы. В основном они сводятся к обработке информации, содержащейся в свойствах одних объектов, и задании по результатам обработки свойств других объектов. При этом вы постоянно обращаетесь к методам различных объектов. Вопросами доступа к свойствам и методам объектов мы и займемся в дальнейших разделах данной главы.

Согласно принципам скрытия информации обычно текст модуля разделяют на *заголовочный файл* интерфейса, который содержит объявления классов, функций, переменных и т.п., и *файл реализации,* в котором содержится описание функций. Стандартное расширение файлов реализации — .cpp. Стандартное расширение заголовочных файлов — .h.

После того как программа написана, на ее основе должен быть создан *выполняемый файл* (модуль). Этот процесс осуществляется в несколько этапов.

Сначала работает *препроцессор,* который преобразует исходный текст. Препроцессор осуществляет преобразования в соответствии со специальными *директивами препроцессора,* которые размещаются в исходном тексте. Препроцессор может в соответствии с этими директивами включать тексты одних файлов в тексты других, развертывать *макросы* — сокращенные обозначения различных выражений и выполнять множество других преобразований.

После окончания работы препроцессора начинает работать *компилятор.* Его задача — перевести тексты модулей в машинный (объектный) код. В результате для каждого исходного файла **.cpp** создается объектный файл, имеющий расширение **.obj.**

После окончания работы компилятора работает *компоновщик,* который объединяет объектные файлы в единый загрузочный выполняемый модуль, имеющий расширение **.exe.** Этот модуль можно запускать на выполнение.

1.2.2 Структура головного файла проекта

В процессе проектирования вами приложения C++Builder автоматически создает коды головного файла проекта, коды отдельных модулей и коды их заголовочных файлов. Головной файл проекта, предназначенного для работы в среде Windows, содержит головную функцию **WinMain.** Если делается консольное приложение, то головной является функция **main.** В прочие модули вы вводите свой код, создавая обработчики различных событий. В заголовочные файлы этих модулей вы вводите свои объявления. Но головной модуль, как правило, вы не трогаете и даже не видите его текст. Только в исключительных случаях вам надо что-то изменять в тексте головного модуля, сгенерированном C++Builder. Тем не менее, хотя бы ради этих исключительных случаев, надо все-таки представлять вид головного файла проекта и понимать, что означают его операторы.

Чтобы увидеть код головного файла проекта, надо выполнить в среде разработки C++Builder команду Project | View Source. Типичный головной файл проекта для Windows имеет следующий вид (в приведенный текст добавлены русские комментарии):

```
//---------------------------------------------------------------------
// директивы препроцессора
#include <vcl.h>
#pragma hdrstop

// макросы, подключающие файлы ресурсов и форм
USERES("Project1.res");
USEFORM("Unit1.cpp", Form1);
USEFORM("Unit2.cpp", Form2);
//---------------------------------------------------------------------

// функция main
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
```

```
        Application->CreateForm(__classid(TForm2), &Form2);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
```

Начинается файл головного модуля строками, первый символ которых — "#". С этого символа начинаются директивы препроцессора (см. подробности в разд. 1.4). Среди них наиболее важны для вас директивы **#include**. Эти директивы подключают в данный файл тексты указанных в них файлов. В частности, подобными директивами включаются в текст заголовочные файлы. Например, директива **#include <vcl.h>** подключает заголовочный файл **vcl.h**, содержащий объявления, используемые в библиотеке визуальных компонентов C++Builder.

После директив препроцессора в файле размещены предложения **USERES** и **USEFORM.** Это макросы, используемые для подключения к проекту файлов форм, ресурсов и др. Препроцессор развернет эти макросы в соответствующий код. В данном случае вы можете видеть два макроса **USEFORM,** подключающих формы. C++Builder автоматически формирует соответствующее предложение с макросом **USEFORM** для каждой формы, вносимой вами в проект. Первый параметр макроса содержит имя файла модуля, соответствующего форме (например, "Unit1.cpp"), а второй параметр — имя формы.

После всех этих вспомогательных предложений в файле расположена главная функция программы - - **WinMain.** Ее первым параметром является дескриптор данного экземпляра приложения. Дескриптор — это некий уникальный указатель, позволяющий Windows разбираться в множестве одновременно открытых окон различных приложений. Иногда вы будете использовать дескрипторы при обращении к различным функциям API Windows (API Windows — это пользовательский интерфейс Windows, содержащий множество полезных функций). Второй параметр **WinMain** - - дескриптор предыдущего экземпляра вашего приложения (если пользователь выполняет одновременно несколько таких приложений). Третий параметр является указателем на строку с нулевым символом в конце, содержащую параметры, передаваемые в программу через командную строку. Иногда такие параметры используются для переключения режимов работы программы или для задания различных опций при запуске приложения из диспетчера программ или функцией **WinExec.** Последний параметр определяет окно приложения. Этот параметр может в дальнейшем передаваться в функцию **ShowWindow.**

Более подробное описание функции **WinMain** вы найдете в гл. 4. В той же главе вы найдете описание головной функции main, используемой в консольных приложениях. В ней тоже имеются параметры, дающие доступ к элементам командной строки. Впрочем, эти параметры и **в WinMain, и в main** используются достаточно редко.

После заголовка функции **WinMain** следует ее тело, заключенное в фигурные скобки. Первый выполняемый оператор тела функции — **Application->Initialize** инициализирует объекты компонентов данного приложения. Последующие операторы **Application->CreateForm** создают объекты соответствующих форм. Формы создаются в той последовательности, в которой следуют эти операторы. Первая из создаваемых форм является главной.

Последний оператор — **Application->Run** начинает собственно выполнение программы. После этого оператора программа ждет соответствующих событий, которые и управляют ее ходом.

Перечисленные операторы тела функции **WinMain** заключены в блок **try,** после которого следует блок **catch.** Это структура, связанная с обработкой так назы-

ваемых исключений — аварийных ситуаций, возникающих при работе программы. Если такая аварийная ситуация возникнет, то будут выполнены операторы, расположенные в блоке **catch.** По умолчанию в этом блоке расположен стандартный обработчик исключений с помощью функции **Application->ShowException.** Подробнее об обработке исключений вы можете посмотреть в разд. 1.12.

Последним оператором тела функции **WinMain** является оператор **return(0),** завершающий приложение с кодом завершения 0.

Все описанные выше операторы головного файла приложения заносятся в него автоматически в процессе проектирования вами приложения. Например, при добавлении в проект новой формы в файл автоматически вставляются соответствующее предложение **USEFORM** и оператор **Application->CreateForm,** создающий форму. Так что обычно ничего в головном файле изменять не надо и даже нет необходимости его смотреть.

Если вам надо ввести какой-то свой текст в головной модуль, вы можете сделать это, введя объявления необходимых констант, переменных, функций, добавить или изменить операторы в теле функции. Например, вам может потребоваться при запуске приложения на выполнение провести какие-то настройки (например, настроить формы на тот или иной язык — русский или английский). Или сделать какой-то запрос пользователю и в зависимости от ответа создавать или не создавать те или иные формы. Или проанализировать параметры, переданные в программу через командную строку.

Только учтите, что все определенные вами в головном файле проекта глобальные константы и переменные будут доступны в другом блоке только в случае, если они объявлены там со спецификацией **extern** (см. разд. 1.6.2). Функции, определенные вами в головном файле проекта, будут доступны в другом блоке только в случае, если там повторен их прототип (см. разд. 1.8).

Посмотрим, как можно вводить собственный код в головной файл. Пусть, например, вы хотите, чтобы вторая форма вашего приложения **Form2** создавалась только в случае, если при запуске приложения через командную строку в него передана опция Y. В этом случае вы можете изменить заголовок функции **WinMain** следующим образом:

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR S, int)
```

Этот заголовок объявляет строку S, в которой будет содержаться текст командной строки. Тогда приведенный выше оператор

```
Application->CreateForm(__classid(TForm2), &Form2);
```

можно заменить оператором

```
if (S[0] — 'Y')
   Application->CreateForm(__classid(TForm2), &Form2);
```

В этом случае, если ваше приложение **Project1** будет запускаться командой **Project1 Y,** то форма **Form2** будет создаваться. В остальных случаях этой формы не будет.

Аналогичный выбор можно сделать, не анализируя командную строку, а предложив пользователю соответствующий вопрос. В этом случае приведенный выше оператор можно заменить следующим (о методе **MessageBox см. в гл.** 9):

```
if (Application->MessageBox(
"Хотите создать вторую форму?",
            "Подтвердите создание второй формы",
            MB_YESNOCANCEL + MB_ICONQUESTION) == IDYES)
   Application->CreateForm(__classid(TForm2), &Form2);
```

Тогда в момент запуска приложения пользователю будет сделан запрос, показанный на рис. 1.1. При положительном ответе пользователя форма будет создана, в противном случае — нет.

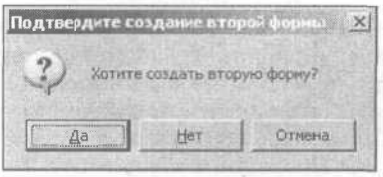


Рис. 1.1
Окно запроса пользователю

Вы можете, конечно, ввести в головной файл программы и другие операторы, функции и т.п. Все это можно сделать, но это будет плохой стиль программирования, поскольку он противоречит принципу модульности. Все необходимые вам в начале выполнения процедуры и функции настройки помещайте в один из модулей форм, а еще лучше — в отдельный модуль без формы. Такой модуль, не связанный с какой-то формой, можно включить в приложение, выполнив в среде разработки C++Builder команду File | New и щелкнув на пиктограмме Unit. В этом или ином модуле вы можете предусмотреть функцию, которая осуществляет все необходимые настройки. Тогда в головной программе достаточно будет вызвать в соответствующий момент эту функцию, передав в нее, если необходимо, какие-то параметры, например, текст командной строки.

Пусть, например, вы написали некоторую функцию, назвав ее **begin,** в которой проводится настройка программы в зависимости от опций, переданных через командную строку. И пусть вы поместили эту функцию в модуль **Unitl.cpp,** а ее объявление — в заголовочный файл этого модуля **Unitl.h** (вне описания класса — см. разд. 1.2.3). Тогда вы можете в головной файл приложения включить директиву препроцессора

```
#include "Unitl.h"
```

подключающую файл **Unitl.h,** изменить заголовок функции **WinMain** на

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR S, int)
```

т.е. ввести параметр S, воспринимающий командную строку, и поставить первым выполняемым оператором функции **WinMain** оператор:

```
begin(S);
```

вызывающий функцию begin и передающий в нее текст командной строки.

Сама функция **begin** может иметь вид:

```
void begin(String s)
{
// операторы анализа командной строки и настройки
}
```

Таким образом вы, не нарушив принципа модульности, можете осуществить все действия, необходимые вам в начале работы программы.

Имя головного файла проекта по умолчанию дается стандартное: **Projectl, Project2** и т.п. Это же имя будет и у выполняемого модуля вашей программы. Так что желательно изменить имя по умолчанию. Для этого достаточно сохранить головной файл проекта под соответствующим именем.

Выше подробно рассмотрен вариант функции **WinMain,** используемой в приложениях Windows. Описание функции **main,** используемой в консольных приложениях, см. в гл. 4.

1.2.3 Структура файлов модулей форм

Рассмотрим теперь, как выглядят тексты модулей форм. Каждый такой модуль состоит из двух файлов: заголовочного, содержащего описание класса формы, и файла реализации. Ниже приведены тексты этих файлов модуля формы, на которой размещена одна метка (компонент типа **TLabel)** и одна кнопка (компонент

типа **TButton).** Подробные комментарии в этом тексте поясняют, куда и что в этот код вы можете добавлять.

Заголовочный файл:

```
//---------------------------------------------------------------------------
tfifndefUnit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
tinclude <StdCtrls.hpp>
#include <Forms.hpp>
// сюда могут помещаться дополнительные директивы
// препроцессора (в частности, include),
// не включаемые в файл автоматически

//---------------------------------------------------------------------------
// объявление класса формы TForm1
class TForm1 : public TForm
{
__published:      // IDE-managed Components
// размещенные на форме компоненты
  TButton *Button1;
   TLabel *Label1;
   void___fastcall Button1Click(TObject *Sender);

private:   // User declarations
// закрытый раздел класса
// сюда могут помещаться объявления типов, переменных, функций,
// включаемых в класс формы, но не доступных для других модулей

public:           // User declarations
// открытый раздел класса
// сюда могут помещаться объявления типов, переменных, функций,
// включаемых в класс формы и доступных для других модулей
   __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
// сюда могут помещаться объявления типов, переменных, функций,
// которые не включаются в класс формы;
// доступ к ним из других блоков возможен только при соблюдении
// некоторых дополнительных условий
#endif
```

Файл реализации:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"

// сюда могут помещаться дополнительные директивы
// препроцессора (в частности, include),
// не включаемые в файл автоматически

// объявление объекта формы Form1
TForm1 *Form1;
//---------------------------------------------------------------------------
```

```
// вызов конструктора формы Form1
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
// сюда вы можете поместить операторы,
// которые должны выполняться при создании формы
}
//---------------------------------------------------------
// сюда могут помещаться объявления типов и переменных,
// доступ к которым из других модулей возможен только при
// соблюдении некоторых дополнительных условий;
// тут же должны быть реализации всех функций, объявленных в
// заголовочном файле, а также могут быть реализации любых
// дополнительных функций, не объявленных ранее

void__fastcall TForm1::Button1Click(TObject *Sender)
(
 Close();
}
```

Рассмотрим подробнее эти файлы. Заголовочный файл начинается с автоматически включенных в него директив препроцессора. В частности, C++Builder сам помещает тут директивы **include** (см. разд. 1.4), подключающие копии файлов, в которых описаны те компоненты, переменные, константы, функции, которые вы используете в данном модуле. Однако для некоторых функций такое автоматическое подключение не производится. В этих случаях разработчик должен добавить соответствующие директивы **include** вручную.

После директив препроцессора следует описание класса формы. Имя класса вашей формы — TForml. Класс содержит три раздела:___published — открытый раздел, содержащий объявления размещенных на форме компонентов и обработчиков событий в них, **private** — закрытый раздел класса, и **public** — открытый раздел класса. В данном случае в разделе____**published** вы можете видеть объявления указателей на два компонента: компонент **Buttonl** типа **TButton** и компонент **Labell** типа **TLabel.** Там же вы видите объявление функции **ButtonlClick** — введенного пользователем обработчика события щелчка на кнопке **Buttonl.** Все, что имеется в разделе___**published,** C++Builder включает в него автоматически в процессе проектирования вами формы. Так что вам не приходится, как правило, работать с этим разделом. А в разделы **private** и **public** вы можете добавлять свои объявления типов, переменных, функций. То, что вы или C++Builder объявите в разделе **public,** будет доступно для других классов и модулей. То, что объявлено в разделе **private,** доступно только в пределах данного модуля. Как вы можете видеть, единственное, что C++Builder самостоятельно включил в раздел **public,** это объявление (прототип) конструктора вашей формы TForml.

После объявления класса следует предложение PACKAGE, которое включается в файл автоматически и которое мы сейчас рассматривать не будем. После этого вы можете разместить объявления типов, переменных, функций, к которым при соблюдении некоторых дополнительных условий (см. разд. 1.8) будет доступ из других модулей, но которые не включаются в класс формы.

Теперь рассмотрим текст файла реализации модуля. После автоматически включенных в этот файл директив препроцессора следует тоже автоматически включенное объявление указателя на объект формы **Forml,** а затем — вызов конструктора формы. Тело соответствующей функции пустое, но вы можете включить в него какие-то операторы. Эти операторы будут выполняться при создании формы. В них можно включить какие-то начальные настройки свойств этой формы.

После конструктора размещаются описания всех функций, объявленных в заголовочном файле. Вы можете также размещать здесь объявления любых типов,

констант, переменных, не объявленных в заголовочном файле, и размещать описания любых функций, не упомянутых в заголовочном файле.

Имена файлам модулей C++Builder дает по умолчанию: для первого модуля имя равно "Unit1", для второго — "Unit2" и т.д.

1.2.4 Доступ к объектам, переменным и функциям модуля

1.2.4.1 Пример модуля, содержащего объекты и процедуры

Рассмотрим, как можно вводить в модуль переменные, функции и осуществлять к ним доступ. Ниже приведен текст кода модуля, в котором на форме размещены два компонента: кнопка Button1 типа TButton и метка Label1 типа TLabel. Кроме того, в модуле введен обработчик события, связанного со щелчком пользователя на кнопке, и в разных местах модуля введены переменные и функции, чтобы можно было видеть, как получить к ним доступ.

Заголовочный файл:

```
//---------------------------------------------------------------------
#ifndef Unit1H
tfdefine Unit1H
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:      // IDE-managed Components
    TLabel *Label1;
    TButton *Button1;
    void___fastcall Button1Click(TObject *Sender);

private:   // User declarations
//Функция F1 и переменна Ch6 доступны только в данном модуле
void___fastcall F1(char Ch);
char Ch6;

public:           // User declarations
    __fastcall TForm1(TComponent* Owner);
// Переменная Ch1 и функция F2  доступны для объектов
// любых классов и для других модулей, но со ссылкой
// на объект
char Ch1;
void___fastcall F2(char Ch);
};
//---------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
// Глобальная переменная Ch2 и функция F3 доступны в пределах
// данного модуля; переменная Ch2 доступна в других модулях,
// если определена там со спецификацией extern;
// функция F3 доступна в других модулях, если там содержится
// ее прототип
char Ch2;
void F3(char Ch);
#endif
```

Файл реализации:

```
//---------------------------------------------------------------------
#include <vcl.h>
```

```
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------
// Глобальная переменная Ch3 и функция F4 доступны в пределах
// данного модуля; переменная Ch3 доступна в других модулях,
// если определена там со спецификацией extern;
// функция F4 доступна в других модулях, если там содержится
// ее прототип
char Ch3;

void F4(char Ch)
{
 Form1->Label1->Caption = Form1->Label1->Caption + Ch +
                          Form1->Ch1;
}
//---------------------------------------------------------------------
void __fastcall TForm1::F1(char Ch)
{
 Label1->Caption = Label1->Caption + Ch + Ch1;
}
//---------------------------------------------------------------------
void __fastcall TForm1::F2(char Ch)
{
 Label1->Caption = Label1->Caption + Ch + Ch1;
}
//---------------------------------------------------------------------
void F3(char Ch)
{
 Form1->Label1->Caption = Form1->Label1->Caption + Ch +
                          Form1->Ch1;
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
// Переменная Ch4 доступна только внутри данной функции
 char Ch4;
 Ch1 = '-';
 Ch2 = 'A';
 Ch3 = 'B';
 Ch4 = 'C';
 Label1->Caption = "";
 F1(Ch1);
 F2(Ch2);
 F3(Ch3);
 F4(Ch4);
 Label1->Font->Color = clRed;
}
```

Помимо функции **ButtonlClick** обработки щелчка на кнопке **Button1** в код занесено в разные места модуля еще несколько одинаковых функций: Fl — F4, и несколько переменных символьного типа: **Ch1** — Ch4. Сейчас мы не будем разбираться подробно в самих функциях. Нам важно понять, как из функций обращать-

ся к различным объектам и переменным. Но краткое описание того, что делают эти функции, надо дать.

У метки типа **TLabel** имеется свойство **Caption** — надпись на метке. Каждая из функций **F1–F4** берет значение этой надписи, прибавляет к ней символ, переданный в нее как параметр Ch, прибавляет далее символ, хранящийся в переменной **Ch1**, и возвращает надпись с этими добавлениями обратно в метку.

Обработчик щелчка на кнопке — функция **TForm1::Button1Click**, задает символьным переменным **Ch1–Ch4** значения символов "-", "A", "B" и "C", затем очищает свойство **Caption** метки **Labell,** занося в него пустую строку, а затем поочередно обращается к функциям **F1–F4**, передавая в них в качестве параметров различные символы. В заключение надпись метки окрашивается в красный цвет. Для этого используется свойство **Font** — шрифт объекта **Labell.** Это свойство само является объектом, имеющим свойство **Color** — цвет. Значение этого свойства изменяет последний оператор процедуры **TForm1::Button1Click.**

1.2.4.2 Доступ к свойствам и методам объектов

Рассмотрим теперь, как получить из программы доступ к свойствам и методам объектов. Доступ к интересующим нас объектам — компонентам можно получить через объявленные в заголовочном файле модуля указатели на эти объекты. Например, в объявлении класса формы **TForml** в заголовочном файле имеется строка

```
TLabel *Label1;
```

Эта строка объявляет **Labell** как указатель на метку — объект типа **TLabel.** Если вы плохо представляете, что такое указатели, посмотрите в гл. 2 разд. 2.8, или просто отнеситесь пока к тому, о чем будет рассказано ниже, как к некоторому обязательному формализму. Честно говоря, программы в C++Builder часто можно писать, не задумываясь о сути этого формализма.

Доступ к элементам класса (данным — свойствам и функциям — методам) обеспечивается одним из следующих двух способов. Можно использовать *операцию стрелка* (символ " " и символ ">", записанные без пробела, т.е. "->") или операцию точка (.). Первая из них применяется при обращении к объекту через указатель на него, вторая — при обращении по имени переменной объекта или по ссылке на него.

Посмотрите, например, в приведенном выше файле реализации модуля тексты функций F1 и F2. Вы увидите в них выражения вида **Label1->Caption.** Они означают: свойство **Caption** объекта **Labell.** То же самое свойство **Caption** можно было бы получить и с помощью выражения **(*Label1).Caption.** Здесь операция разыменование указателя **(*Labell)** дает сам объект, к которому можно применять операцию точка.

Хотя эти два способа доступа к свойствам и методам объекта эквивалентны, обычно в C++Builder используется операция стрелка. Поэтому в дальнейшем изложении в данной книге мы практически всегда будем пользоваться этой формой записи.

Иногда свойство объекта является в свою очередь объектом. Тогда в обращении к этому свойству указывается вся цепочка предшествующих объектов. Например, метки имеют свойство **Font** — шрифт, которое в свою очередь является объектом. У этого объекта имеется множество свойств, в частности, свойство **Color** - цвет шрифта. Чтобы сослаться на цвет шрифта метки **Labell,** надо написать **Label1->Font->Color** (см. в тексте примера функцию **TForml::ButtonlClick).** Это означает: свойство **Color** объекта **Font,** принадлежащего объекту **Labell.**

Аналогичная нотация используется и для доступа к методам объекта. Например, для метки, как и для большинства других объектов, определен метод **Hide,** который делает метку невидимой. Если вы в какой-то момент решили сделать метку **Labell** невидимой, можете написать оператор

```
Label1->Hide();
```

1.2.4.3 Различие переменных и функций, включенных и не включенных в описание класса

Теперь посмотрим, чем различаются переменные и функции, включенные и не включенные в описание класса. Переменные и функции, включенные в описание класса, обычно называются соответственно *данными-элементами* и *функциями-элементами*. Применительно к объектно-ориентированному проектированию в C++Builder чаще их называют свойствами и методами. В приведенном в разд. 1.2.4.1 примере переменная **Ch1** и функции F1 и F2 включены в описание класса, а переменные Ch2, Ch3 и функции F3 и F4 объявлены вне класса. В чем будет проявляться различие в их использовании?

Если в приложении создается только один объект данного класса (в нашем примере — только один объект формы класса **TForm1**), то различие в основном чисто внешнее. Для функций, объявленных в классе, в их описании к имени функции должна добавляться ссылка на класс с помощью так называемой бинарной операции разрешения области действия "::" — см. разд. 1.9.13. В нашем примере имена функций **F1**, **F2** и **ButtonlClick** в описании этих функций заменяются на **TForml::Fl, TForm1::F2 и TForml::ButtonlClick.** Тем самым указывается, что речь идет о функциях класса **TForml.** Для функций F3 и F4, объявленных вне класса, такого дополнения к имени не требуется.

Необходимость добавления в имена функций, описанных в классе, ссылок на класс объясняется просто. Вы можете вне класса описать другую свою функцию с тем же именем (например, Fl), что и у функции класса. И тогда из функций, не описанных в классе, вы сможете ссылаться на обе эти функции **F1,** только на одну из них непосредственно — по имени Fl, а на другую через объект класса — **Forml::Fl.** Благодаря этому, при описании своих функций вне класса вы можете даже не знать имен всех функций, описанных в классе (может быть этот класс описан в другом модуле, текст которого вы не видели). Никакой путаницы при этом не возникнет.

Таким образом, применение операции разрешения области действия позволяет объявить в разных классах и вне классов переменные и функции с одинаковыми именами. В этом случае операция разрешения области действия указывает, о какой именно переменной или функции идет речь. В некоторых случаях при работе с C++Builder разрешение области действия приходится делать вручную. Это происходит в тех случаях, когда компилятор выдает сообщение, что не может выбрать одну из нескольких альтернатив, например, не знает, к какому классу относится указанный вами метод.

Обращение к переменным и функциям, описанным внутри и вне класса, из функций, описанных вне класса, различается. К переменным и функциям, описанным вне класса, обращение происходит просто по их именам, а к переменным и функциям, описанным в классе, через имя объекта класса. Поэтому в нашем примере в функциях F3 и F4 обращение к переменной СЫ имеет вид **Form1-> СЫ.** По той же причине и обращение к свойству **Caption** объекта **Label**1 в этих функциях имеет вид **Forml->Labell->Caption.** Только через ссылку на объект **Forml** внешние по отношению к классу функции могут получить доступ ко всему, объявленному в классе.

Все эти ссылки на объект не требуются в функциях, объявленных в классе. Поэтому в функциях **TForml::Fl, TForml::F2 и TForm1::Button1Click** ссылки на переменную СЫ и на объект **Labell** не содержат дополнительных ссылок на объект формы.

Если в приложении создается несколько объектов одного класса, например, несколько форм класса **TForml** (это часто делается в приложениях с интерфейсом множества документов MDI), то проявляются более принципиальные различия между переменными, описанными внутри и вне класса. Переменные вне класса (в на-

шем примере Ch2 и Ch3) так и остаются в одном экземпляре. А переменные, описанные в классе (в нашем примере Ch1), тиражируются столько раз, сколько объектов данного класса создано. Т.е. в каждом объекте класса TForml будет своя переменная Ch1, и все они друг с другом никак не будут связаны. Таким образом, в переменную, описанную внутри класса, можно заносить какую-то информацию, индивидуальную для каждого объекта данного класса. А переменная, описанная в модуле вне описания класса, может хранить только одно значение.

Отметим без детальных пояснений еще одну особенность, которую вы уже, вероятно, заметили в приведенных текстах файлов. Перед именами функций-элементов класса ставится опция компилятора___fastcall. Не вдаваясь в детали, скажем, что эта опция влияет на процесс компиляции и обеспечивает передачу параметров функции в быстрые регистры, что ускоряет вызов функции. Для функций-элементов классов эту опцию следует указывать всегда. Для других функций ее можно указывать, а можно и не указывать. Впрочем, эту опцию целесообразно указывать и для функций, не являющихся элементами класса (в приведенном примере это не сделано, просто чтобы подчеркнуть различие между функциями-элементами и прочими функциями).

В заключение просуммируем изложенные правила.

- В реализацию функций-элементов должна добавляться ссылка на класс с помощью операции разрешения области действия (::).
- Функции-элементы объявляются и описываются с применением опции компилятора___fastcall.
- В функциях-элементах обращение к другим функциям-элементам и данным-элементам того же класса может осуществляться без указания на объект.
- В функциях, не являющихся элементами класса данного объекта, доступ к функциям-элементам (методам) и данным-элементам (свойствам) осуществляется через указатель на объект с помощью операции стрелка (->) или (значительно реже) с помощью разыменования указателя на объект и операции точка (.).

1.3 Компиляция и компоновка проекта

Превращение проекта в выполняемый модуль включает в себя два последовательно протекающих процесса: компиляцию и компоновку проекта. Компиляция, в свою очередь, включает в себя работу препроцессора, упрощающего и проверяющего исходный текст, и собственно компиляцию.

Компиляция приложения в C++Builder может выполняться несколькими способами. Компиляция с последующим выполнением приложения осуществляется командой Run | Run, или соответствующей быстрой кнопкой, или «горячей» клавишей F9. В этом случае производится компиляция программы, ее компоновка, создается выполняемый модуль .exe и он запускается на выполнение. Впрочем, создание модуля .exe и выполнение будет проводиться только в случае, если при компиляции и компоновке не обнаружены неисправимые ошибки.

В процессе компиляции и компоновки на экране появляется окно, приведенное на рис. 1.2. В его верхней строке вы видите имя компилируемого проекта. В следующей строке отображается текущая операция: компиляция определенного модуля (на рис. 1.2 показан момент компиляции модуля Unit2.cpp) или компоновка (Linking). В третьей строке окна отображается текущая строка модуля (Current line), обрабатываемая компилятором, и общее число строк в модуле (Total lines). В нижней строке отображается обнаруженное на данный момент число замечаний (Hints), предупреждений (Warnings) и ошибок (Errors). Клавиша Cancel внизу окна позволяет прервать процесс компиляции и компоновки.

Если в компилируемом файле встретились неисправимые ошибки, выполняемый файл не будет создан. Если ошибок нет, файл создастся, но и в этом случае у компилятора могут быть предупреждения и замечания, которые вам надо внимательно изучить.

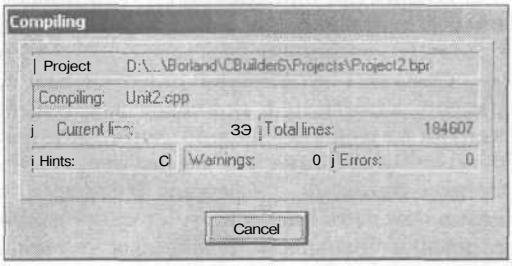


Рис. 1.2
Окно компиляции и компоновки

При компиляции проекта, состоящего из нескольких модулей, компилируются только те модули, тексты которых были изменены с момента предыдущей компоновки проекта. Это существенно экономит время компиляции.

При выполнении команды Run вы можете задать командную строку, если ваше приложение предусматривает передачу в него каких-то параметров. Для этого надо сначала выполнить команду Run Parameters и в открывшемся окне написать требуемую командную строку.

Не всегда вам надо компилировать проект и тут же выполнять его. Часто вам важнее просто проверить, не содержат ли ваши последние изменения кода каких-то ошибок. В этом случае вам не имеет смысла терять время на выполнение проекта и лучше воспользоваться другими командами меню: Project Compile Unit, Project | Make Project или Project Build Project.

Команда Compile выполняет компиляцию только того модуля, который выделен вами в окне Редактора Кода или в Менеджере Проектов. Эта команда позволяет наиболее быстро проверить наличие ошибок или замечаний при компиляции модуля, так как не осуществляется компоновка программы и не компилируются никакие другие модули. Если компиляция прошла успешно, создается объектный файл .obj откомпилированного модуля.

Команда Make выполняет компиляцию всех тех модулей, тексты которых были изменены с момента предыдущей компоновки проекта. Если компиляция прошла успешно, то создаются объектные файлы модулей .obj и осуществляется компоновка программы. Если и она прошла успешно, то создается выполняемый модуль .exe. Таким образом, отличие Make от Run только в том, что после компоновки не производится выполнение приложения.

Команда Build подобна команде Make за одним исключением — компилируются все модули, независимо от того, когда они в последний раз изменялись. Конечно, выполнение этой команды требует наибольшего времени. Но иногда только ее и можно использовать. Например, если с момента последней компиляции вы ничего не изменяли в модулях, но хотите откомпилировать проект с новыми опциями компилятора, например, изменить уровень оптимизации. Тогда все иные команды, кроме Build, не произведут повторной компиляции. Так что эта команда во многих случаях необходима.

Помимо описанных команд компиляции имеется еще две: Project | Make All Projects и Project | Build All Projects. Они подобны рассмотренным командам Make и Build, но при работе с группой проектов относятся не к одному, а ко всем проектам группы.

По умолчанию все команды компиляции в C++Builder 6 выполняются в фоновом режиме. Эта новая возможность, введенная начиная с C++Builder 5, позволяет осуществлять во время компиляции и компоновки любые другие работы в ИСР.

Это, конечно, удобно, но не всегда. Дело в том, что фоновая компиляция осуществляется медленнее. Кроме того, по завершении компиляции в фоновом режиме окно компилятора исчезает, и при этом не показываются результаты компиляции: прошла ли она успешно, или имеются замечания. Поэтому, если у вас нет каких-то работ в ИСР, которые можно выполнять во время компиляции, лучше отключить фоновый режим компиляции. Это можно сделать, выполнив команду Tools | Environment Options и выключив на странице Preferences опцию Background Compilation.

Если фоновый режим компиляции отключен, то после окончания компиляции рассмотренными командами (кроме Run) в окне рис. 1.2 во второй строке появляется одно из трех итоговых сообщений: "Done: Make" "Результат: выполнено", "Done: There are errors" "Результат: имеются ошибки", "Done: There are warnings" "Результат: имеются предупреждения".

Более подробные сведения о компиляции и компоновке проектов в C++Builder и рекомендации по настройке и ускорению этих процессов вы найдете в книге [1].

1.4 Директивы препроцессора

Обработка программы препроцессором происходит перед ее компиляцией. На этом этапе предварительной обработки вы можете выполнить следующие действия: включить в компилируемый файл другие файлы, определить *символические константы* и *макросы,* задать режим *условной компиляции* программного кода и *условного выполнения директив препроцессора.* Все директивы препроцессора начинаются с символа "#" и до начала директивы в строке могут находиться только символы пробела. Любая строка, начинающаяся с символа "#", воспринимается как директива препроцессора.

Точка с запятой в конце директивы препроцессора не ставится.

1.4.1 Директива #include

Директива **#include** применяется для включения копии указанного в директиве файла в то месте, где находится эта директива. Существуют три формы директивы **#include:**

```
#include <имя_файла>
tinclude "имя_файла"
#include идентификатор макроса
```

Последняя форма предполагает, что первый значащий символ после слова **include** не равен ни '<', ни ' ' '. Предполагается, что макрос, идентификатор которого используется в этой форме директивы, предварительно определен и использует одну из первых двух форм директивы **#include.**

Различие между первыми двумя формами директивы заключается в методе поиска препроцессором включаемого файла. Если имя файла заключено в угловые скобки "<" и ">", как это делается для включения заголовочных файлов стандартной библиотеки, то последовательность поиска препроцессором заданного файла в каталогах определяется заданными каталогами включения **(include directories).** Если же имя файла заключено в кавычки, препроцессор ищет файл, просматривая каталоги в следующей последовательности:

- каталог того файла, который содержит директиву **#include**
- каталоги файлов, которые включили в данный файл директивой **#include**
- текущий каталог

Вызов макроса осуществляется выражением:

```
идентификатор_макроса(аргументы)
```

Макрос, определяемый директивой препроцессора #define, это символическое имя некоторых операций. Как и в случае символических констант, идентификатор макроса заменяется на замещающий текст до начала компиляции программы. Но сначала в замещающий текст подставляются значения параметров, а затем уже этот расширенный макрос подставляется в текст вместо идентификатора макроса и списка его параметров.

Например, следующий макрос с одним параметром определяет площадь круга, воспринимая передаваемый в него параметр как радиус:

```
#define CIRC (x)  (3.14159 * (x) * (x))
```

Везде в тексте файла, где появится идентификатор CIRC (A), значение аргумента A будет использовано для замены x в замещающем тексте и этот расширенный текст макроса будет использован для замещения. Например, оператор с макросом в тексте программы

```
S = CIRC(4);
```

примет вид:

```
S = (3.14159 * (4) * (4));
```

Поскольку это выражение состоит только из констант, его значение будет вычислено во время компиляции и полученный результат будет присвоен переменной S во время выполнения программы.

Если вызов имеет вид

```
S = CIRC(a + b);
```

то после расширения макроса текст будет иметь вид:

```
S = (3.14159 * (a + b) * (a + b));
```

В данном случае аргумент макроса является выражением, содержащим переменные a и b. Поэтому вычисления будет осуществляться не во время компиляции, а во время выполнения программы.

Обратите внимание на круглые скобки вокруг каждого включения параметра x в тексте рассмотренного макроса и вокруг всего выражения. При вызове типа CIRC(4) они кажутся излишними. Но во втором примере вызова при отсутствии скобок расширение привело бы к оператору:

```
S = 3.14159 * a + b * a + b;
```

Тогда в соответствии со старшинством операций (см. разд. 1.9.16) сначала выполнилось бы умножение 3.14159 * a, затем b * a, а затем результаты этих умножений сложились бы друг с другом и с b. Конечно, результат вычислений был бы неверным.

Исходя из сказанного, можно посоветовать при объявлении макроса всегда заключать в скобки параметры в замещающем тексте и сам замещающий текст. Это избавит от возможных неприятностей, связанных с неверной последовательностью вычислений при расширении макроса.

Приведем еще один пример: макрос, определяющий площадь эллипса через значения его полуосей, может быть объявлен директивой

```
#define Ell(x,y)  (3.14159 * (x) * (y))
```

Вызов этого макроса может иметь вид:

```
S = Ell(R1, R2);
```

С точки зрения получаемых результатов вычислений макросы эквивалентны функциям. Например, вычисление площади круга можно было бы оформить функцией:

```
double circ(double x)
{
 return 3.14159 * x * x;
)
```

и вызывать ее оператором:

```
S = circ(a + b);
```

Таким образом, возникает вопрос, что выгоднее использовать: макросы или функции.

Вызов функции сопряжен с накладными расходами и затягивает выполнение программы. Это соображение работает в пользу использования макросов. С другой стороны, макрос расширяется во всех местах текста, где используется его вызов. Если таких мест в программе много, то это увеличивает размер текста и, соответственно, размер выполняемого модуля. Так что функции позволяют сокращать объем выполняемого файла, а макросы — сокращать скорость выполнения. Правда, макросы тоже могут быть связаны с дополнительными накладными расходами. В приведенном примере значение параметра a + b вычисляется дважды, в то время, как в функции это вычисление осуществляется только один раз. Конечно, для таких простых вычислений это не существенно. Но если в качестве параметра передается сложное выражение, обращающееся в свою очередь к каким-нибудь сложным функциям, то эти дополнительные накладные расходы могут стать заметными и затянуть вычисления.

Недостатком макросов является отсутствие встроенного контроля согласования типов аргументов и формальных параметров. Отсутствие соответствующих предупреждений компилятора может приводить к ошибкам программы, которые трудно отлавливать. Но наиболее существенный недостаток макросов — возможность появления побочных эффектов, если в качестве аргумента в макрос передается некоторое выражение. Например, если описанный выше макрос **CIRC,** вычисляющий площадь круга, вызвать следующим образом:

```
S = CIRC(a++)
```

предполагая рассчитать площадь и затем операцией постфиксного инкремента (см. разд. 1.9.2) увеличить радиус на 1, то макрос будет расширен так:

```
S = (3.14159 * (a++) * (a++));
```

При этом площадь будет вычислена верно, но постфиксный инкремент вычислится два раза. В результате значение радиуса a будет увеличено не на 1, а на 2.

Если же это макрос вызвать следующим образом:

```
S = CIRC(++a)
```

предполагая увеличить радиус на 1 и вычислить площадь круга с таким увеличенным радиусом, то макрос будет расширен так:

```
S = (3.14159 * (++a) * (++a));
```

При этом площадь будет определена неверно, так как в процессе вычислений радиус будет увеличен дважды и выражение окажется эквивалентным следующему:

```
S = (3.14159 * (a + 1) * (a + 2));
```

Всех этих побочных эффектов не будет, если вместо макроса использовать описанную выше функцию **circ.**

При выборе реализации вычислений функцией или макросом надо обеспечивать компромисс между скоростью вычислений и затратами памяти. Для небольших функций, возможно, наилучшим решением является применение встраиваемых функций **(inline** — см. разд. 1.7.6). Для них проблемы оптимальной реализации решает компилятор, и делает это он, вероятно, не хуже нас с вами.

1.4.2.3 Директива #undef

Определения символических констант и макросов могут быть аннулированы при помощи директивы препроцессора **#undef,** имеющей вид:

```
#undef идентификатор
```

Директива отменяет определение символической константы или макроса с указанным идентификатором. Таким образом, область действия символической константы или макроса начинается с места их определения и заканчивается явным их аннулированием директивой **#undef** или концом файла. После аннулирования соответствующий идентификатор может быть снова использован в директиве **#define.**

Например, возможен следующий код:

```
#define MyConst 128
// Здесь константа MyConst равна 128
...
#undef MyConst
// Здесь константу MyConst использовать нельзя
...
#define MyConst 64
// Здесь константа MyConst равна 64
...
```

1.4.3 Условная компиляция: директивы #if, #endif, #ifdef, tfifndef. #else, #elif

Условная компиляция дает возможность программисту управлять выполнением директив препроцессора и компиляцией программного кода. Каждая условная директива препроцессора вычисляет значение целочисленного константного выражения. Операции преобразования типов, операция **sizeof** и константы перечислимого типа не могут участвовать в выражениях, вычисляемых в директивах препроцессора.

Условная директива препроцессора #if во многом похожа на оператор **if.** Ее синтаксис имеет вид:

```
#if условие
  фрагмент кода
#endif
```

В этой записи условие является целочисленным выражением. Если это выражение возвращает не нуль (истинно), то фрагмент кода, заключенный между директивой #if и директивой **#endif,** компилируется. Если же выражение возвращает нуль (ложно), то этот фрагмент игнорируется и препроцессором, и компилятором.

В условиях, помимо обычных выражений, можно использовать конструкцию

```
defined идентификатор
```

defined возвращает 1, если указанный идентификатор ранее был определен директивой **#define,** и возвращает 0 в противном случае. Например, возможен следующий код:

```
#if defined Debug && !defined MyConst
фрагмент кода
#endif
```

Фрагмент кода будет выполняться, если ранее была записана директива

```
#define Debug
```

и не было директивы

```
#define MyConst
```

или эта директива была отменена директивой

```
#undef MyConst
```

Конструкция **#if defined** может быть заменена эквивалентной ей директивой **#ifdef,** а конструкция #if **!defined** — директивой **#ifndef.** Например, тексты

```
#ifdef Size
...
#endif
```

и

```
#if defined Size
...
#endif
```

эквивалентны.

Можно использовать более сложные конструкции условных директив препроцессора при помощи директив **#elif** (эквивалент **else if** в обычной структуре **if)** и **#else** (эквивалент else в структуре if). Например, в коде

```
#if условие 1
  фрагмент кода 1
#elif условие 2
  фрагмент кода 2
#else
  фрагмент кода 3
#endif
```

фрагмент кода 1 будет компилироваться, если выполняется условие 1, фрагмент кода 2 будет компилироваться, если выполняется условие 2, а фрагмент кода 3 будет компилироваться, если не выполняется ни одно из предыдущих условий.

Условная компиляция может быть полезна во многих случаях. Например, нередко в процессе отладки приложения в него полезно ввести различные отладочные печати, позволяющие следить за ходом выполнения программы. Если вы не хотите, чтобы эти печати оставались в окончательном варианте программы, вы можете в разных местах приложения ввести конструкции вида

```
#ifdef Debug
операторы отладки
#endif
```

Тогда, если в начале программы вы введете директиву

```
#define Debug
```

операторы отладки будут компилироваться и выполняться. Но когда вы уберете или закомментируете эту директиву **#define,** определяющую введенный вами идентификатор **Debug,** все операторы отладки исчезнут из текста. Можно поступить даже проще, ничего не изменяя в тексте, а оперируя опцией Conditionals на странице Directories/Conditionals диалогового окна, вызываемого командой Project | Options.

Конечно, вы могли бы поступить иначе: ввести переменную булева типа **Debug,** задать ей в начале выполнения приложения значение **true** и оформлять отладки следующим образом:

```
#if (Debug)
(
 операторы отладки
}
```

Если в дальнейшем заменить задаваемое значение **Debug** на **false,** то операторы отладки перестанут выполняться. Отличие этого подхода от использования директив препроцессора заключается в том, что коды операторов отладки в этом слу-

чае останутся в тексте программы, увеличивая размер выполняемого модуля. А директивы условной компиляции просто уберут отладочный код из программы.

Приведем еще один пример использования условной компиляции. Если вы взглянете на заголовочный файл любого модуля формы вашего приложения, то увидите, что C++Builder первыми операторами вставляет в него директивы вида:

```
ifndef Unit1H
define Unit1H
```

А завершается заголовочный файл директивой

```
#endif
```

Что это дает? Это позволяет исключить зацикливание при циклических директивах **#include,** включающих в различных модулях заголовочные файлы друг друга. Когда в приложение первый раз включается модуль **Unit1.h,** то выполняются указанные выше первые две директивы и идентификатор **UnitlH** оказывается определен. После этого компилируется текст файла. Но если в результате директив **#include** этот же файл будет включаться еще один раз, то обнаружится, что идентификатор **UnitlH** уже определен, и повторной компиляции файла не произойдет.

1.4.4 Директивы #error, #line, #pragma

Директива препроцессора **#error** имеет следующий синтаксис:

```
#error errmsg
```

Директива печатает в процессе компиляции сообщение об ошибке вида:

```
Error: filename line# : Error directive: errmsg
```

где errmsg — сообщение, заданное директивой #error. После печати этого сообщения компиляция прекращается.

Директива используется в сочетании с директивами условной компиляции и срабатывает при возникновении условий, не позволяющих продолжить работу. Например:

```
#ifndef Unit1H
#error Не найден файл Unit1.h
```

Директива препроцессора **#line** задает целочисленное константное начальное значение номера строки для нумерации следующих за директивой строк исходного текста программы. Возможны две формы директивы:

```
#line номер_строки
#line номер_строки "имя_файла"
```

Элемент директивы номер_строки задает начальное значение номера строки. Все последующие строки исходного текста программы будут нумероваться, начиная с этого номера. Если в директиву включено имя файла, то не только изменяется нумерация последующих строк программы, но и компилятор во всех своих сообщениях будет ссылаться на файл с указанным именем. Директива **#line** обычно используется для того, чтобы сделать сообщения о синтаксических ошибках и предупреждения компилятора более удобными для понимания. Номера строк не добавляются в исходный файл. Пример директивы:

```
#line 100 "Unit1.cpp"
```

Применение директивы **#line** делает работу с отладчиком C++Builder не очень удобной. При возникновении ошибки курсор в окне Редактора Кода останавливается не на строке с ошибкой, а на начале текущего файла или, если в директиве указано имя другого существующего файла, то на начале этого файла. Так что можно рекомендовать не использовать без особой надобности директиву **#line.**

Директива **#pragma** имеет следующий синтаксис:

```
#pragma имя опции
```

и вызывает действия, зависящие от указанной опции. Список возможных опций вы можете найти во встроенной справке C++Builder. Он довольно обширен и связан с различными режимами работы препроцессора.

Пример директивы **#pragma** вы можете видеть в любом модуле своего проекта. Первые две строки файла любого модуля имеют вид:

```
#include <vcl.h>
#pragma hdrstop
```

Здесь использована опция **hdrstop.** Она связана с особенностью работы препроцессора, производительность которого существенно повышается, если учитывается, что некоторое количество заголовочных файлов общее для всех модулей. Директива **#pragma hdrstop** указывает компилятору конец списка таких общих файлов. Так что надо следить за тем, чтобы не добавлять перед этой директивой включение каких-то заголовочных файлов, не являющихся общими для других модулей.

В файлах модулей вы можете увидеть еще две директивы **#pragma**:

```
#pragma package(smart_init)
#pragma resource "*.dfm"
```

Первая из них определяет последовательность инициализации пакетов такой, какая устанавливается взаимными ссылками использующих их модулей. Вторая говорит препроцессору, что для формы надо использовать файл **.dfm** с тем же именем, что и имя данного файла. Во избежание всяких неприятностей лучше не трогать и не изменять эти директивы.

Вы может включать в модуль директивы вида:

```
#pragma message(текст);
```

или

```
#pragma message(идентификатор);
```

где текст — произвольная строка, а идентификатор — некий идентификатор, объявленный предварительно директивой **#define.** Введенные с помощью **#pragma** message тексты появятся как сообщения компилятора во время компиляции вашего приложения и позволят проследить ход компиляции. Но для того, чтобы сообщения появились, надо выполнить команду Projects | Options, перейти в открывшемся диалоговом окне на страницу Compiler и включить индикатор Show general messages. Если вы это проделали и включили, например, в начале обработчика щелчка на кнопке **Button1** вашего приложение оператор

```
#pragma message("компилируется TForm1::Button1Click")
```

а в конце того же обработчика операторы

```
#define text "компиляция TForm1::Button1Click завершена"
#pragma message (text)
```

то при компиляции приложения в окне сообщений вы увидите строки:

```
[C++] Unit1.cpp(1):
[C++]  Loaded cached pre-compiled headers
[C++]  компилируется TForm1::Button1Click
[C++]  компиляция TForm1::Button1Click завершена
[Linker]
```

А если при компиляции обнаружились какие-то ошибки, они будут размещены между указанными вами сообщениями, так что вам проще будет понять, где эти ошибки.

В модуль можно также включать директивы

```
#pragma startup имя_функции приоритет
#pragma exit имя_функции приоритет
```

Первая из них указывает имя функции, которая должна вызываться в самом начале выполнения приложения, до того, как будет вызвана функция **WinMain** или **main.** Вторая указывает имя функции, которая должны вызываться перед завершением программы функцией **_exit.** Функции должны быть определены да появления в тексте этих директив. Точнее, реализованы они могут быть после, но тогда до директив должны быть расположены их прототипы.

Функции не должны принимать никаких параметров и не должны возвращать результат. Иначе говоря, они должны быть объявлены как

```
void func(void);
```

Параметр приоритет может не указываться. Тогда, если вы включите в модуль несколько директив, вызывающих различные функции, то функции, вызываемые перед началом выполнения, будут вызываться в той последовательности, в которой расположены директивы **#pragma startup.** Но эту последовательность можно изменить, задавая в директивах соответствующие приоритеты. Приоритеты задаются целыми числами от 64 до 255. Приоритеты от 0 да 63 используются библиотечными функциями C++ и в приложениях задаваться не должны.

Высший приоритет — 0. Приоритет по умолчанию — 100. Если заданы приоритеты, то функции с более высоким приоритетом перед началом выполнения вызываются первыми, а перед завершением выполнения — последними.

Рассматриваемые директивы можно использовать для начальной установки каких-то глобальных переменных, определения типов, каких-то запросов пользователю, а также для зачистки «мусора» при аварийном завершении приложения. Приведем чисто демонстрационный пример. Введите в приложение следующие операторы:

```
void f1(void);
void f2(void);
#pragma startup f1
#pragma startup f2

AnsiString Name="Неизвестный";
void f1(void)
{
 if (! InputQuery("Пожалуйста, представьтесь",
                  "Укажите, как в дальнейшем обращаться к Вам", Name))
  ShowMessage("Вы не представились, господин неизвестный");
 else ShowMessage("Здравтвуйте, господин " + Name + " !");
}
void f2(void)
{
 ShowMessage("Начало работы");
}
```

Выполните это приложение. Вы увидите, что прежде, чем откроется его главная форма, будет вызвана функция f1 и пользователю будет показано диалоговое окно с просьбой представиться. Вид этого окна вы можете посмотреть в гл. 4 в описании функции **InputQuery.** Введенное пользователем имя записывается в глобальную переменную **Name** и может использоваться при выполнении программы для формирования обращений к пользователю.

После вызова **f1** последует вызов функции **f2,** сообщающей о начале работы, и только после этого пользователь увидит главную форму приложения.

Если вы замените директиву вызова f1 на

```
#pragma startup f1 101
```

то сначала будет вызвана функция **f2,** имеющая по умолчанию приоритет 100, и только после этого вызовется функция **f1.**

1.4.5 Операции препроцессора # и

Операция препроцессора # применяется к параметрам макросов, представляющим собой лексемы (текст). Операция преобразует лексему в строку символов, взятую в кавычки. Например, если определен следующий макрос:

```
#define Pers(x) Label1->Caption = "Сотрудник " #x
```

и в тексте программы он вызван оператором

```
Pers(Иванов);
```

то он будет расширяться до

```
Labell->Caption = "Сотрудник " "Иванов"
```

Строка "Иванов" заменила параметр #x в замещающем тексте. Строки, разделенные символами пробела, сцепляются (склеиваются) во время предварительной обработки, так что вышеприведенный оператор эквивалентен оператору

```
Label1->Caption = "Сотрудник Иванов"
```

Операция ## выполняет конкатенацию (сцепление, склеивание) двух лексем. Например, если определен макрос

```
#define Concat(x,y) x ## y
```

то встреченное в тексте программы выражение Concat(Edit,l) будет преобразовано в Edit1.

1.5 Константы

1.5.1 Неименованные константы

Константы могут использоваться непосредственно в тексте программы в любых операторах и выражениях. Имеется 4 типа констант: целые, с плавающей запятой, символьные (включая строки) и перечислимые. Например: 25 и -5 — целые константы, 4.8, 5e15, 5E15, -5.1e8 — константы с плавающей запятой, 'A', '\0', '\n', '007' — символьные константы, "Это строка" - строковая константа.

Целые константы могут быть десятичные, восьмеричные и шестнадцатеричные. Восьмеричные начинаются с символа нуля, после которого следуют восьмеричные цифры (от 0 до 7). Например: 032. Запись константы вида 08 будет воспринята как ошибка, поскольку 8 не является восьмеричной цифрой. Восьмеричные константы не могут превышать значения 037777777777. Значения, большие этой величины, усекаются.

Шестнадцатеричные константы начинаются *с* символов нуля и X или x, после которых следуют Шестнадцатеричные цифры (от 0 до F, можно записывать в верхнем или нижнем регистрах). Например: 0XF01. Шестнадцатеричные константы не могут превышать значения 0xFFFFFFFF. Значения, большие этой величины, усекаются.

Символьные константы должны заключаться в одинарные кавычки. Эти константы хранятся как char, signed **char** или unsigned char.

Строковые константы заключаются в двойные кавычки. Они хранятся как последовательность символов, завершающаяся нулевым символом '\0'. Пустая строка содержит только нулевой символ.

Если две строковые константы разделены в тексте только пробельным символом, они склеиваются в одну строку. Например:

```
"Это начало строки, " "а это ее продолжение"
```

ИЛИ

```
"Это начало строки, "
"а это ее продолжение"
```

воспримутся как константа

```
"Это начало строки, а это ее продолжение"
```

Перенос длинной строки с одной строчки кода в другую можно делать не только так, как показано выше, но и помещая в конец первой строчки одиночный символ обратного слэша '\'. Например, запись

```
"Это начало строки, \
а это ее продолжение"
```

воспримется как одна строка.

В строковой константе можно использовать управляющие символы, предваряемые обратным слэшем. Например, константа

```
"\"Имя\"\t \tАдрес\nИванов\t \tМосква"
```

будет при отображении на экране выглядеть так

```
"Имя"          Адрес
Иванов         Москва
```

Кавычки после символа "\" воспринимаются как символ кавычек, а не как окончание строки. Символы "\t" и "\n" означают соответственно табуляцию и перевод строки.

Если в константу должен быть включен обратный слэш "\", то надо поместить подряд два слэша. Например, строка

```
"c:\\test\\test.cpp"
```

будет откомпилирована (если компиляция осуществляется с опцией **-A**) как

```
"c:\test\test.cpp"
```

Константы перечислимого типа объявляются следующим образом:

```
enum имя {значения};
```

Например, оператор

```
enum color { red, yellow, green };
```

объявляет переменную с именем **color,** которая может принимать константные значения **red, yellow** или **green.** Эти значения в дальнейшем можно использовать как константы для присваивания переменной **color** или для проверки ее значения. Этим константам соответствуют целые значения, определяемые их местом в списке объявления: **red** — 0, **yellow — 1, green** — 2. Эти значения можно изменить, если инициализировать константы явным образом. Например, объявление

```
enum color { red, yellow = 3, green = red + 1};
```

приведет к тому, что значения констант будут равны: **red** — 0, **yellow** — 3, **green — 1.** При этом не обязательно должна соблюдаться уникальность значений. Несколько констант в списке могут иметь одинаковые значения.

В C++Builder имеется ряд предопределенных констант, основные из которых **true** — истина, **false** — ложь, **NULL** — нулевой указатель.

1.5.2 Именованные константы

Именованная константа — это константа, которой присвоен некоторый идентификатор. Объявление именованной константы является указателем для компилятора заменить во всем тексте этот идентификатор значением константы. Такая замена производится только в процессе компиляции и не отражается на исходном тексте.

Цель объявления именованной константы — сделать текст более осмысленным и облегчить при необходимости изменение значения константы во всем тексте. Например, если в тексте многократно используется число 55, означающее максимально допустимое значение каких-то переменных, то проверки

```
if (N > NMax) ...
```

более понятны, чем

```
if (N > 55) ...
```

При необходимости сменить это число, проще изменить его в одном месте программы — в объявлении константы NMax, чем искать по всему тексту числа 55, которые, к тому же, в разных частях программы могут иметь разный смысл.

Именованные константы объявляются так же, как переменные (см. разд. 1.6.1), но с добавлением модификатора **const:**

```
const тип имя_константы = значение;
```

Например:

```
const float Pi = 3.14159;
```

В качестве значения константы можно указывать и константное выражение, содержащее ранее объявленные константы. Например, если вы объявили константу **Pi,** то далее можете объявить константы

```
const float Pi2 = 2 * Pi; //удвоенное число Пи
const float Kd = Pi/180;  // коэффициент пересчета градусов в радианы
```

Для целых констант тип можно не указывать:

```
const maxint = 12345;
```

Но будьте осторожны, не забывайте указывать тип для констант, тип которых отличен от **int.** Например, объявление "const Pi = 3.14159;" присвоит константе Pi значение 3, поскольку константа без указания типа считается целой.

Попытка где-то в тексте изменить значение именованной константы приведет к ошибке компиляции с выдачей соответствующего сообщения.

Приведем еще примеры объявления именованных констант:

```
char *const str1 = "Привет!";
char const *str2 = "Всем привет!";
```

Первое объявление вводит константу **str1,** являющуюся постоянным указателем на строку. Второе объявляет указатель **str2** на строковую константу. Этот указатель не является константой. Его в процессе выполнения программы можно изменить так, чтобы он указывал на другую строковую константу. Иначе говоря, оператор

```
str2 = str1;
```

допустим, а оператор

```
str1 = str2;
```

вызовет ошибку компиляции.

1.5.3 Объявленные (manifest) константы

В C++Builder предопределен ряд глобальных идентификаторов — макросов, называемых иногда объявленными **(manifest)** константами. Большинство из них начинаются с двух символов подчеркивания. В приведенной ниже таблице для большей наглядности и во избежание путаницы между этими символами подчеркивания введены пробелы, т.е. вместо (__) записано (___). В реальных идентификаторах этот пробел не должен фигурировать. Ниже приводится только часть объяв-

ленных констант. Остальные вы можете посмотреть во встроенной справке Borland C++Builder.

Макрос	Значение	Описание
_ _BCOPT_ _	1	Определен в любом компиляторе, производящем оптимизацию
_ _BCPLUSPLUS_ _	0x0530	Определен, если вы выбрали компиляцию C++; в последующих версиях значение будет увеличено
_ _BORLANDC_ _	0x0530	Номер версии
_ _CDECL_ _	1	Определен, если установлено соглашение вызова **cdecl**
_CHAR_UNSIGNED	1	Определен, если выбрана опция, что по умолчанию тип char эквивалентен unsigned char; при опции **–K** макрос не определен
_ _CONSOLE_ _		Определен для консольных приложений
_CPPUNWIND	1	По умолчанию определен и показывает, что доступно разматывание стека; при **–xd–** не определен
_ _cplusplus	1	Определен в режиме C++
_ _DATE_ _	строка	Дата компиляции исходного файла (строка в формате "Mmm dd yyyy", например, "Jan 19 1994")
_ _DLL_ _	1	Определен при использовании опции **–WD**
_ _FILE_ _	строка	Предполагаемое имя исходного файла.
_ _FLAT_ _	1	Определен при компиляции в модели памяти flat с разрядностью 32 бита
_ _LINE_ _	целое	Номер текущей строки исходного текста программы
_ _PASCAL_ _	1	Определен, если установлено соглашение вызова Pascal
_ _STDC_ _	1	Используется для указания, что данная реализация удовлетворяет стандартам ANSI. Определена, если вы компилировали с опцией **–A**
_ _TEMPLATES_ _	1	Определен для файлов C++, означает, что поддерживаются шаблоны
_ _TIME_ _	строка	Время компиляции исходного файла (символьная строка формата "hh:mm:ss")
_ _WIN32_ _	1	Определен для приложений консольных и GUI

Макросы _ _DATE_ _, _ _FILE_ _, _ _LINE_ _, _ _STDC_ _ и _ _TIME_ _ не должны появляться в файле непосредственно за директивами #define и #undef.

1.6 Переменные

1.6.1 Объявление переменных

Переменная является идентификатором, обозначающим некоторую область в памяти, в которой хранится значение переменной. Это значение может изменяться во время выполнения приложения.

Объявление переменной имеет вид:

```
тип список_идентификаторов_переменных;
```

Список идентификаторов может состоять из идентификаторов переменных, разделенных запятыми. Например:

```
int x1, x2;
```

Одновременно с объявлением некоторые или все переменные могут быть инициализированы. Например:

```
int x1 = 1, x2 = 2;
```

Для инициализации можно использовать не только константы, но и произвольные выражения, содержащие объявленные ранее константы и переменные. Например:

```
int x1 = 1, x2 = 2 * x1;
```

Объявление переменных может быть отдельным оператором или делаться внутри таких операторов, как, например, оператор цикла:

```
for ( int i = 0; i < 10; i++)
```

1.6.2 Классы памяти

Каждая переменная характеризуется некоторым *классом памяти,* который определяет ее *время жизни* — период, в течение которого эта переменная существует в памяти. Одни переменные существуют недолго, другие — неоднократно создаются и уничтожаются, третьи — существуют на протяжении всего времени выполнения программы.

В C++Builder имеется четыре спецификации класса памяти: **auto,** register, **extern и static.** Спецификация класса памяти идентификатора определяет его класс памяти, область действия и пространство имен.

Областью действия (областью видимости) идентификатора называется область программы, в которой на данную переменную (как, впрочем, и на любой идентификатор — константу, функцию и т.п.) можно сослаться. На некоторые переменные можно сослаться в любом месте программы, тогда как на другие — только в определенных ее частях.

Класс памяти определяется, в частности, местом объявления переменной. *Локальные переменные* объявляются внутри некоторого блока или функции. Эти переменные видны только в пределах того блока, в котором они объявлены. *Блоком* называется фрагмент кода, ограниченный фигурными скобками "{}". *Глобальные переменные* объявляются вне какого-либо блока или функции.

Спецификации класса памяти могут быть разбиты на два класса: *автоматический класс памяти* с *локальным временем жизни* и *статический класс памяти с глобальным временем жизни.* Ключевые слова **auto и register** используются для объявления переменных с локальным временем жизни. Эти спецификации применимы только к локальным переменным. Локальные переменные создаются при входе в блок, в котором они объявлены, существуют лишь во время активности блока и исчезают при выходе из блока.

Спецификация **auto,** как и другие спецификации, может указываться перед типом в объявлении переменных. Например:

```
auto float x, y;
```

Локальные переменные являются переменными с локальным временем жизни по умолчанию, так что ключевое слово **auto** используется редко. Далее мы будем ссылаться на переменные автоматического класса памяти просто как на автоматические переменные.

Пусть, например, имеется следующий фрагмент кода:

```
{
 int i = 1;
 ...
 i++;
 ...
}
```

к которому в ходе работы программы происходит неоднократное обращение. При каждом таком обращении переменная i будет создаваться заново (под нее будет выделяться память) и будет инициализироваться единицей. Затем в ходе работы программы ее значение будет увеличиваться на 1 операцией инкремента. В конце выполнения этого блока переменная исчезнет и выделенная под нее память освободится. Следовательно, в такой локальной переменной невозможно хранить какую-то информацию между двумя обращениями к блоку.

Спецификация класса памяти **register** может быть помещена перед объявлением автоматической переменной, чтобы компилятор сохранял переменную не в памяти, а в одном из высокоскоростных аппаратных регистров компьютера. Например:

```
register int i = 1;
```

Если интенсивно используемые переменные, такие как счетчики или суммы, могут сохраняться в аппаратных регистрах, накладные расходы на повторную загрузку переменных из памяти в регистр и обратную загрузку результата в память могут быть исключены. Это сокращает время вычислений.

Компилятор может проигнорировать объявления **register.** Например, может оказаться недостаточным количество регистров, доступных компилятору для использования. К тому же оптимизирующий компилятор способен распознавать часто используемые переменные и решать, помещать их в регистры или нет. Так что явное объявление спецификации **register** используется редко.

Обсуждая способность компилятора по своему разумению помещать переменные в регистры, надо сказать, что могут быть ситуации, когда это недопустимо. Например, если переменная может асинхронно изменяться в процессе выполнения каким-то фоновым процессом. Такие переменные надо помечать модификатором **volatile.** Например:

```
volatile int tik;
```

Модификатор **volatile** указывает компилятору, что переменная может изменяться каким-то другим процессом. Например, это может быть связано с отлавливанием каких-то прерываний, с сообщениями, поступающими от портов ввода, с параллельно выполняемой нитью многопоточного процесса. Компилятор не должен помещать такую процедуру в регистры и не должен осуществлять проверку ее значений, так как они могут изменяться.

C++ расширил действие модификатора **volatile** на классы (см. разд. 2.14) и на их функции-элементы. Чтобы не возвращаться к этому модификатору, отмечу, что если объект объявлен **volatile,** то он может использовать только функции-элементы, также объявленные **volatile.**

Ключевые слова **extern и static** используются, чтобы объявить идентификаторы переменных как идентификаторы статического класса памяти с глобальным временем жизни. Такие переменные существуют с момента начала выполнения программы. Для таких переменных память выделяется и инициализируется сразу после начала выполнения программы.

Существует два типа переменных статического класса памяти: глобальные переменные и локальные переменные, объявленные спецификацией класса памяти **static.** Глобальные переменные по умолчанию относятся к классу памяти extern. Глобальные переменные создаются путем размещения их объявлений вне описания какой-либо функции и сохраняют свои значения в течение всего времени выполнения программы. На глобальные переменные может ссылаться любая функция, которая расположена после их объявления или описания в файле.

Переменные, используемые только в отдельной функции, предпочтительнее объявлять как локальные переменные этой функции, а не как глобальные переменные. Это облегчает чтение программы и позволяет избежать случайного доступа к таким переменным других функций.

Локальные переменные, объявленные с ключевым словом **static,** известны только в том блоке, в котором они определены. Но в отличие от автоматических переменных, локальные переменные **static** сохраняют свои значения в течение всего времени выполнения программы. При каждом следующем обращении к этому блоку локальные переменные содержат те значения, которые они имели при предыдущем обращении.

Вернемся к уже рассмотренному выше примеру, но укажем для переменной i статический класс:

```
{
 static int i = 1;
 ...
 i++;
 ...
}
```

Инициализация переменной i произойдет только один раз за время выполнения программы. При первом обращении к этому блоку значение переменной i будет равно 1. К концу выполнения блока ее значение станет равно 2. При следующем обращении к блоку это значение сохранится и при окончании повторного выполнения блока i будет равно 3. Таким образом, статическая переменная способна хранить информацию между обращениями к блоку и, следовательно, может использоваться, например, как счетчик числа обращений.

Все числовые переменные статического класса памяти принимают нулевые начальные значения, если программист явно не указал другие начальные значения. Статические переменные — указатели, тоже имеют нулевые начальные значения.

Спецификации класса памяти **extern** используются в программах с несколькими файлами. Пусть, например, в модуле **Unit1** в файле **Unitl.cpp** или **Unitl.h** (это безразлично) объявлена глобальная переменная

```
int a = 5;
```

Тогда, если в другом модуле **Unit2** в файле **Unit2.cpp или Unit2.h** объявлена глобальная переменная

```
extern int a;
```

то компилятор понимает, что речь идет об одной и той же переменной. И оба модуля могут с ней работать. Для этого даже нет необходимости связывать эти модули директивой **#include** (см. разд. 1.4.1), включающей в модуль **Unitl** заголовочный файл второго модуля.

Подробнее области видимости переменных рассмотрены в разд. 1.8.

1.7 Функции

1.7.1 Объявление и описание функций

Функции представляют собой программные блоки, которые могут вызываться из разных частей программы. При вызове в них передаются некоторые переменные, константы, выражения, являющиеся аргументами, которые в самих процедурах и функциях воспринимаются как формальные параметры. При этом функции возвращают значение определенного типа, которое замещает в вызвавшем выражении имя вызванной функции.

Например, оператор

```
I = 5 * F(X);
```

вызывает функцию F с аргументом X, умножает возвращенное ею значение на 5 и присваивает результат переменной I.

Допускается также вызов функции, не использующий возвращаемого ею значения. Например:

```
F(X);
```

В этом случае возвращаемое функцией значение игнорируется.

Функция описывается следующим образом:

```
тип_возвращаемого_значения имя_функции(список_параметров)
(
  операторы тела функции
}
```

Первая строка этого *описания,* содержащая тип возвращаемого значения, имя функции и список параметров, называется *заголовком* функции. Тип возвращаемого значения может быть любым, кроме массива и функции. Могут быть также функции, не возвращающие никакого значения. В заголовке таких функций тип возвращаемого значения объявляется **void.**

Если тип возвращаемого значения не указан, он по умолчанию считается равным **int.** Впрочем, можно посоветовать не злоупотреблять этой возможностью. Лучше всегда указывать тип возвращаемого функцией значения, кроме главной функции **main.** Указание типа делает программу более наглядной и предотвращает возможные ошибки, связанные с неправильным преобразованием типов.

Список параметров, заключаемый в скобки, в простейшем случае (более сложные формы задания списка параметров будут рассмотрены позднее) представляет собой разделяемый запятыми список вида

```
тип параметра иденификатор_параметра
```

Например, заголовок:

```
double FSum(double X1,double X2, int A)
```

объявляет функцию с именем **FSum,** с тремя параметрами **X1,** X2 и A, из которых первые два имеют тип **double,** а последний — int. Тип возвращаемого результата — **double.** Имена параметров **X1,** X2 и A — локальные, т.е. они имеют значение только внутри данной функции и никак не связаны с именами аргументов, переданных при вызове функции. Значения этих параметров в начале выполнения функции равны значениям аргументов на момент вызова функции. Подробнее эти вопросы будут рассмотрены в разд. 1.7.2.

Ниже приведен заголовок функции, не возвращающей никакого значения:

```
void SPrint(AnsiString S)
```

Она принимает один параметр типа строки и, например, отображает его в каком-нибудь окне приложения.

Если функция не принимает никаких параметров, то скобки или оставляются пустыми, или в них записывается ключевое слово **void.** Например:

```
void F1(void)
```

или

```
void F1()
```

Первая из приведенных записей предпочтительнее, так как делает программу более переносимой.

Обратите внимание на то, что роль пустого списка параметров функции в C++ существенно отличается от аналогичного списка в языке C. В C это означает, что все проверки аргументов отсутствуют (т.е. вызов функции может передать любой аргумент, который требуется). А в C++ пустой список означает отсутствие аргументов. Таким образом, программа на C, использующая эту особенность, может сообщить о синтаксической ошибке при компиляции в C++.

Как правило (хотя формально не обязательно), помимо описания функции в текст программы включается также *прототип* функции — ее предварительное объявление. Прототип представляет собой тот же заголовок функции, но с точкой с запятой ";" в конце. Кроме того, в прототипе можно не указывать имена параметров. Если вы все-таки указываете имена, то их областью действия является только этот прототип функции. Вы можете использовать те же идентификаторы в любом месте программы в любом качестве. Таким образом, указание имен параметров в прототипе обычно преследует только одну цель — документирование программы, напоминание вам или сопровождающему программу человеку, какой параметр что именно обозначает.

Примеры прототипов приведенных выше заголовков функций:

```
double FSum(double X1,double X2, int A);
void SPrint(AnsiString S);
void F1(void);
```

или

```
double FSum(double, double, int);
void SPrint(AnsiString);
void F1();
```

Введение в программу прототипов функций преследует несколько целей. Во-первых, это позволяет использовать в данном модуле функцию, описанную в каком-нибудь другом модуле. Тогда из прототипа компилятор получает сведения, сколько параметров, какого типа и в какой последовательности получает данная функция. Во-вторых, если в начале модуля вы определили прототипы функций, то последовательность размещения в модуле описания функций безразлична. При отсутствии прототипов любая используемая функция должна быть описана до ее первого вызова в тексте. Это прибавляет вам хлопот, а иногда при взаимных вызовах функций друг из друга вообще невозможно. И, наконец, прототипы, размещенные в одном месте (обычно в начале модуля), делают программу более наглядной и самодокументированной. Особенно в случае, если вы снабжаете прототипы хотя бы краткими комментариями.

Если предполагается, что какие-то из описанных в модуле функций могут использоваться *в* других модулях, прототипы этих функций следует включать в заголовочный файл. Тогда в модулях, использующих данные функции, достаточно будет написать директиву **#include** (см. разд. 1.4.1), включающую данный заголовочный файл, и не надо будет повторять прототипы функций.

Обычно функции принимают указанное в прототипе число параметров указанных типов. Однако могут быть функции, принимающие различное число параметров (например, библиотечная функция printf) или параметры неопределенных за-

ранее типов. В этом случае в прототипе вместо неизвестного числа параметров или вместо параметров неизвестного типа ставится многоточие "...". Многоточие может помещаться только в конце списка параметров после известного числа параметров известного типа или полностью заменять список параметров. Например:

```
int prf(char *format, ...);
```

Функция с подобным прототипом принимает один параметр **format** типа **char** * (например, строку форматирования) и произвольное число параметров произвольного типа. Функция с прототипом

```
void Fp(...);
```

может принимать произвольное число параметров произвольного типа.

Если в прототипе встречается многоточие, то типы соответствующих параметров и их количество компилятором не проверяются.

Объявлению функции могут предшествовать спецификаторы класса памяти **extern** или **static.** Спецификатор **extern** предполагается по умолчанию, так что записывать его не имеет смысла. К функциям, объявленным как **extern,** можно получить доступ из других модулей программы (см. заключительную часть разд. 1.8.1). Если же объявить функцию со спецификатором **static,** например

```
static void F(void);
```

то доступ к ней из других модулей невозможен. Это надо использовать в крупных проектах во избежание недоразумений при случайных совпадениях имен функций в различных модулях.

Теперь рассмотрим описание тела функции. Тело функции пишется по тем же правилам, что и любой код программы, и может содержать объявления типов, констант, переменных и любые выполняемые операторы. Не допускается объявление и описание в теле других функций. Таким образом, функции не могут быть вложены друг в друга.

Надо иметь в виду, что все объявления в теле функции носят локальный характер. Объявленные переменные доступны только внутри данной функции. Если их идентификаторы совпадают с идентификаторами каких-то глобальных переменных модуля, то эти внешние переменные становятся невидимыми и недоступными. В этих случаях получить доступ к глобальной переменной можно, поставив перед ее именем два двоеточия "::", т.е. применив унарную операцию разрешения области действия.

Локальные переменные не просто видны только в теле функции, но по умолчанию они и существуют только внутри функции, создаваясь в момент вызова функции и уничтожаясь в момент выхода из функции. Если требуется этого избежать, соответствующие переменные должны объявляться со спецификацией **static** (подробнее см. в разд. 1.6.2).

Выход из функции может осуществляться следующими способами. Если функция не должна возвращать никакого значения, то выход из нее происходит или по достижении закрывающей ее тело фигурной скобки, или при выполнении оператора **return.** Если же функция должна возвращать некоторое значение, то нормальный выход из нее осуществляется оператором

```
return выражение
```

где выражение должно формировать возвращаемое значение и соответствовать типу, объявленному в заголовке функции.

Например:

```
double FSum(double X1,double X2, int A)
{
  return A * (X1 + X2);
}
```

Ниже приведен пример функции, не возвращающей никакого значения:

```
void SPrint(AnsiString S)
{
 if (S != "")
   ShowMessage(S);
}
```

Здесь возврат из функции происходит по достижении закрывающейся фигурной скобки тела функции. Приведем вариант той же функции, использующий оператор return:

```
void SPrint(AnsiString S)
{
 if (S == "") return;
 ShowMessage (S);
}
```

Прервать выполнение функции можно также генерацией какого-то исключения (см. разд. 1.12.6). Наиболее часто в этих целях используется процедура **Abort,** генерирующая "молчаливое" исключение **EAbort,** не связанное с каким-то сообщением об ошибке. Если в программе не предусмотрен перехват этого исключения, то применение функции **Abort** выводит управление сразу наверх из всех вложенных друг в друга вызовов функций.

Возвращаемое функцией значение может включать в себя вызов каких-то функций. В том числе функция может вызывать и саму себя, т.е. допускается рекурсия. В качестве примера приведем функцию, рекурсивно вычисляющую факториал. Как известно, значение факториала равно n! = n x (n-1) x (n-2) x ... x 1, причем считается, что 1! = 1 и 0! = 1. Факториал можно вычислить с помощью простого цикла **for** (и это, конечно, проще). Но можно факториал вычислять и с помощью рекуррентного соотношения n! = n x (n-1)!. Для иллюстрации рекурсии воспользуемся именно этим соотношением. Тогда функция **factorial** вычисления факториала может быть описана следующим образом:

```
unsigned long factorial (unsigned long n)
{
   if (n <= 1)
         return 1;
   else
         return n * factorial(n — 1);
}
```

Если значение параметра n равно 0 или 1, то функция возвращает значение 1. В противном случае функция умножает текущее значение n на результат, возвращаемый вызовом той же функции **factorial,** но со значением параметра n, уменьшенным на единицу. Поскольку при каждом вызове значение параметра уменьшается, рано или поздно оно станет равно 1. После этого цепочка рекурсивных вызовов начнет свертываться и в конце концов вернет значение факториала.

1.7.2 Передача параметров в функции по значению и по ссылке

Список параметров, передаваемый в функции, как было показано в предыдущем разделе, состоит из имен параметров и указаний на их тип. Например, в заголовке

```
double FSum(double X1,double X2, int A)
```

указано три параметра **X1,** X2, A и определены их типы. Вызов такой функции может иметь вид:

```
Pr(Y, X2, 5);
```

Это только один из способов передачи параметров в процедуру, называемый *передачей по значению.* Работает он так. В момент вызова функции в памяти создаются временные переменные с именами X1, X2, A, и в них копируются значения аргументов Y, X2 и константы *5.* На этом связь между аргументами и переменными X1, X2, A разрывается. Вы можете изменять внутри процедуры значения X1, X2 и A, но это никак не отразится на значениях аргументов. Аргументы при этом надежно защищены от непреднамеренного изменения своих значений вызванной функцией. Это предотвращает случайные побочные эффекты, которые так сильно мешают иногда созданию корректного и надежного программного обеспечения.

К недостаткам такой передачи параметров по значению относятся затраты времени на копирование значений и затраты памяти для хранения копии. Если речь идет о какой-то переменной простого типа, это, конечно, не существенно. Но если, например, аргумент — массив из тысяч элементов, то соображения затрат времени и памяти могут стать существенными.

Еще одним недостатком передачи параметров по значению является невозможность из функций изменять значения некоторых аргументов, что во многих случаях очень желательно.

Возможен и другой способ передачи параметров — *вызов по ссылке.* В случае вызова по ссылке оператор вызова дает вызываемой функции возможность прямого доступа к передаваемым данным, а также возможность изменения этих данных. Вызов по ссылке хорош в смысле производительности, потому что он исключает накладные расходы на копирование больших объемов данных; в то же время он может ослабить защищенность, потому что вызываемая функция может испортить передаваемые в нее данные.

Вызов по ссылке можно осуществить двумя способами: с помощью ссылочных параметров и с помощью указателей. Ссылочный параметр — это псевдоним соответствующего аргумента. Чтобы показать, что параметр функции передан по ссылке, после типа параметра в прототипе функции ставится символ амперсанда "&"; такое же обозначение используется в списке типов параметров в заголовке функции. Перед амперсандом и после него могут вставляться пробельные символы. Например, идентичные объявления

```
int &count
int & count
int& count
```

в списке параметров заголовка функции могут читаться как "count является ссылкой на int". В вызове такой функции достаточно указать имя переменной, и она будет передана по ссылке. Реально в функцию передается не сама переменная, а ее адрес, полученный операцией адресации "&". Тогда упоминание в теле вызываемой функции переменной по имени ее параметра в действительности является обращением к исходной переменной в вызывающей функции и эта исходная переменная может быть изменена непосредственно вызываемой функцией.

Например:

```
void square(int &);       // Прототип функции вычисления квадрата

void square(int &a)       // Заголовок функции
{
   a *= a;                      // Изменение значения параметра
}
```

Вызываться подобная функция может обычным способом передачей в нее имени аргумента. Например:

```
int x1 = 2;
square(x1);
```

В результате подобного вызова переменная **xl** получит значение 4.

Поскольку ссылочные параметры упоминаются в теле вызываемой функции просто по имени, программист может нечаянно принять ссылочный параметр за параметр, переданный по значению. Это может привести к неприятным ошибкам, если исходные значения переменных изменяются вызывающей функцией.

Альтернативной формой передачи параметра по ссылке является использование указателей (см. разд. 2.8 гл. 2). Тогда адрес переменной передается в функцию не операцией адресации "&", а операцией косвенной адресации "*". В списке параметров подобной функции перед именем переменной указывается символ "*", свидетельствуя о том, что передается не сама переменная, а указатель на нее. В теле функции тоже перед именем параметра ставится символ операции разыменования "*", чтобы получить доступ через указатель к значению переменной (пояснения всего этого вы можете найти в разд. 2.8 гл. 2). А при вызове функции в нее в качестве аргумента должна передаваться не сама переменная, а ее адрес, получаемый с помощью операции адресации "&".

Приведем пример той же рассмотренной ранее функции square, но с передачей параметра по ссылке с помощью указателя:

```
void square(int *);     // Прототип функции вычисления квадрата

void square(int *a)     // Заголовок функции

{
   *a *= *a;            // Изменение значения параметра
}
```

Вызов подобной функции может осуществляться, например, следующим образом:

```
int x1 = 2;
square(&x1);
```

1.7.3 Применение при передаче параметров спецификации const

В предыдущем разделе была рассмотрена передача параметров по ссылке. Она решает сразу две задачи: исключает накладные расходы, связанные с копированием передаваемых значений, и дает функции доступ для изменения значений передаваемых аргументов. Однако иногда требуется решать только первую задачу: избавиться от копирования громоздких аргументов типа больших массивов. Но при этом не требуется позволять функции изменять значения аргументов.

Это может быть осуществлено передачей в функцию аргументов как констант. Для этого перед соответствующими переменными в списке ставится ключевое слово const.

При использовании ссылочного параметра заголовок функции (именно заголовок описания, поскольку в прототипе спецификатор const указывать не обязательно) может иметь следующий вид:

```
double F(const &A)
```

В этом случае аргумент A не будет копироваться при вызове функции, но внутри функции изменить значение A будет невозможно. При попытке сделать такое изменение компилятор выдаст сообщение: "Cannot modify a const object".

Подобная передача параметра как константы позволяет сделать код более эффективным, так как при этом компилятору заведомо известно, что никакие изменения параметра невозможны.

При использовании указателей для передачи параметров в функцию возможны четыре варианта: неконстантный указатель на неконстантные данные, неконстантный указатель на константные данные, константный указатель на некон-

стантные данные и константный указатель на константные данные. Каждая комбинация обеспечивает доступ с разным уровнем привилегий.

Наивысший уровень доступа предоставляется неконстантным указателем на неконстантные данные — данные можно модифицировать посредством разыменования указателя, а сам указатель может быть модифицирован, чтобы он указывал на другие данные. Это описанная в предыдущем разделе передача параметров по ссылке с помощью указателя. В этом варианте передачи параметров спецификатор const не используется.

Неконстантный указатель на константные данные — это указатель, который можно модифицировать, чтобы указывать на любые элементы данных подходящего типа, но сами данные, на которые он ссылается, не могут быть модифицированы. Например, прототип:

```
void F(const char *sPtr);
```

объявляет функцию, в которую передается указатель **sPtr,** указывающий на константные данные типа const **char** * — в данном случае строку (массив символов). В теле функции такой указатель можно менять, перемещая его с одного обрабатываемого символа на другой. Но сами элементы строки (массива) изменять невозможно, так как они объявлены константными. Таким образом, исходные значения предохраняются от их несанкционированного изменения.

• Константный указатель на неконстантные данные — это указатель, который всегда указывает на одну и ту же ячейку памяти, данные в которой можно модифицировать посредством указателя. Этот вариант, например, реализуется по умолчанию для имени массива. Имя массива — это константный указатель на начало массива. Используя имя массива и индексы массива можно обращаться ко всем данным в массиве и изменять их. Прототип функции с передачей константного указателя на неконстантные данные может иметь вид:

```
void F( char *const sPtr);
```

Наименьший уровень привилегий доступа предоставляет константный указатель на константные данные. Такой указатель всегда указывает на одну и ту же ячейку памяти и данные в этой ячейке нельзя модифицировать. Это выглядит так, как если бы массив нужно было передать функции, которая только просматривает массив, использует его индексы, но не модифицирует сам массив. Прототип функции с подобной передачей параметра может иметь вид:

```
void F (const char *const sPtr);
```

1.7.4 Параметры со значениями по умолчанию

Обычно при вызове функции в нее передается конкретное значение каждого параметра. Но программист может указать, что параметр является параметром по умолчанию, и приписать этому параметру значение по умолчанию. Делается это заданием в заголовке функции после имени параметра символа "=", после которого записывается значение по умолчанию. Пусть, например, вы хотите написать функцию, которая рассчитывает суммарную силу, действующую на тело объемом V с плотностью P, погруженное в жидкость (например, воду) с плотностью PH2O. Как известно, формула, выражающая эту суммарную силу, направленную вверх (если ответ будет отрицательным, значит сила направлена вниз — тело тонет), следующая: F = G * V * (P — PH2O), где G — ускорение свободного падения.

Функцию, определяющую эту силу, можно описать следующим образом:

```
double Arh(double V = 1,double P = 0.5,double PH20 = 1,double G = 9.81)
( return G * V * (PH20 - P); }
```

Здесь всем параметрам даны значения по умолчанию. Объем V по умолчанию принят равным 1 м^3, плотность тела P по умолчанию равна 0,5 т/м^3 (плотность не-

которых пород дерева), плотность воды RH2O принята по умолчанию равной 1 т/м^3, а ускорение свободного падения G принято равным 9,81 м/с^2.

Если при вызове функции параметр по умолчанию не указан, то в функцию автоматически передается его значение по умолчанию. Например, если вызвать приведенную функцию оператором

```
F = Arh();
```

то значение F будет равно силе при значениях всех параметров по умолчанию.

Аргументы по умолчанию должны быть самыми правыми (последними) аргументами в списке параметров функции. Если вызывается функция с двумя или более параметрами по умолчанию и если пропущенный параметр не является самым правым в списке, то все параметры справа от пропущенного тоже пропускаются.

Например, вызов той же функции оператором

```
F = Arh(2);
```

позволяет рассчитать силу, действующую на тело объемом 2 м^3 при значениях всех остальных параметров по умолчанию. Вызов функции оператором

```
F = Arh(2,2.6);
```

позволяет рассчитать силу, действующую на алюминиевое (плотность 2,6 т/м^3) тело объемом 2 м^3 при значениях остальных параметров по умолчанию. Аналогично, задав при вызове три параметра можно рассчитать силу, действующую на тело, погруженное в жидкость другой плотности, а задав все четыре параметра можно определить силу, действующую на тело при эксперименте, проводящемся не на уровне моря (при этом изменится ускорение свободного падения).

Этот пример показывает, что последними в списке параметров со значениями по умолчанию надо указывать те параметры, значения которых в реальных задачах чаще всего остаются равными заданным по умолчанию.

Пропускать при вызове можно только некоторое число последних параметров в списке. Например, нельзя вызвать функцию таким образом:

```
F = Arh(2,,1.1);   // Ошибочный вызов
```

Параметры по умолчанию должны быть указаны при первом упоминании имени функции — обычно в прототипе. Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций.

1.7.5 Передача в функции переменного числа параметров

Иногда в функции требуется передавать некоторое число фиксированных параметров плюс неопределенное число дополнительных параметров. В этом случае заголовок функции имеет вид:

```
тип имя_функции(список_аргументов,...)
```

В данном случае список аргументов включает в себя конечное число обязательных аргументов (этот список не может быть пустым), после которого ставится многоточие на месте неопределенного числа параметров. Для работы с этими параметрами в файле stdarg.h определен тип списка va_list и три макроса: va_start, **va_arg** и **va_end.**

Макрос va_start имеет синтаксис:

```
void va_start(va_list ap, lastfix)
```

Этот макрос начинает работу со списком, устанавливая его указатель ap на первый передаваемый в функцию аргумент из списка с неопределенным числом аргументов. Параметр lastfix — это имя последнего из обязательных аргументов функции.

Макрос va_arg имеет синтаксис:

```
type va_arg(va_list ap, type)
```

Макрос возвращает значение очередного аргумента из списка. Параметр type указывает тип аргумента. Перед вызовом **va_arg** значение ap должно быть установлено вызовом va_start или va_arg. Каждый вызов **va_arg** переводит указатель ap на следующий аргумент.

Макрос va_end имеет синтаксис:

```
void va_end(va_list ap)
```

Макрос завершает работу со списком, освобождая память. Он должен вызываться после того, как с помощью **va_arg** прочитан весь список аргументов. В противном случае могут быть непредсказуемые последствия.

Рассмотрим пример. Пусть требуется создать функцию average, которая рассчитывает и отображает в метке Label1 среднее значение передаваемых в нее целых положительных чисел. Функция принимает в качестве первого аргумента некоторое сообщение, которое должно отображаться перед результатами расчета. Список обрабатываемых чисел может быть любой длины и заканчиваться нулем. Такая функция может быть реализована следующим образом:

```
#include <stdarg.h>
...
void average(AnsiString mess,...)
{
 double A = 0;
 int i = 0,arg;
 va_list ap;
 va_start(ap,  mess);
 while ((arg = va_arg(ap,int)) != 0)
 {
  i++;
  A += arg;
 }
 Form1->Label1->Caption = mess + "N = " +IntToStr(i) +
                          ", среднее = "+FloatToStr(A/i);
 va_end(ap);
}
```

Вызов функции может быть, например, таким:

```
average("Результаты экзамена: ",4,2,3,5,4,0);
```

В результате функция выдаст в метку Label1 сообщение:

```
Результаты экзамена: N = 5, среднее = 3,6
```

Функцию average можно было бы организовать иначе, не вводя специальную конечную метку в список (в приведенном примере — 0), а предваряя список аргументов параметром N, указывающим размер списка:

```
void average(AnsiString mess,int N,...)
{
 double A = 0;
 valistap;
 va_start(ap, N);
 for (int i = 0; i < N; i++)
  A += va_arg(ap,int);
 Form1->Label1->Caption = mess + "N = " +IntToStr(N) +
                          ", среднее = "+FloatToStr(A/N);
 va_end(ap);
}
```

Вызов функции может быть, например, таким:

```
, average("Результаты экзамена: ",5,4,2,3,5,4);
```

1.7.6 Встраиваемые функции inline

Реализация программы как набора функций хороша с точки зрения разработки программного обеспечения, но вызовы функций приводят к накладным расходам во время выполнения. В C++ для снижения этих накладных расходов на вызовы функций — особенно небольших функций — предусмотрены *встраиваемые (inline) функции.* Спецификация inline перед указанием типа результата в объявлении функции «советует» компилятору сгенерировать копию кода функции в соответствующем месте, чтобы избежать вызова этой функции. Это эквивалентно объявлению соответствующего макроса (см. разд. 1.4.2.2). В результате получается множество копий кода функции, вставленных в программу, вместо единственной копии, которой передается управление при каждом вызове функции.

Компилятор может игнорировать спецификацию inline, что обычно и делает для всех функций, кроме самых небольших.

Спецификацию inline целесообразно применять только для небольших и часто используемых функций. Использование функций inline может уменьшить время выполнения программы, но при этом может увеличить ее размер. В целом, применение функций inline предпочтительнее объявления макросов, поскольку в данном случае вы даете возможность компилятору оптимизировать код.

При использовании функции inline надо учитывать, что внесение в нее каких-то изменений может потребовать перекомпиляции всех «потребителей» этой функции — всех модулей, в которых она вызывается. Это может оказаться существенным моментом для развития и поддержки некоторых больших программ.

Приведем пример использования спецификации inline. Пусть, например, вам во многих частях программы приходится вычислять длину окружности, заданной своим радиусом R. Тогда вы можете оформить эти вычисления, определив встраиваемую функцию:

```
inline double Circ(double R){return 6.28318 * R; }
```

Обращение в любом месте программы вида **Circ(2)** приведет к встраиванию в соответствующем месте кода "6.28318 * 2" (если компилятор сочтет это целесообразным).

1.7.7 Перегрузка функций

C++ позволяет определить несколько функций с одним и тем же именем, если эти функции имеют разные наборы параметров (по меньшей мере разные типы параметров). Эта особенность называется *перегрузкой функции.* При вызове перегруженной функции компилятор C++ определяет соответствующую функцию путем анализа количества, типов и порядка следования аргументов в вызове. Перегрузка функции обычно используется для создания нескольких функций с одинаковым именем, предназначенных для выполнения сходных задач, но с разными типами данных. Применение при этом перегруженных функций делает программы более понятными и легко читаемыми.

Пусть, например, вы хотите определить функции, добавляющие в заданную строку типа **(char** *) символ пробела и значение целого числа, или значение числа с плавающей запятой, или значение булевой переменной. Причем хотите обращаться в любом случае к функции, которую называете, например, ToS, предоставив компилятору самому разбираться в типе параметра и в том, какую из функций надо вызывать в действительности. Для решения этой задачи вы можете описать следующие функции:

```
char * ToS(char *S,int X)
(return strcat(strcat(S," "), IntToStr(X).c_str()); }

char * ToS(char *S, double X)
```

```
{return strcat(strcat(S," "),FloatToStr(X).c_str());}

char * ToS(char *S, bool X)
(if (X) return strcat(S," true");
 else return strcat(S," false");}
```

Тогда в своей программе вы можете написать, например, вызовы:

```
char S[128] = "Значение =";
char S1 = ToS(S,5);
```

или

```
char S[128] = "Значение =";
char S2 = ToS(S,5.3);
```

или

```
char S[128] = "Значение =";
char S3 = ToS(S,true);
```

В первом случае будет вызвана функция с целым аргументом, во втором – с аргументом типа double, в третьем — с булевым аргументом. Вы видите, что перегрузив соответствующие функции вы существенно облегчили свою жизнь, избавившись от необходимости думать о типе параметра.

Приведем еще один пример, в котором перегруженные функции различаются количеством параметров. Ниже описана перегрузка функции, названной **Area** и вычисляющей площадь круга по его радиусу R, если задан один параметр, и площадь прямоугольника по его сторонам a и **b**, если задано два параметра:

```
double Area(double R) { return 6.28318 * R * R; }
double Area(double a, double b) ( return a * b; }
```

Тогда операторы вида

```
S1 = Area(1);
S2 = Area(1,2);
```

приведут в первом случае к вызову функции вычисления площади круга, а во втором — к вызову функции вычисления площади прямоугольника.

Перегруженные функции различаются компилятором с помощью их *сигнатуры* — комбинации имени функции и типов ее параметров. Компилятор кодирует идентификатор каждой функции по числу и типу ее параметров (иногда это называется декорированием имени), чтобы иметь возможность осуществлять надежное связывание типов. Надежное связывание типов гарантирует, что вызывается надлежащая функция и что аргументы согласуются с параметрами. Компилятор выявляет ошибки связывания и выдает сообщения о них.

Для различения функций с одинаковыми именами компилятор использует только списки параметров. Перегруженные функции не обязательно должны иметь одинаковое количество параметров.

Программисты должны быть осторожными, имея дело в перегруженных функциях с параметрами по умолчанию, поскольку это может стать причиной неопределенности. Функция с пропущенными аргументами по умолчанию может оказаться вызванной аналогично другой перегруженной функции; это синтаксическая ошибка.

Рассмотренный аппарат перегрузки функций — только один из возможных способов решения поставленной задачи, правда, универсальный, позволяющий работать и с разными типами параметров, и с разным числом параметров. В следующем разделе рассмотрен еще один механизм — шаблоны, позволяющий решать аналогичные задачи, правда, для более узких классов функций.

1.7.8 Шаблоны функций

Перегруженные функции обычно используются для выполнения сходных операций над различными типами данных. Если операции идентичны для каждого типа, это можно выполнить более компактно и удобно, используя *шаблоны функций.* Вам достаточно написать одно единственное определение шаблона функции. Основываясь на типах аргументов, указанных в вызовах этой функции, C++ автоматически генерирует разные функции для соответствующей обработки каждого типа. Таким образом, определение единственного шаблона определяет целое семейство решений.

Все определения шаблонов функций начинаются с ключевого слова **template,** за которым следует список формальных типов параметров функции, заключенный в угловые скобки "<" и ">". Каждый формальный тип параметра предваряется ключевым словом **class.** Формальные типы параметров — это встроенные типы или типы, определяемые пользователем. Они используются для задания типов аргументов функции, для задания типов возвращаемого значения функции и для объявления переменных внутри тела описания функции. После шаблона следует обычное описание функции.

Каждый формальный параметр в определении шаблона должен хотя бы однажды появиться в списке параметров функции. Каждое имя формального параметра в списке определения шаблона должно быть уникальным. Отсутствие ключевого слова **class** перед каждым формальным параметром шаблона функции является ошибкой.

Приведем пример шаблона функции, возвращающей минимальный из трех передаваемых в нее параметров любого (но одинакового) типа:

```
template <class T>
T min(T x1, T x2, T x3)
{
   T lmin = x1;
   if (x2 < lmin)
         lmin = x2;
   if (x3 < lmin)
         lmin = x3;
   return lmin;
}
```

В заголовке шаблона этой функции объявляется единственный формальный параметр T как тип данных, который должен проверяться функцией min. В следующем далее заголовке функции этот параметр T использован для задания типа возвращаемого значения (T **min)** и для задания типов всех трех параметров xl – x3. В теле функции этот же параметр T использован для указания типа локальной переменной **lmin.**

Объявленный таким образом шаблон можно использовать, например, следующим образом:

```
int i1 = 1, i2 = 3, i3 = 2;
double r1 = 2.5, r2 = 1.7, r3 = 3.4;
AnsiString s1 = "строка 1", s2 = "строка 2", s3 = "строка 3";
Label1->Caption = min(i1,i2,i3);
Label2->Caption = min(r1,r2,r3);
Label3->Caption = min(s3, s2, s1);
```

Когда компилятор обнаруживает вызов min в исходном коде программы, этот тип данных, переданных в min, подставляется всюду вместо T в определении шаблона и C++ создает законченную функцию для определения максимального из трех значений указанного типа данных. Затем эта созданная функция компилируется. Таким образом, шаблоны играют роль средств генерации кода.

Например, при вызове функции с тремя целыми параметрами компилятор сгенерирует функцию:

```
int min(int x1, int x2, int x3)
{
   int lmin =  x1;
   if (x2 < lmin)
         lmin = x2;
   if (x3 < lmin)
         lmin = x3;
   return lmin;
}
```

Приведенный шаблон будет работать для любых предопределенных или введенных пользователем типов, для которых определена операция отношения <.

При описании шаблонов функций можно широко использовать ключевое слово **typename.** Оно означает, что идентификатор, записанный после него, является именем типа. Пока не все компиляторы C++ поддерживают это ключевое слово. Но компилятор C++Builder 6 поддерживает.

Словом **typename,** во-первых, можно заменить ключевое слово **class** в первой строке объявления шаблона. Например:

```
template <typename T>]
```

Эта замена не дает ничего принципиально нового, но более отвечает контексту. Ведь объявляемый формальный тип параметра вовсе не обязательно должен быть классом, как можно было бы подумать при применении ключевого слова class.

Можно также использовать слово **typename** в описании вводимых в шаблоне типов данных.

1.8 Области видимости переменных и функций

1.8.1 Правила, определяющие область видимости

Область видимости или *область действия* переменной или функции — это часть программы, в которой на нее можно ссылаться. При решении вопросов видимости важнейшее значение имеет понятие блока. Блок — это фрагмент кода, ограниченный фигурными скобками "{ }".

Существуют четыре области действия идентификатора переменной или функции — *область действия функция, область действия файл, область действия блок* и *область действия прототип функции.*

Идентификатор, объявленный вне любой функции (на внешнем уровне), имеет *область действия файл.* Такой идентификатор «известен» всем функциям от точки его объявления до конца файла. Переменные, объявленные таким способом, называются *глобальными.* Глобальные переменные, описания функций и прототипы функций, находящиеся вне функции — все они имеют областью действия файл.

Метки (идентификаторы с последующим двоеточием, например, **start:**) — единственные идентификаторы, имеющие *областью действия функцию.* Метки можно использовать всюду в функции, в которой они появились, но на них нельзя ссылаться вне тела функции. Метки используются в структурах **switch** (как метки case) и в операторах **goto** (см. разд. 1.10.1.2 и 1.10.1.3). Метки относятся к тем деталям реализации, которые функции «прячут» друг от друга. Это скрытие — один из наиболее фундаментальных принципов разработки хорошего программного обеспечения.

Идентификаторы, объявленные внутри блока (на внутреннем уровне), имеют *областью действия блок.* Область действия блок начинается объявлением иденти-

фикатора и заканчивается конечной правой фигурной скобкой блока. Если имеются вложенные блоки, то переменная внешнего блока видна и во вложенных блоках.

Локальные переменные, объявленные в начале функции, имеют областью действия блок так же, как и параметры функции, являющиеся локальными переменными.

Любой блок может содержать объявления переменных. Если блоки вложены и идентификатор во внешнем блоке или идентификатор глобальной переменной идентичен идентификатору во внутреннем блоке, одноименный идентификатор внешнего блока или глобальный «невидим» (скрыт) до момента завершения работы внутреннего блока. Это означает, что пока выполняется внутренний блок, он видит значение своих собственных локальных идентификаторов, а не значения идентификаторов с идентичными именами в охватывающем блоке.

Например:

```
int i = 1, k = 4    // объявление глобальных переменных
{
 int i = 5, j = 2; // объявление переменных внешнего блока

 ...                // видны переменные j, k
                    // и переменная i этого блока
 {
  int i = 7;        // объявление переменной i внутреннего блока

  ...               // видны переменные j, k
                    // и переменная i внутренняя
 }
...                 // видны переменные j, k
                    // и переменная i внешнего блока
}
```

Локальная переменная не только видима в пределах блока, в котором она объявлена. Ее время жизни тоже определяется временем выполнения блока. Переменная создается в момент входа в блок и разрушается в тот момент, когда управление выходит за пределы блока. Таким образом, подобная переменная не может сохранять какие-то значения в промежутках между выполнением операторов блока. В приведенном выше примере переменная i во внутреннем блоке будет создаваться каждый раз, когда управление передается в этот блок, ей каждый раз будет присваиваться значение 7, и она каждый раз будет разрушаться при выходе из блока.

Сказанное выше о времени жизни относится к так называемым автоматическим **(auto)** переменным и не относится к статическим переменным, объявленным как **static.** Например:

```
static int i = 7;
```

Такие статические переменные существуют все время работы программы и инициализируются только один раз. Таким образом, в этих переменных можно накапливать какую-то информацию. Например, они могут служить счетчиками числа обращений к блоку (см. соответствующий пример в разд. 1.6.2). Но областью действия таких переменных является только блок, в котором они объявлены.

Из внутреннего блока можно получить доступ к одноименной глобальной переменной с помощью унарной операции разрешения области действия "::". Например, выражение ::I означает глобальную переменную I, даже если в данном блоке объявлена локальная переменная I. В приведенном ранее примере с вложенными блоками можно, например, записать во внутреннем блоке оператор

```
i = ::i + 1;
```

Этот оператор присвоит внутренней переменной i значение на единицу большее значения глобальной переменной i.

Подчеркнем, что таким образом можно получить доступ только к одноименной глобальной переменной, а не к локальной переменной, описанной во внешнем блоке.

Единственными идентификаторами с *областью действия прототип функции* являются те, которые используются в списке параметров прототипа функции (см. разд. 1.7.1). Прототипы функций не требуют имен в списке параметров — требуются только типы. Если в списке параметров прототипа функции используется имя, компилятор это имя игнорирует. Идентификаторы, используемые в прототипе функции, можно повторно использовать где угодно в программе, не опасаясь двусмысленности.

Теперь остановимся на проблемах видимости переменных в приложениях, имеющих несколько модулей. Пусть вы имеете два модуля — **Unit1** и **Unit2** и хотите в модуле **Unit2** видеть и использовать переменные и функции, объявленные в модуле **Unitl.** Вы можете в модуле **Unit2** видеть те переменные, которые являются глобальными в модуле **Unitl,** т.е. объявлены вне каких-нибудь функций в заголовочном файле модуля или в его файле реализации. Но для того, чтобы это было возможно, вы должны повторно объявить их (без инициализации) в модуле **Unit2** со спецификацией **extern.** Например, если в модуле **Unitl** имеется объявление глобальной переменной

```
int al = 10;
```

то в модуле Unit2 вы можете использовать эту переменную, если запишете объявление

```
extern int a1;
```

Причем, это не зависит от того, включили ли вы директивой **#include** заголовочный файл **Unit1.h** в модуль Unit2, или нет.

Отметим еще одну особенность использования переменных, описанных в другом модуле. Если в заголовочном модуле **Unitl** объявлена описанная выше переменная al, а в модуле **Unit2** вы включили директивой **#include** заголовочный файл **Unitl.h,** но не записали объявление этой переменной со спецификацией **extern** (вообще не дали объявление al), то в модуле **Unit2** будет создана копия переменной al, инициализированная согласно объявлению в **Unitl.** Но это будет копия, совершенно изолированная от переменной al в модуле **Unitl.** В модулях **Unitl и Unit2** будут существовать две различные переменные с одним именем **a1.** И изменение одной из них никак не скажется на значении другой.

Все сказанное относится только к глобальным переменным. Локальные переменные, объявляемые внутри функций, невозможно видеть в другом модуле.

Теперь рассмотрим видимость функций в приложениях, имеющих несколько модулей. Если в модуле **Unitl** в его заголовочном файле вне описания класса вы объявили некоторую функцию F, то в другом модуле **Unit2** вы можете использовать ее при выполнении одного из двух условий:

- вы включаете директивой **#include** в модуль **Unit2** заголовочный файл **Unitl.h**
- вы повторяете в модуле **Unit2** (в заголовочном файле или файле реализации) объявление функции F

В обоих случаях вы сможете вызвать функцию F из любого места модуля **Unit2.**

Если же функция F объявлена в модуле **Unitl** не заголовочном файле, а в файле реализации, то единственный способ использовать ее в модуле **Unit2** — повторить в нем объявление функции.

Если вы хотите предотвратить возможность обращения к функции из другого модуля, ее надо объявить со спецификацией **static.** Например:

```
static void F(void);
```

Подведем некоторые итоги проведенного рассмотрения проблем видимости переменных и функций.

- Переменные, объявленные в заголовочном файле модуля или в файле его реализации вне описания класса и функций, являются глобальными. Они доступны везде внутри данного модуля. Для доступа к ним из внешних модулей в этих модулях должно быть повторено их объявление (без инициализации) с добавлением спецификации **extern.**
- Функции, объявленные в заголовочном файле модуля вне описания класса, являются глобальными. Они доступны везде внутри данного модуля. Для доступа к ним из внешних модулей в этих модулях или надо повторить их объявление, или включить директивой **#include** заголовочный файл того модуля, в котором функции описаны.
- Функции, объявленные в файле реализации модуля, являются глобальными. Они доступны везде внутри данного модуля. Для доступа к ним из внешних модулей в этих модулях надо повторить их объявление.
- Элементы (переменные и функции), объявленные в классе в разделе **private,** видимы и доступны только внутри данного модуля. При этом из функций, объявленных внутри класса, к ним можно обращаться непосредственно по имени, а из других функций — только со ссылкой на объект данного класса. Если в модуле описано несколько классов, то объекты этих классов взаимно видят элементы, описанные в их разделах **private.**
- Элементы, объявленные в классе в разделе **public,** видимы и доступны для объектов любых классов и для других модулей, в которых директивой **#include** включен заголовочный файл данного модуля. При этом из объектов того же класса, к ним можно обращаться непосредственно по имени, а из других объектов и процедур — только со ссылкой на объект данного класса.
- В классах, помимо обсуждавшихся ранее, могут быть еще разделы **protected** — защищенные. Элементы, объявленные в классе в разделе **protected,** видимы и доступны для любых объектов внутри данного модуля, а также для объектов классов — наследников данного класса в других модулях. Объекты из других модулей, классы которых не являются наследниками данного класса, защищенных элементов не видят.
- Элементы, объявленные внутри функции или блока, являются локальными, т.е. они видимы и доступны только внутри данной функции или данного блока. При этом время жизни переменных, объявленных внутри функции или блока, определяется временем активности данного блока. Сделать локальную переменную существующей постоянно можно с помощью спецификации static.
- Переменные и функции, объявленные в головном файле проекта, являются глобальными для этого файла. Если требуется доступ к ним из других модулей, то для функций в них должны быть повторены их объявления, а для переменных — повторено объявление (без инициализации) со спецификацией **extern.**
- Если во внутреннем блоке объявлена переменная с тем же именем, что во внешнем блоке, или с тем же именем, что и глобальная переменная, то соответствующая внешняя или глобальная переменная в блоке не видна. В этом случае подучить доступ к одноименной глобальной переменной можно только с помощью унарной операции разрешения области действия "::".

1.8.2 Явное определение доступа с помощью объявлений namespace и using

Изложенные в предыдущем разделе правила определяют автоматически устанавливаемые области видимости. Однако такого неявного задания областей видимости иногда может быть недостаточно. Если речь идет о большом проекте, который создается несколькими разработчиками, всегда возможно перекрытие идентификаторов, определенных в разных местах программы. Поэтому желателен инструмент, позволяющий явным образом указывать области видимости идентификаторов.

Таким инструментом является объявление области видимости имен ключевым словом **namespace** и последующее объявление использования функций и переменных из той или иной области ключевым словом **using.**

Синтаксис объявления области видимости:

```
namespace имя_области
{
объявления типов, переменных и функций
}
```

Например:

```
namespace A{
 int i = 1;
 void F1(int i)
 {
  Form1->Label1->Caption = "Область A: i = " + IntToStr(i);
 }
}
namespace B{
 int i = 2;
 void F1(int i)
 {
  Form1->Label1->Caption = "Область B: i = " + IntToStr(i);
 }
}
```

Приведенные операторы объявляют две области видимости с именами A и B. В обеих областях объявлены переменные i и функции **F1.**

Объявление области с тем же именем может повториться в программе и содержать объявления каких-то новых переменных и функций. Соответствующие идентификаторы добавятся в указанную область.

Доступ к объявленным переменным и функциям из любой точки файла может осуществляться несколькими способами. Самый простой — с помощью операции разрешения области действия "::". Например, оператор

```
B::F1(A::i);
```

вызовет функцию **F1** из области B и передаст в нее значение переменной i из области A.

Подобный доступ гибкий, но он требует каждый раз указывать область видимости. Если явное указание областей видимости сделано для того, чтобы устранить появившиеся в программе случайные наложения идентификаторов, то явное указание при каждом применении идентификатора соответствующей области действия потребует исправлений во многих местах программы и может привести к появлению ошибок. Более простой способ указания области действия — применение ключевого слова **using.** Одна из возможных форм применения **using:**

```
using namespace имя_области;
```

Например, если поместить в тексте оператор

```
using namespace A;
```

то все последующие операторы будут брать идентификаторы из области A. Тогда, например, размещенный где-то в тексте после **using** оператор

```
F1(i);
```

вызовет функцию F1 из области A и передаст в нее значение переменной i из области A.

Операторы **using** могут иметь и другую форму, определяющую область для конкретного идентификатора:

```
using имя_области :: идентификатор;
```

Например, после операторов

```
using A::F1;
using B::i;
```

оператор

```
F1(i);
```

вызовет функцию **F1** из области A и передаст в нее значение переменной i из области B.

При объявлении области видимости с помощью namespace в теле объявления могут присутствовать не только объявления переменных и функций, но и операторы **namespace,** определяющие некоторые внутренние области видимости, и операторы **using namespace,** ссылающиеся на ранее определенные области. Таким образом, области видимости могут быть вложенные. Например, объявления могут иметь вид:

```
namespace A {
 ...
}
namespace B {
 using namespace A;
 ...
 namespace C {
  ...
 }
}
```

Здесь область B использует ранее объявленную область A и содержит внутри себя вложенную область C. Доступ к вложенным областям осуществляется последовательным применением операции разрешения области действия. Например:

```
using namespace B :: C;
```

1.9 Операции

1.9.1 Общее описание

Операции подобны встроенным функциям языка. Они применяются к выражениям — *операндам.* Большинство операций имеют два операнда, один из которых помещается перед знаком операции, а другой — после. Например, операция сложения "+" имеет два операнда: X + Y и складывает их. Такие операции называются бинарными. Существуют и унарные операции, имеющие только один операнд, помещаемый после знака операции. Например, запись "-X" означает применение к операнду X операции унарного минуса "-".

В сложных выражениях последовательность выполнения операций определяется скобками, старшинством операций, а при одинаковом старшинстве — ассоциативностью операций. Эти вопросы будут обсуждены в разд. 1.9.16.

1.9.2 Арифметические операции

Арифметические операции применяются к действительным числам, целым числам и указателям. Определены следующие бинарные арифметические операции:

Обозначение	Операция	Типы операндов и результата	Пример
+	сложение	арифметический, указатель	X + Y
–	вычитание	арифметический, указатель	X - Y
*	умножение	арифметический	X* Y
/	деление	арифметический	X / Y
%	остаток целочисленного деления	целый	I % 6

Определены следующие унарные арифметические операции:

Обозначение	Операция	Типы операндов и результата	Примеры
+	Унарный плюс (подтверждение знака)	арифметический	+7
–	Унарный минус (изменение знака)	арифметический	-X
++	инкремент	арифметический, указатель	i++; ++i
—	декремент	арифметический, указатель	i--;--i

Для арифметических операций действуют следующие правила.

Бинарные операции сложения "+" и вычитания " " применимы к целым и действительным числам, а также к указателям.

В операции сложения указателем может быть только один из двух операндов. В этом случае второй операнд должен быть целым числом. Указатель, участвующий в операции сложения, должен быть указателем на элемент массива. В этом случае добавление к указателю целого числа эквивалентно сдвигу указателя на заданное число элементов массива.

В операции вычитания указатель на элемент массива может быть первым операндом (тогда второй операнд — целое число) или оба операнда могут быть указателями на элементы одного массива. Вычитание из указателя целого числа эквивалентно сдвигу указателя на заданное число элементов массива. Вычитание двух указателей возвращает число элементов массива, расположенных между теми элементами, на которые указывают указатели. Подробнее об арифметике указателей см. в гл. 2 в разд. 2.8.

В операциях умножения "*" и деления "/" операнды могут быть любых арифметических типов. При разных типах операндов применяются стандартные правила автоматического приведения типов (см. разд. 2.2). В операции вычисления остатка от деления "%" оба операнда должны быть целыми числами.

В операциях деления и вычисления остатка второй операнд не может быть равен нулю. Если оба операнда в этих операциях целые, а результат деления является не целым числом, то знак результата вычисления остатка совпадает со знаком первого операнда, а для операции деления используются следующие правила:

1. Если первый и второй операнд имеют одинаковые знаки, то результат операции деления — наибольшее целое, меньшее истинного результата деления.

2. Если первый и второй операнд имеют разные знаки, то результат операции деления — наименьшее целое, большее истинного результата деления.

Округление всегда осуществляется по направлению к нулю.

Унарные операции инкремента "++" и декремента "--" сводятся к увеличению "++" или уменьшению "--" операнда на единицу. Операции применимы к операндам, представляющим собой выражения любых арифметических типов или типа указателя. Причем выражение должно быть модифицируемым L-значением, т.е. должно допускать изменение. Например, ошибочным является выражение ++(a + b), поскольку (a + b) не является переменной, которую можно модифицировать.

Операции инкремента и декремента выполняются быстрее, чем обычное сложение и вычитание. Поэтому, если переменная a должна быть увеличена на 1, лучше применить операцию "++", чем выражения a = a + 1 или оператор a += 1, использующий описанную в разд. 1.9.4 операцию "+=".

Если операция инкремента или декремента помещена перед переменной, говорят о *префиксной форме записи* инкремента или декремента. Если операция инкремента или декремента записана после переменной, то говорят о *постфиксной форме записи*. При префиксной форме переменная сначала увеличивается или уменьшается на единицу, а затем это ее новое значение используется в том выражении, в котором она встретилась. При постфиксной форме в выражении используется текущее значение переменной, и только после этого ее значение увеличивается или уменьшается на единицу.

Например, в результате выполнения операторов

```
int i = 1, j;
j = i++ * i++;
```

значение переменной i будет равно 3, а переменной j — 1. Оператор, присваивающий значение переменной j, будет работать следующим образом: сначала значение i, равное 1, умножится само на себя, т.е. вычислится значение выражения в правой части оператора; затем это значение присвоится переменной j, а значение i увеличится на 1 в результате первой операции инкремента и еще раз увеличится на 1 в результате второй операции инкремента.

Если изменить эти операторы следующим образом:

```
int i = 1, j;
j = ++i * ++i;
```

то результат будет другим: значение i будет равно 3, а значение j — 9. В этом случае оператор, присваивающий значение переменной j будет работать следующим образом: сначала выполнится первая операция инкремента и значение i станет равно 2; затем выполнится вторая операция инкремента и значение i станет равно 3; а затем это значение i умножится само на себя, т.е. вычислится значение выражения в правой части оператора и это значение присвоится переменной j.

1.9.3 Особенности выполнения арифметических операций с целыми и действительными числами

В этом разделе мы рассмотрим ошибки, которые могут возникать при выполнении арифметических операций. Начнем с операций с целыми числами. При целочисленном делении на нуль генерируется исключение **EDivByZero** и его можно обрабатывать методами, рассмотренными в разд. 1.12. Но результат сложения, вычитания, умножения целых чисел обычно не проверяется на переполнение. В редких случаях при переполнении генерируется исключение **EIntOverflow**. А чаще, если результат превышает максимальное значение для данного типа, полученное значение будет неправильным. Например, следующие операторы:

```
int i = 2147483646;
int i1 = 1;
```

```
int i2 = 2;
int i3 = i + i1;
int i4 = i + i2;
int i5 = i3 * i3;
```

дадут значения i3 = 2147483647, i4 = -2147483648 и i5 = 1. Первое из них правильное, второе — совершенно неверное отрицательное число (минимально возможное значение целого int), а третье — также абсурдный результат. Неверные результаты элементарных арифметических операций объясняются тем, что максимально допустимое значение int — 2147483647. При вычислении i4 это значение превышено на 1, что привело к появлению отрицательного знака, так как лишний старший бит, установленный в 1, воспринимается как признак отрицательного числа. С превышением допустимого значения связан и результат вычисления i5. Подобное поведение целых чисел надо учитывать и отслеживать программно. Иначе могут возникнуть трудно отлавливаемые ошибки выполнения.

Способ обработки ошибок выполнения операций с действительными числами в C++Builder зависит от маски FPU (floating-point unit) — слова, управляющего исключениями при операциях с плавающей запятой. В число ошибок входят ошибочная операция, ненормализованная операция, деление на нуль, переполнение, потеря порядка, потеря точности. Приведем некоторые примеры, связанные с такими ошибками. Операторы

```
double x = 1;
double y = 0;
double z = x / y;
```

вызывают ошибку деления на нуль. Операторы

```
float x;
double y = 1.5e-100;
x = y;
```

вызывают ошибки потери порядка и потери точности, поскольку переменная типа float не может хранить столь малого числа. Операторы

```
float x = 1e20;
float y = x * x;
```

вызывают ошибку переполнения. Ошибки, связанные с ошибочными или ненормализованными операциями, вообще говоря, в нормально скомпилированных программах обычно не возникают.

При возникновении одной из перечисленных выше ошибок она отражается в *слове состояния*. Это слово можно видеть в процессе отладки в окне FPU. Первые биты этого слова, относящиеся к рассматриваемым ошибкам, следующие:

Флаг	Описание	Бит
IE	ошибочная операция	0
DE	ненормализованная операция	1
ZE	деление на нуль	2
OE	переполнение	3
UE	потеря порядка	4
PE	потеря точности	5

При возникновении той или иной ошибки соответствующий бит устанавливается в 1.

Последующее зависит от установки масок в *управляющем слове*. Основные биты этого слова определяют следующее:

Флаг	Описание	Биты
IM	Маска генерации исключений при ошибочных операциях	0
DM	Маска исключений ненормализованных операций	1
ZM	Маска исключений деления на нуль	2
OM	Маска исключений переполнения	3
UM	Маска исключений потери порядка	4
PM	Маска исключений точности	5
PC	Управление точностью	8, 9
RC	Управление округлением	10, 11

Первые шесть битов управляющего слова определяют маски генерации исключений. Если в соответствующем бите записан 0, то при возникновении ошибки, связанной с этим битом, генерируется исключение. Тогда его надо перехватывать обычными способами (см. разд. 1.12). Если же в бите записана 1, генерация исключения маскируется. В этом случае ошибка выполнения операции приведет к тому, что в качестве результата будет выдано одно из следующих значений: "INF" (положительная бесконечность), "-INF" (отрицательная бесконечность), "NAN" (нецифровое значение). Так будут выглядеть результаты операции, если отобразить их в виде строк. Последующее использование полученных значений в арифметических операциях приведет к выдаче в качестве результата аналогичных значений.

Задавать маски исключений можно функциями **SetExceptionMask, _control87, Set8087CW,** описанными в гл. 4. Остановимся на одной из них — функции **_control87**, а остальные, в некоторых отношениях более удобные, вы можете посмотреть в гл. 4. Функция **_control87** объявлена следующим образом:

```
unsigned int _control87(unsigned int newcw, unsigned int mask);
```

Параметр **newcw** содержит устанавливаемое значение управляющего слова FPU. А параметр **mask** содержит маску установки. В управляющем слове заменяются только те биты, для которых в **mask** заданы 1. Функция возвращает новое значение управляющего слова.

Например, оператор

```
_control87(0x3F, 0x3F);
```

маскирует генерацию всех исключений при выполнении арифметических операций.

Оператор

```
_control87(0, 0x3F);
```

стирает маски всех исключений.

Оператор

```
unsigned int m = _control87(0, 0);
```

заносит в переменную m текущее значение управляющего слова, не изменяя его.

Если генерация исключений замаскирована, можно определить, произошла ли ошибка при выполнении операций с плавающей запятой по описанному выше слову состояния. Это слово возвращается функцией **_status87.** Отдельные биты слова можно проверять операцией И. Например, оператор

```
if ( _status87() & 0x4)
  ShowMessage("Деление на нуль");
```

прореагирует на произошедшее в предшествующих операциях деление на нуль. А оператор

```
if ( _status87() & 0x3D)
  ShowMessage("Ошибка операции с плавающей запятой");
```

прореагирует на любую ошибку операций с плавающей запятой.

Конечно, в реальной программе вместо указанных в приведенных операторах абстрактных сообщений надо дать пользователю какие-то более осмысленные пояснения и советы, что надо делать для продолжения работы. А можно принять какие-то меры для исправления ошибки.

Маскирование исключений облегчает решение некоторых задач, но при отсутствии в программе соответствующих проверок оно может привести к маскированию ошибок. Так что пользоваться им надо осторожно.

Биты 8 и 9 управляющего слова содержат флаг PC, управляющий точностью вычислений с плавающей запятой. Этот флаг может принимать следующие значения:

0	точность, соответствующая типу float (4 байта)
1	не используется
2	точность, соответствующая типу double (8 байт)
3	точность, соответствующая типу long double (10 байт)

К битам управления точностью, как и ко всем остальным битам управляющего слова FPU, можно получить доступ с помощью описанной выше функции **_control87**. Остановимся на ней. А описание более специализированной функции управления точностью **SetPrecisionMode** вы можете посмотреть в гл. 4. Пусть, например, в вашем приложении есть операторы

```
double x = 3;
long double z = 1 / x;
```

Тогда, если перед вычислением значения z вы установите функцией _control87 соответствующее значение флага точности, получите следующий результат:

оператор задания флага PC	z
_control87(0xF000, 0x300)	**0.333333343267440796**
_control87(0xF200, 0x300)	**0.333333333333333315**
_control87(0xF300, 0x300)	**0.333333333333333333**

Биты 10 и 11 управляющего слова содержат флаг RC, управляющий округлением при вычислениях с плавающей запятой. Этот флаг может принимать следующие значения:

0	округление к ближайшему значению
1	округление к меньшему значению, т.е. округление в сторону отрицательной бесконечности
2	округление к большему значению, т.е. округление в сторону положительной бесконечности
3	усечение младших разрядов, т.е. округление в сторону нуля

К битам управления округлением можно получить доступ с помощью все той же функции **_control87** (посмотрите также в гл. 4 описание более специализированной функции **SetRoundMode).** Пусть, например, в вашем приложении есть операторы

```
float x = 3;
float y, z;
y = 1 / x;
2 = -1 / x;
```

Если перед вычислением значений y и **z** вы установите функцией _control87 соответствующие значения флага округления, то получите результат, представленный в таблице:

оператор задания флага RC	**y**	**z**
_control87(0xF000,OxCOO)	**0,33333334327**	**-0,33333334327**
_control87(0xF400, OxCOO)	**0,33333331347**	**-0,33333334327**
_control87(0xF800, OxCOO)	**0,33333334327**	**-0,33333331347**
_control87(0xFC00, OxCOO)	**0,33333331347**	**-0,33333331347**

1.9.4 Операции присваивания, отличие присваивания от метода Assign

В C++ определен ряд операций присваивания.

Обозначение	Операция	Типы операндов и результата	Пример
=	присваивание	любые	X = Y
+=	присваивание со сложением	арифметические, указатели, структуры, объединения	X += Y
−=	присваивание с вычитанием	арифметические, указатели, структуры, объединения	X −= Y
*=	присваивание с умножением	арифметические	X *= Y
/=	присваивание с делением	арифметические	X /= Y
%=	присваивание остатка целочисленного деления	целые	X %= Y
<<=	присваивание со сдвигом влево	целые	X <<= Y
>>=	присваивание со сдвигом вправо	целые	X >>= Y
&=	присваивание с поразрядной операцией И	целые	X &= Y
^=	присваивание с поразрядной операцией исключающее ИЛИ	целые	X ^= Y
|=	присваивание с поразрядной операцией ИЛИ	целые	X |= Y

Помимо простой операции присваивания "=" все прочие являются составными операциями. Они присваивают первому операнду результат применения соответствующей простой операции, указанной перед символом "=", к первому и второму операндам.

Например, выражение X += Y эквивалентно выражению X = X + Y, но записывается компактнее и может выполняться быстрее. Аналогично определяются и другие операции присваивания: X %= Y эквивалентно X = X % Y и т.д. (см. соответствующие простые операции в разд. 1.9.2 и 1.9.6).

При записи составных операций присваивания между символом операции и знаком равенства пробел не допускается.

В операциях присваивания первый операнд не может быть нулевым указателям.

Операции присваивания возвращают как результат присвоенное значение. Благодаря этому они допускают сцепление. Например, вы можете написать:

```
A = (B = C = 1) +1;
```

Выполняются операции присваивания справа налево. Поэтому приведенное выражение задаст переменным B и C значения 1, а переменной A — 2. Вычисляться это будет следующим образом. Сначала выполняются операции, заключенные в скобки, а из них первой — самая правая (т.е. C = 1). Эта операция вернет 1, так что далее будет выполнена операция B = 1. Она вернет значение 1, после чего выполнится операция сложения 1 + 1. Полученное в результате значение 2 присвоится переменной A.

Применительно к указателям на объекты надо четко представлять различие между оператором присваивания и методом копирования Assign, свойственным многим классам объектов. Метод Assign используется следующим образом:

```
объект_приемник->Assign(объект_источник);
```

Например:

```
A->Assign(B);
```

Этот оператор копирует содержание объекта B (все его свойства) в объект A.

Для тех же самых объектов A и B можно записать оператор присваивания:

```
A = B;
```

Различие между двумя приведенными операторами следующее. Метод Assign копирует содержимое одного объекта в другой. Таким образом, в памяти будет иметься два объекта A и B одинакового содержания. А оператор присваивания, примененный к указателям (имя объекта — это указатель на объект), присваивает указателю A значение указателя B. Таким образом, и A, и B будут указывать на один и тот же объект в памяти. А тот объект, на который до выполнения этого оператора указывал A, может быть вообще потерян, если в программе где-то не хранится другой указатель на него.

1.9.5 Операции отношения и эквивалентности

Операции отношения и эквивалентности используются при сравнении двух операндов. Они возвращают true — истина, если указанное соотношение операндов выполняется, и false (0) — ложь, если соотношение не выполняется. Определены следующие операции отношения:

Обозначение	Операция	Типы операндов	Пример
==	Равно	арифметический, указатели	I == Max
N	Не равно	арифметический, указатели	X != Y
<	Меньше чем	арифметический, указатели	X< Y
>	Больше чем	арифметический, указатели	Len > 0
<–	Меньше или равно	арифметический, указатели	Cnt <= I
>=	Больше или равно	арифметический, указатели	I >= 1

Операнды должны иметь совместимые типы, за исключением целых и действительных типов, которые могут сравниваться друг с другом.

Применять операции "<", "<=", ">", ">=" к указателям имеет смысл, только если оба операнда указывают на элементы одного массива.

Операции "==" и "!=" могут применяться к указателям на любые объекты. В этом случае они вернут соответственно **true и false,** только если указатели указывают на один и тот же объект.

Следует предостеречь от довольно распространенной ошибки: случайного применения вместо операции эквивалентности "==" операции присваивания "=". Например, если вы по ошибке вместо оператора

```
if(A == 2) ...;
```

написали оператор

```
if (A = 2) ...;
```

то это не будет расценено как синтаксическая ошибка. Дело в том, что в C++ любое выражение, имеющее некоторое значение, может использоваться в условных операторах, в частности, в if. Если значение выражения 0, то оно трактуется как **false.** Любое другое значение трактуется как **true.** Поэтому результат операции A = 2 будет трактоваться как **true** и независимо того, чему было равно значения A до выполнения этого ошибочного оператора, условие в операторе **if** всегда будет считаться выполненным. К тому же эта ошибка приведет к несанкционированному изменению значения A.

К счастью, компилятор C++Builder замечает подобные недоразумения и при записи в операторе if операции присваивания на всякий случай делает замечание: "Possibly incorrect assignment" (Возможно некорректное присваивание). Это не ошибка, а только замечание. Так что если вы не обратите внимание на него, то потратите потом много времени на поиск ошибки в программе.

1.9.6 Логические операции

Логические операции принимают в качестве операндов выражения скалярных типов и возвращают результат булева типа: **true** или **false (0).**

Обозначение	Операция	Пример
!	Отрицание	!A
&&	Логическое И	A && B
||	Логическое ИЛИ	A|| B

Унарная операция логического отрицания "!" возвращает **true,** если операнд возвращает ненулевое значение. Таким образом, выражение !A эквивалентно выражению A == 0.

Операция логического И "&&" возвращает **true,** если оба ее операнда возвращают ненулевые значения. Если хотя бы один операнд возвращает **0 (false),** то операция И также возвращает **false.** Поэтому для сокращения времени расчета, если первый операнд возвращает нуль, то второй операнд даже не вычисляется.

Операция логического ИЛИ "||" возвращает **true,** если хотя бы один ее операнд возвращает ненулевое значение. Если оба операнда возвращают 0 **(false),** то операция ИЛИ также возвращает **false.** Для сокращения времени расчета, если первый операнд возвращает ненулевое значение, то второй операнд даже не вычисляется.

1.9.7 Поразрядные логические операции

Поразрядные логические операции работают с целыми числами и оперируют с их двоичными представлениями, т.е. работают с двоичными разрядами операндов.

Обозначение	Операция	Пример
~	поразрядное отрицание	~ X
&	поразрядное И	X & Y
|	поразрядное ИЛИ	X| Y
*	поразрядное исключающее ИЛИ	X ^ Y
<<	поразрядный сдвиг влево	X << 2
>>	поразрядный сдвиг вправо	Y>>I

Операция поразрядного отрицания "~" инвертирует каждый бит операнда.

Поразрядные операции "&", "|" и " " работают в соответствии со следующей таблицей, где E1 и E2 — сравниваемые биты операндов:

E1	E2	E1 & E2	E1 * E2	E1 | E2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Операция поразрядного сдвига вправо ">>" сдвигает биты левого операнда на число разрядов, указанное правым операндом. При этом правые биты теряются. Если левый операнд представляет собой целое без знака, то левые освободившиеся биты заполняются нулями. В противном случае они заполняются символом знака. Сдвиг целого числа на n разрядов вправо эквивалентен целочисленному делению его на 2^n.

Операция поразрядного сдвига влево "<<" сдвигает биты левого операнда на число разрядов, указанное правым операндом. При этом левые биты теряются, а правые заполняются нулями. Сдвиг целого числа на п разрядов влево эквивалентен умножению его на 2^n.

1.9.8 Операция запятая (последование)

Операция запятая ",", называемая операцией последования, соединяет два произвольных выражения, которые вычисляются слева направо. Сначала вычисляется выражение левого операнда. Тип его результата считается **void.** Затем вычисляется выражение правого операнда. Значение и тип результата операции последования считается равным значению и типу правого операнда.

Например, фрагмент текста

```
a = 4;
b = a + 5;
```

можно записать как

```
a = 4, b = a + 5;
```

Можно рекурсивно соединить операциями запятая последовательность выражений:

```
выражение_1, выражение_2, ..., выражение_n
```

Выражения будут вычисляться слева направо, а самое правое выражение определит значение и тип всей этой последовательности.

Соединяться запятыми могут не только выражения присваивания, но и другие. Например, вызов функции с тремя параметрами может иметь вид

```
func(i, (j = 1, j + 4), k);
```

Здесь в качестве второго параметра передается значение операции последования, заключенной в скобки. В результате вызов производится со следующими аргументами: (i, 5, k).

Операция последования используется в основном в операторах цикла **for** (см. разд. 1.10.2.1) для задания в заголовке некоторой совокупности действий. Например, цикл подсчета суммы элементов некоторого массива можно осуществить циклом **for** без использования операции последования:

```
int A[10], sum, i;
...
sum = A[0];
for (i = 1; i < 10; i++)
    sum += A[i];
```

То же самое можно реализовать более компактно с помощью операции последования:

```
int A[10], sum, i;
...
for (i = 1, sum = A[0]; i < 10; sum += A[i], i++);
```

Здесь операция последования использована дважды: при задании действий, выполняемых перед началом цикла (задание начальных значений i и sum), и при описании действий, выполняемых в теле цикла (суммирование значений элементов в sum и инкремент счетчика i).

Можно посоветовать не использовать без нужды операцию последования ",". Применение ее оправдано только при объединении одинаковых по смыслу выражений в основном в операторах циклов. Более широкое применение операции последования ухудшает читаемость кода, маскирует ошибки, усложняет сопровождение программы.

1.9.9 Условная операция (?:)

Условная операция "?:" - единственная трехчленная (тернарная) операция в C++, имеющая три операнда. Ее синтаксис:

```
условие ? выражение_1 : выражение_2
```

Первый операнд является условием, второй операнд содержит значение условного выражения в случае, если условие истинно (возвращает ненулевое значение), а третий операнд равен значению условного выражения, если условие ложно (возвращает нуль). Например, оператор

```
Label1->Caption =
        grade > 3 ? "Вы хорошо знаете материал" : "Плохо";
```

в зависимости от значения переменной **grade** выдаст текст "Вы хорошо знаете материал" при значении **grade,** превышающем 3, и текст "Плохо" при меньшем значении **grade.**

Оператор с условной операцией выполняет фактически те же функции, что и оператор **if...else** (см. разд. 1.10.1.1). Но в ряде случаев применение условной операции компактнее и нагляднее оператора **if...else.** К тому же иногда условная операция может использоваться в таких ситуациях, когда применение оператора **if...else** синтаксически невозможно.

В условной операции условие может быть любым скалярным выражением. Условные выражения могут быть практически любого типа (арифметические, указатели, структуры, объединения), но типы двух выражений в операции должны быть согласованными. В качестве условных выражений могут также фигурировать какие-то исполняемые действия.

1.9.10 Операция sizeof

Операция **sizeof** определяет размер в байтах своего операнда — переменной, объекта, типа. Возвращаемый результат имеет тип **size_t (unsigned).**

Операция имеет две формы:

```
sizeof выражение
sizeof (имя_типа)
```

Например:

```
sizeof *Label1;
sizeof(TLabel);
sizeof a;
sizeof  (int);
```

Во всех случаях операция возвращает целое, равное числу байтов в объекте **(*Labell),** типе **(TLabel, int),** переменной (а).

Надо учесть, что размер переменной, объекта, типа может изменяться в зависимости от машины и от используемой версии программного обеспечения. Поэтому во всех случаях, когда вам требуется знать размер объекта или типа, нельзя полагаться на документацию, а надо использовать операцию **sizeof.**

Если операндом является выражение, то sizeof возвращает суммарный объем памяти, занимаемый всеми переменными и константами, входящими в него. Если операндом является массив, то возвращается объем памяти, занимаемый всеми элементами массива (т.е. имя массива не воспринимается в данном случае как указатель). Число элементов в массиве можно определить выражением **sizeof array/ sizeof array[0].**

Если операндом является параметр, объявленный как тип массива или функции, то возвращается размер только указателя. К функциям операция **sizeof** не применима.

Если операция **sizeof** применяется к структуре или объединению, она возвращает общий объем памяти, включая все наполнение этого объекта.

1.9.11 Операция typeid

Операция **typeid** возвращает информацию времени выполнения **type_info,** о типе или выражении. Операция имеет две формы:

```
typeid( выражение )
typeid( тип )
```

Если операндом является разыменованный указатель или ссылка на полиморфный тип, операция **typeid** возвращает динамический тип того реального объекта, на который ссылается указатель или ссылка. Если оператор не полиморфный, возвращается статический тип объекта.

1.9.12 Операции адресации (&) и косвенной адресации (*)

При работе с указателями и при передаче в функции параметров по ссылке используются операции "&"— адресации, и "*" — косвенной адресации или разыменования. Применение этих операций при работе с указателями подробно рассмотрено в разд. 2.8. Применение их при передаче параметров в функции рассмотрено в разд. 1.7.2.

1.9.13 Операции разрешения области действия (::)

Операции разрешения области действия обозначаются двумя двоеточиями, записываемыми без пробела "::". Имеется две различных операции:

```
унарная:

:: переменная

и бинарная:

класс :: элемент_класса
```

Унарная операция разрешения области действия позволяет получить доступ к глобальной переменной из блока, в котором объявлена локальная переменная с тем же именем. Например, выражение **::I** означает глобальную переменную I, даже если в данном блоке или в одном из обрамляющих блоков объявлена локальная переменная I. Подробнее об областях действия (видимости) см. в разд. 1.8.

Бинарная операция разрешения области действия позволяет сослаться на данные-элемент или функцию-элемент класса, даже если имеются одноименные переменные или функции, определенные вне класса или в нескольких классах. Она используется также при описании функции-элемента вне класса. Вы можете увидеть автоматическое применение этой операции в любом модуле, создаваемом C++Builder, если взглянете на заголовок любого обработчика событий.

1.9.14 Операции доступа к элементам: точка (.) и стрелка (->)

Доступ к элементам структур и классов может осуществляться двумя операциями: операцией точки "." или операцией стрелки "->". Если доступ осуществляется через объект, то используется операция точка. Например, если объект с именем A имеет свойство **Prop** и метод **F(),** то доступ к ним дается выражениями:

```
A.Prop
A.F()
```

Если доступ осуществляется через указатель на объект, что чаще всего практикуется для доступа к компонентам в C++Builder, то используется операция стрелка. Например:

```
Label1->Caption
Label1->Hide()
```

Правда, и в случае, если вы имеете указатель на объект, вы можете использовать операцию точка, но тогда вы сначала должны разыменовать указатель:

```
(*Label1).Caption
```

Впрочем, вряд ли подобное усложнение записи целесообразно.

1.9.15 Операции поместить в поток (<<) и взять из потока (>>)

Операции поместить в поток "<<" и взять из потока ">>" предназначены для работы с потоками, как со стандартными потоками **cout** и **cm,** используемыми в основном в консольных приложениях, так и с файлами (см. гл. 2, разд. 2.10.3.1). В приведенных ниже примерах мы будем ориентироваться на то, что создается файловый поток outfile для вывода данных и файловый поток infile для чтения данных. Для этого должны быть выполнены операторы

```
#include <fstream.h>
// создание потока outfile, связанного с файлом "Test.dat"
ofstream outfile("Test.dat");
if(!outfile)
 {
ShowMessage("Файл не удается создать");
  return;
 }

...   // операторы поместить в поток

outfile.close();    // закрытие файла
// создание потока infile, связанного с файлом "Test.dat"
ifstream infile("Test.dat");
if(!infile)
 {
ShowMessage("Файл не удается открыть");
  return;
 }

...   // операторы взять их потока

infile.close();    // закрытие файла
```

Пояснения этих операторов см. в гл. 2, в разд. 2.10.3.1.

Вывод в потоки может быть выполнен с помощью операции поместить в поток, т.е. перегруженной операции "<<". Операция "<<" перегружена для вывода элементов данных встроенных типов, для вывода строк и вывода значений указателей. Она позволяет также с помощью манипуляторов потока осуществлять вывод целых чисел в десятичном, восьмеричном и шестнадцатеричном форматах, вывод значений с плавающей запятой с различной точностью, с указанием по выводу десятичной точки, в экспоненциальном формате или в формате с фиксированной точкой, вывод данных с выравниваем относительно какой-либо границы поля указанной ширины, вывод данных с полями, заполненными заданными символами, вывод буквами в верхнем регистре в экспоненциальном формате и при выводе шестнадцатеричных чисел.

Операция "<<" помещает в поток, являющийся ее первым операндом, аргумент, являющийся ее вторым операндом. Размещение в потоке происходит в текстовом виде. Например, оператор

```
outfile << "Привет!";
```

поместит в файл текст "Привет!". Операторы

```
int i = 25;
outfile << i;
```

поместят в файл текст "25".

Операция "<<" возвращает ссылку на объект своего первого операнда, т.е. на поток. Это позволяет использовать сцепленные операции поместить в поток, например, оператор

```
outfile << "2 * 2 = " << (2 * 2);
```

поместит в файл текст "2 * 2 = 4". Это произойдет потому, что левая операция << поместит текст "2 * 2 = " и вернет **outfile,** после чего правая операция << будет иметь вид-

```
outfile << (2 * 2);
```

и добавит к тексту результат своего правого операнда.

Проверить работу этого и рассматриваемых далее операторов можно, например, в следующем тестовом приложении. Разместите на форме компонент **Memo,** кнопку и в обработчик ее события **OnClick** вставьте операторы:

```
char sin [80];

ofstream outfile("Test.dat");
if(!outfile)
{
 ShowMessage("Файл не удается создать");
 return;
}

// операторы записи в файл, например:
outfile << "2 * 2 = " << 2 * 2;
// закрытие файла
outfile.close();

// открытие файла как входного потока
ifstream infile("Test.dat");
if(!infile)
{
 ShowMessage("Файл не удается открыть");
 return;
}
Memo1->Clear();
while(!infile.eof())
{
 infile.getline(s1,80);
 Memo1->Lines->Add(AnsiString(s1));
}
// закрытие файла
infile.close();
```

Подробное пояснение этих операторов вы найдете в разд. 2.10.3.1. А смысл их сводится к тому, что создается файл "Test.dat", связанный с потоком **outfile,** затем в него заносится операциями "<<" некоторый текст, после чего файл закрывается. Затем он опять открывается, связываясь с потоком **infile,** и строки из него считываются и переносятся в окно **Memo1.**

Продолжим рассмотрение операции "<<". Последовательное применение операций поместить в поток (сцепленных или задаваемых самостоятельными операторами) приводит к занесению текстов в одну строку, как в рассмотренном выше примере. Если требуется перейти на новую строку, то можно или ввести в текст символ конца строки "\n" или применить манипулятор потока **endl** (сокращение от **end line** — конец строки). Например, операторы

```
outfile << "2 * 2 :\n" << (2 * 2) << "\n";
```

и

```
outfile << "2 * 2 :" << endl << (2 * 2) << endl;
```

дадут один и тот же результат: первая строка будет содержать текст "2 * 2 :", вторая - "4", а курсор файла будет переведен на третью строку.

В предыдущих примерах выводились константы и константные выражения. При выводе переменных все работает точно так же. Например, операторы

```
int i = 25, j = 2;
outfile << i << " * " << j << " = " << (i * j) << endl;
```

и операторы

```
int i = 25, j = 2;
char s[80] = "25 * 2 = ";
outfile << s << (i * j) << endl;
```

выводят в файл один и тот же текст: "25 * 2 = 50".

Предыдущий пример показывает, что вывод строки типа **char** * осуществляется просто записью в качестве правого операнда указателя на эту строку. Однако для строк типа **AnsiString** (см. разд. 2.5.2, 3.1.6, 3.4.2.3) операция "<<" не перегружена. Поэтому при выводе таких строк надо использовать приведение их к типу char * с помощью метода **c_str:**

```
' AnsiString sa = "Это строка AnsiString";
outfile << sa.c_str() << endl;
```

При выводе могут использоваться и достаточно сложные выражения. В приведенном ниже примере предполагается наличие двух окон редактирования **Edit1** и Edit2, в которые пользователь вводит целые числа, а программа выводит результат их сравнения:

```
int i = StrToInt(Edit1->Text);
int j = StrToInt(Edit2->Text);
outfile << i << (i == j ? " " : " не ") << "равно " << j
        << endl;
```

В зависимости от введенных чисел будет выведен текст "... равно ..." или "... не равно ...".

Обратите внимание на то, что условный оператор заключен в скобки. Это необходимо делать, поскольку операция поместить в поток имеет сравнительно высокий приоритет (см. разд. 1.9.16) и без скобок она применилась бы только к переменной i, что вызвало бы сообщение о синтаксической ошибке.

Операция "<<" позволяет выводить и указатели. Например, вы можете написать оператор

```
outfile << Memo1 endl;
```

и он выведет текст типа "0063F610" — шестнадцатеричный адрес объекта Memo1. Особым приемом надо выводить при необходимости указатель на строку типа **char** *. Если записать в операции поместить в поток сам указатель, например, s, то выведется не указатель, а содержимое строки. Так перегружена операция "<<" при выводе строк. Если же нужен именно адрес, то перед именем указателя надо поместить операцию приведения типа (void *). Например:

```
outfile << (void *)s << endl;
```

Рассмотренные выше примеры далеко не исчерпывают возможностей вывода с помощью операции "<<". При выводе можно использовать немало манипуляторов потоков, позволяющих форматировать текст, выводимый операцией "<<". Ранее был рассмотрен только один манипулятор потока — endl. Описание других манипуляторов приведено в гл. 2, в разд. 2.10.3.2.

Теперь остановимся на операции взять из потока ">>". Эта операция извлекает данные из потока, заданного ее левым операндом, и заносит их в переменную,

заданную правым операндом. Операция возвращает поток, указанный как ее левый операнд. Благодаря этому допускаются сцепленные операции взять из потока. Например, оператор

```
infile >> i >> j;
```

прочтет, начиная с текущей позиции файла, связанного с потоком infile, два целых числа в переменные i и j. Если в текущей позиции файла первому из чисел предшествуют пробельные символы или разделители, то они будут пропущены. За окончание числа операция примет первый отличный от цифры символ, в частности, пробельный. Поэтому, если эти два числа были ранее записаны в файл например, оператором

```
outfile << i << ' ' << j << endl;
```

то они прочтутся нормально. Но если они были записаны оператором

```
outfile << i << j << endl;
```

т.е. без пробела, то их цифры будут слиты вместе и это составное число прочтется как i, а при чтении j произойдет ошибка.

Операцией ">>" можно вводить из файла строки в переменные типа **char** *. Например, операторы:

```
char s[80];
infile >> s;
```

осуществляют чтение из файла в строку s. Но при этом читается не вся строка, а только одна лексема — последовательность символов, заканчивающаяся пробельным или разделительным символом.

Если при выполнении операции взять из потока считывается символ конца потока, то операция возвращает 0. Этим можно воспользоваться, чтобы, например, читать все содержимое файла, разбитое на лексемы:

```
while(infile>>s1)
{
 ...
}
```

1.9.16 Приоритет и ассоциативность операций

В сложных выражениях, содержащих несколько операций, последовательность их выполнения определяется прежде всего приоритетом операций. Имеется 16 уровней приоритета, приведенных ниже в таблице. Некоторые из этих уровней содержат всего по одной операции. Наивысший уровень имеют операции, приведенные в первой строке таблицы, низший — в последней. Операции, указанные в одной строке, имеют одинаковый уровень старшинства.

Там, где в таблице встречаются дубликаты операций (например, дубликаты имеют операции сложения и вычитания), первая относится к унарной операции, а вторая — к бинарной.

Если в выражении встречаются записанные подряд операции одного уровня старшинства, то последовательность их выполнения определяется ассоциативностью, которая может быть слева направо или справа налево.

Все операции, перечисленные в таблице, были рассмотрены в предыдущих разделах, кроме операций new и delete, которые будут рассмотрены в разд. 1.11.

Операция	Ассоциативность
0 [] -> ::	слева направо
! ~ + - ++ — & * sizeof new delete	справа налево

Операция	Ассоциативность
.* -> *	слева направо
* / %	слева направо
+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&	слева направо
^	слева направо
|	слева направо
&&	слева направо
||	справа налево
?:	слева направо
= *= /= %= += -= &= ^= |= <<= >>=	справа налево
,	слева направо

Например, выражение a + b * c / d будет выполняться как a + ((b * c) / d). Сначала выполнятся операции умножения и деления, имеющие более высокий приоритет, чем операция сложения. Поскольку ассоциативность операций умножения и деления слева направо, то прежде всего будет выполнено умножения b * c, а затем результат разделится на c. В заключение результат этого деления прибавится к a.

Вы можете легко изменять последовательность действий, применяя скобки, которые имеют очень высокий приоритет.

1.9.17 Перегрузка операций

Все операции C++ могут быть перегружены, кроме операций точка ".", разыменование "*", разрешение области действия "::", условная "?:" и **sizeof**.

Операции "=", "[]", "()" и "->" могут быть перегружены только как нестатические функции-элементы. Они не могут быть перегружены для перечислимых типов.

Все остальные операции можно перегружать, чтобы применять их к каким-то новым типам объектов, вводимым пользователем. Кроме того, многие операции уже перегружены в C++. Например, арифметические операции применяются к разным типам данных — целым числам, действительным и т.д., именно в результате того, что они перегружены.

Операции перегружаются путем составления описания функции (с заголовком и телом), как это делается для любых функций, за исключением того, что в этом случае имя функции состоит из ключевого слова **operator,** после которого записывается перегружаемая операция. Например, имя функции **operator+** можно использовать для перегрузки операции сложения.

Чтобы использовать операцию над объектами классов, эта операция *должна* быть перегружена, но есть два исключения. Операция присваивания "=" может быть использована с каждым классом без явной перегрузки. По умолчанию операция присваивания сводится к побитовому копированию данных-элементов класса. Такое побитовое копирование опасно для классов с элементами, которые указывают на динамически выделенные области памяти; для таких классов следует явно

перегружать операцию присваивания. Операция адресации "&" также может быть использована с объектами любых классов без перегрузки; она просто возвращает адрес объекта в памяти. Но операцию адресации можно также и перегружать.

Перегрузка не может изменять старшинство и ассоциативность операций. Нельзя также изменить число операндов операции. Например, унарную операцию можно перегрузить только как унарную.

Перегрузка больше всего подходит для математических классов. Они часто требуют перегрузки значительного набора операций, чтобы обеспечить согласованность со способами обработки этих математических классов в реальной жизни. Например, было бы странно перегружать только сложение класса комплексных чисел, потому что обычно с комплексными числами используются и другие арифметические операции.

Цель перегрузки операций состоит в том, чтобы обеспечить такие же краткие выражения для типов, определенных пользователем, какие C++ обеспечивает с помощью богатого набора операций для встроенных типов. Однако перегрузка операций не выполняется автоматически; чтобы выполнить требуемые операции, программист должен написать функции, осуществляющие перегрузки операций.

Ниже приведен упрощенный пример создания класса комплексных чисел **Complex,** в котором переопределены операции сложения, вычитания и присваивания. Дается описание не всех функций, поскольку очевидно, что сложение и вычитание — операции идентичные с точностью до знака.

```
class Complex {
public:
 double Re;                                  // действительная часть
 double Im;                                  // мнимая часть
 Complex(double = 0.0, double = 0.0);        // конструктор
// операции сложения
 Complex operator+(const Complex &) const;   // бинарная
 Complex operator+() const;                  // унарная
// операции вычитания
 Complex operator-(const Complex &) const;   // бинарная
 Complex operator-() const;                  // унарная
 Complex &operator=(const Complex &);        // присваивание
};

// Конструктор
Complex::Complex(double R, double I)
 (
 Re = R;
 Im = I;
}
// Перегруженная бинарная операция сложения
Complex Complex::operator+(const Complex &X) const
{
 Complex R;
 R.Re = Re + X.Re;
 R.Im = Im + X.Im;
 return R;
}
...
// Перегруженная унарная операция вычитания
Complex Complex::operator-() const
{
 Complex R;
 R.Re = -Re;
 R.Im = -Im;
 return R;
}
// Перегруженная операция присваивания
```

```
Complex & Complex::operator=(const Complex SR)
{
 Re = R.Re;
 Im = R.Im;
 return *this; // возможность сцепления
}
```

В этом классе вводится два открытых данных-элемента: Re — действительная часть комплексного числа, и Im — мнимая часть. Конструктор по умолчанию задает действительную и мнимую части равными 0.

Оператор

```
Complex operator+(const Complex &) const;
```

объявляет прототип бинарной операции сложения. На то, что это операция бинарная, указывает наличие параметра — правого операнда. Когда компилятор встретит в тексте операцию "A + B", примененную к переменным типа **Complex,** он, незримо для пользователя, заменит ее выражением **"A.operator+(B) "**.

Оператор

```
Complex operator+() const;
```

объявляет прототип унарной операции сложения, поскольку список параметров пуст.

Прототипы операций вычитания и присваивания строятся аналогично.

В реализации функции бинарного сложения создается локальная переменная R типа **Complex,** в которой формируется возвращаемое значение. В процессе формирования к данным левого операнда производится обращение просто по именам Re и Im, а второй операнд является параметром, передаваемым в функцию по ссылке. В заключение значение сформированной переменной возвращается как результат функции.

В функции операции присваивания данные параметра просто пересылаются в поля Re и Im. Обратите внимание на последнюю строку, которая возвращает *this. Указатель this является указателем на объект данного класса. Подобный возврат ссылки необходим, чтобы можно было использовать сцепленные операции присваивания. Рассмотрим это подробнее.

Если компилятор встречает в тексте выражение A = B, примененное к переменным типа **Complex,** он заменяет его выражением **A.operator=(B).** А что произойдет, если встретится выражение A = B = C ? Поскольку ассоциативность операции присваивания справа налево, то сначала заменится вторая часть выражения на **B.operator=(C).** После замены первого знака равенства получится выражение **A.operator=(B.operator=(C)).** Для того чтобы это работало, нужно, чтобы выражение возвращало ссылку на объект B. Это и делается, возвращением в функции ссылки *this. Тогда обеспечивается правильное выполнение сцепленных присваиваний.

С описанным классом вы можете, например, выполнять такие действия:

```
Complex A(1,1), B(2,2),C,D;
A = -A;
C = A + B + B;
D = A - B;
A = B = C;
```

1.10 Операторы

1.10.1 Операторы передачи управления

1.10.1.1 Условные операторы выбора if

Оператор if предназначен для выполнения тех или иных действий в зависимости от истинности или ложности некоторого условия. Условие задается выражением, имеющим результат булева типа.

Оператор имеет две формы: **if** и **if...else.** Форма if имеет вид:

```
if (условие) оператор;
```

Скобки, обрамляющие условие, обязательны.

Условием может быть выражение, преобразуемое в булев тип. Если условие истинно (возвращает **true** — ненулевое значение), то указанный в конструкции if оператор выполняется. В противном случае управление сразу передается следующему за конструкцией if оператору. Например, в результате выполнения операторов

```
C = A;
if (B > A) C = B;
```

переменная C станет равна максимальному из чисел A и B, поскольку оператор C = B будет выполнен только при B > A.

Поскольку в C++ арифметическое (целое или действительное) значение может преобразовываться к булеву (любое ненулевое значение воспринимается как **true,** а нулевое — как false), то условие может иметь целый тип. Например:

```
int a, b, c;
...
if(a - b/c) ...;
```

В данном случае условие **if(a — b/c)** эквивалентно **if(a == b/c),** поскольку a — b/c возвращает нуль при равенстве a и b/c. Аналогичные условия формально можно записывать и для действительных чисел:

```
double a, b, c;
...
if(a - b/c) ...;
```

Но из-за ошибок округления это может не сработать, даже если теоретически значения a и b/c должны совпадать.

В условии можно объявлять переменные. Например:

```
if (int v = func(a)) ...;
```

В этом случае область действия и существования объявленной переменной – только данная структура **if,** включая ее выполняемый оператор.

Форма конструкции **if...else** имеет вид:

```
if (условие) оператор1;
else оператор2;
```

Если условие возвращает **true,** то выполняется первый из указанных операторов, в противном случае выполняется второй оператор. Обратите внимание, что в конце первого оператора перед ключевым словом **else** ставится точка с запятой.

Приведем примеры.

```
if (J == 0)
  ShowMessage("Деление на нуль");
else
  Result = I/J;
```

В качестве и первого, и второго оператора могут, конечно, использоваться и составные операторы:

```
if (J == 0)
 {
  ShowMessage("Деление на нуль");
  Result = 0;
 }
else
  Result = I/J;
```

Опять обратите внимание, что после фигурной скобки перед **else** точка с запятой не ставится.

При вложенных конструкциях **if** могут возникнуть неоднозначности в понимании того, к какой из вложенных конструкций **if** относится элемент **else.** Компилятор всегда считает, что **else** относится к последней из конструкций **if,** в которой не было раздела **else.**

Например, в конструкции

```
if (условие1)
if (условие2)
оператор1;
  else оператор2;
```

else будет отнесено компилятором ко второй конструкции **if,** т.е. **оператор2** будет выполняться в случае, если первое условие истинно, а второе ложно. Иначе говоря, вся конструкция будет прочитана как

```
if (условие1)
 {
   if (условие2) оператор1;
else оператор2;
 }
```

Если же вы хотите отнести **else** к первому **if,** это надо записать в явном виде с помощью фигурных скобок:

```
if (условие1)
  !
    if (условие2) оператор1;
  }
else оператор2;
```

1.10.1.2 Условный оператор множественного выбора switch

Оператор **switch** позволяет провести анализ значения некоторого выражения и в зависимости от его значения выполнить те или иные действия. В общем случае формат записи оператора **switch** следующий:

```
switch (выражение_выбора) {
case значение_1 : оператор_1;
                  break;        // не обязательно
  ...
case значение_n : оператор_n;
                  break;        // не обязательно
default : оператор;             // не обязательно
}
```

В этой конструкции выражение выбора должно иметь порядковый тип — целый, перечислимый и т.д. Поэтому, например, нельзя использовать выражения, возвращающие действительные числа или строки.

Значения, указываемые в метках **case,** должны быть константными выражениями, соответствующими возможным значениям выражения выбора. После значения ставится двоеточие ":", а затем пишется оператор (может писаться состав-

ной оператор), который должен выполняться, если выражение приняло указанное в метке значение.

Если значение выражения выбора совпало со значением, указанным в одной из меток **case,** то выполняется оператор, записанный после этой метки, после чего, если не принять соответствующих мер, будут выполняться все последующие операторы остальных меток. Поскольку это обычно нежелательно, то, как правило, после оператора, который должен выполняться, записывают оператор

```
break;
```

Он прерывает выполнение структуры **switch** и управление передается следующему за ней оператору.

Если значение выражения выбора не соответствует ни одному из перечисленных в метках, то выполняется оператор, следующий за меткой **default.** Впрочем, метка **default** не обязательно должна включаться в структуру **switch.** В этом случае, если не нашлось соответствующего значения выражения выбора, то ни один оператор не будет выполнен.

Значения в метках могут содержать константы и константные выражения, которые совместимы по типу с объявленным выражением и которые компилятор может вычислить заранее, до выполнения программы. Недопустимо использование переменных и многих функций. В метках не допускается повторение одних и тех же значений, поскольку в этом случае выбор был бы неоднозначным.

Приведенный ниже пример анализирует переменную Key типа **char,** содержащую символ, введенный пользователем в ответ на некоторый вопрос. При положительном ответе вызывается процедура **FYes,** при отрицательном — FNo, при иных ответах отображается сообщение об ошибке.

```
switch (Key) {
case 'y': case 'Y': { FYes(); break; }
case 'n': case 'N': { FNo(); break; }
default :  ShowMessage("Ошибочный ответ");
}
```

Обратите внимание, что при необходимости выполнять одинаковые действия при нескольких значениях выражения выбора, надо размещать подряд несколько меток case.

1.10.1.3 Оператор передачи управления goto

Оператор **goto** позволяет прервать обычный поток управления и передать управление в произвольную точку кода, помеченную специальной меткой. В свое время при появлении концепции структурного программирования на оператор **goto** обрушился поток критики и его применение стало рассматриваться как дурной тон. Действительно, чрезмерно широкое применение **goto** делает структуру программы крайне запутанной и затрудняет ее сопровождение. Однако во многих случаях стремление обойтись без оператора **goto** не только не упрощает код, а еще более его запутывает. Так что этот оператор, безусловно, имеет право на существование.

Метка в тексте программы обозначается идентификатором с последующим двоеточием. Например,

```
Lbegin:
```

Метка отмечает точку, в которую передается управление оператором **goto.** Метка может располагаться в любом месте блока, как после оператора **goto,** передающего на нее управление, так и до этого оператора. Надо только иметь в виду, что передача управления извне внутрь цикла может приводить к непредсказуемым последствиям, так что таких ситуаций следует избегать.

Метки имеет областью действия функцию. Метки можно использовать всюду в функции, в которой они появились, но на них нельзя ссылаться вне тела функ-

ции. Метки используются также в структурах **switch** (как метки **case** -- см. разд. 1.10.1.2).

После метки следует оператор, на который передается управление.

Сам оператор **goto** имеет форму:

```
goto метка;
```

Таким образом, организация работы с операторами **goto** может выглядеть, например, так:

```
goto L1;
...
second: ...
...
L1: ...
...
if (...) goto L1;
else goto second;
```

При этом, как видно, можно ссылаться на метки, расположенные после или до оператора **goto.**

1.10.2 Операторы циклов

1.10.2.1 Оператор for

Оператор **for** обеспечивает циклическое повторение некоторого оператора (в частности, составного оператора) заданное число раз. Повторяемый оператор называется *телом цикла.* Повторение цикла обычно определяется некоторой управляющей переменной (счетчиком), которая изменяется при каждом выполнении тела цикла. Повторение завершается, когда управляющая переменная достигает заданного значения.

Синтаксис структуры **for:**

```
for (выражение1; выражение2; выражение3) оператор;
```

где выражение1 задает начальное значение переменной, управляющей циклом, выражение2 является условием продолжения цикла, а выражение3 изменяет управляющую переменную.

Структура **for** работает следующим образом. Сначала выполняется выражение1 (оно может состоять и из ряда выражений, разделенных запятой т.е. может использоваться операция последования — см. разд. 1.9.8). Это выражение задает начальные значения переменной (или переменных) цикла. Затем проверяется выражение2 — условие продолжения цикла. Если условие истинно (возвращает **true** — ненулевое значение), то выполняется тело цикла — оператор, записанный в структуре **for.** После завершения тела цикла выполняется выражение3, определяющее обычно изменение переменной цикла. Затем опять проверяется условие, записанное как выражение2, и при истинности этого условия выполнение цикла продолжается. Как только в каком-нибудь цикле выражение2 вернет **false** (нулевое значение), цикл прерывается и управление передается оператору, расположенному следом за структурой **for.**

Приведем примеры использования цикла **for.** Следующие операторы вычисляют максимальное значение и сумму элементов, расположенных в массиве целых чисел **Data** размерностью 10:

```
int Max, Sum;
Max = Sum = Data[0];
for(int i = 1; i < 10; i++)
   {
    if (Data[i] > Max) Max = Data[i];
    Sum += Data[i];
   }
```

Здесь первое выражение в структуре **for** вводит целую переменную i, являющуюся счетчиком циклов, и инициализирует ее значением 1. Второе выражение проверяет условие завершения цикла. В данном случае цикл должен завершиться, когда переменная **i,** используемая в теле цикла как индекс массива, примет значение, большее 9. Третье выражение структуры **for** увеличивает после каждого выполнения цикла значение i на 1 с помощью операции инкремента.

В данном случае переменная i объявлена в заголовке структуры **for.** Значит ее область действия только эта структура. После завершения циклов переменная i удаляется из памяти.

При использовании компилятора **BCC32.EXE,** запускаемого из командной строки, подобное уничтожение локальной переменной, объявленной в цикле, можно отменить опцией **-Vd.** Но вряд ли это имеет смысл делать

Теперь рассмотрим пример использования в структуре **for** операции запятая (см. разд. 1.9.8). Пусть в приведенном выше примере нам надо найти только сумму элементов массива. Тогда, если объявить переменные i и Sum до начала цикла, собственно цикл можно весь разместить в заголовке структуры **for:**

```
int Sum,i;
for(Sum = Data[0],i = 1; i < 10; Sum += Data[i++]);
```

В этом примере первое выражение структуры for включает в себя два оператора, разделенных операцией запятая и задающих начальные значения переменной Sum, накапливающей сумму, и переменной цикла i. Третье выражение структуры **for** объединяет в одном операторе формирование суммы и постфиксный инкремент переменной цикла i. После структуры **for** стоит точка с запятой, что означает пустое тело цикла.

В приведенных примерах переменная цикла увеличивалась на единицу при каждом цикле. Можно, конечно, организовывать циклы с уменьшением переменной. Ниже приведен такой пример. В нем приведена программа, которая берет строку, записанную в окне редактирования **Edit1,** шифрует ее сложением по операции исключающее ИЛИ каждого символа строки с произвольным ключом и возвращает строку в окно редактирования. Если повторно применить эту процедуру с тем же ключом к зашифрованной строке, то будет произведена дешифровка и в окне отобразится исходная строка.

```
AnsiString s;
char Key = 'A';
s = Edit1->Text;
for (int i = s.Length(); i >0; s[i--] = s[i] ^ Key);
Edit1->Text = s;
```

, В этом примере начальное значение переменной i задано равным числу символов в строке, полученному применением функции **Length().** В дальнейшем при каждом выполнении цикла i уменьшается на 1 постфиксной операцией декремента. В этом случае целесообразно именно уменьшение счетчика цикла, поскольку в противном случае во втором выражении структуры **for** пришлось бы каждый раз проверять с помощью функции **Length(),** не кончилась ли строка. А в приведенном варианте обращение к **Length()** производится всего один раз.

Выражения в структуре **for** являются необязательными. Иногда может отсутствовать первое выражение, если начальное значение управляющей переменной задано где-то в другом месте программы. Если отсутствует второе выражение, предполагается, что условие продолжения цикла всегда истинно и таким образом создается бесконечно повторяющийся цикл. Выйти из такого цикла можно, проверив в теле цикла какие-то условия и прервав выполнение передачей управления за пределы цикла оператором **goto** или применить другие способы прерывания, рассмотренные в разд. 1.10.2.4. Может отсутствовать в структуре for и третье выражение, если приращение переменной осуществляется операторами в теле структуры или если приращение не требуется.

При пропуске какого-то из выражений, точка с запятой после пропущенного выражения (кроме третьего) должна писаться. Например, в заголовке

```
for(; i<10;) ...;
```

пропущено первое условие и третье.

Если условие продолжения цикла не удовлетворяется с самого начала, то операторы тела структуры **for** не выполняются ни разу.

Операторы цикла **for** могут быть вложенные. Следующий пример содержит три вложенных цикла **for,** осуществляющих вычисление матрицы Mat, равной произведению двух квадратных матриц **Mat1** и **Mat2** размером M на M. Все матрицы представлены двумерными массивами. Формула для вычисления:

$$\mathrm{Mat}[I,J] = \sum_{K=1}^{M} \mathrm{Mat1}[I,K] \cdot \mathrm{Mat2}[K,J].$$

```
int I, J, K, X;
for(I = 1; I <= M; I++)
  for(J = 1; J <= M; J++)
  {
   X = 0;
   for(K = 1; K <= M; K++)
       X += Mat1[I][K] * Mat2[K][J];
   Mat[I] [J] = X;
  }
```

1.10.2.2 Оператор do...while

Структура **do...while** используется для организации циклического выполнения оператора или совокупности операторов, называемых телом цикла, до тех пор, пока не окажется нарушенным некоторое условие. Синтаксис управляющей структуры **do...while:**

```
do оператор while (условие);
```

Структура работает следующим образом. Выполняется оператор тела цикла. Затем вычисляется условие — выражение, которое должно возвращать результат булева типа. Если выражение возвращает **true** (не нулевое значение), то повторяется выполнение тела цикла и после этого снова вычисляется выражение. Такое циклическое повторение цикла продолжается до тех пор, пока проверяемое выражение не вернет **false** (нуль). После этого цикл завершается и управление передается оператору, следующему за структурой **do...while.**

Поскольку проверка выражения осуществляется после выполнения тела цикла, то цикл будет заведомо выполнен хотя бы один раз, даже если выражение сразу ложно. С другой стороны, программист должен быть уверен, что выражение рано или поздно вернет false. Если этого не произойдет, то программа «зациклится», т.е. цикл будет выполняться бесконечно. Иногда такие бесконечные циклы используются. Но в этом случае внутри тела цикла должно быть предусмотрено его прерывание в какой-то момент, например, оператором **break** или другими способами, рассмотренными в разд. 1.10.2.4.

Обычно оператор **do** целесообразно использовать для организации поиска среди множества объектов такого, который обладает каким-то определенным свойством. Причем заранее должно быть известно, что множество объектов не пустое, т.е. хотя бы один объект в нем имеется. К тому же должен быть критерий, позволяющий проверить, не является ли текущий объект последним. Тогда тело цикла включает операторы перехода к новому объекту и какой-то его обработки, а условие **while** включает проверку, является ли объект не последним и отсутствуют ли у него искомые свойства. Если объект последний или искомые свойства найдены, выполнение цикла прерывается. Если же объект не последний и искомые свойства у него не найдены, осуществляется переход к следующему объекту.

Если множество проверяемых объектов может быть пустым, следует использовать другой оператор цикла — **while** (см. разд. 1.10.2.3). Если число повторений циклов заранее известно, лучше применять оператор **for** (см. разд. 1.10.2.1).

Ниже приведен пример, в котором в файле **Filel.txt** ищется строка, содержащая фрагмент текста (последовательность символов с учетом регистра), указанный пользователем в окне редактирования **Edit1**. Проверка наличия в строке заданного фрагмента проверяется функцией **strstr.** Окончание файла проверяется функцией **feof.**

```
FILE *F;
char S[256] = "";
AnsiString SKey = Edit1->Text;
if((F = fopen("File1.txt","r")) == NULL)
{
 ShowMessage("Файл не найден");
 return;
}

do
 fgets(S,256,F);
while ( !feof(F)S& (strstr(S,SKey.c_str() ) == NULL) ) ;

fclose(F);
if (strstr(S,SKey.c_str()) == NULL)
  ...
```

Цикл будет выполняться, до тех пор, пока не достигнут конец файла и пока функция **strstr** возвращает **NULL** (фрагмент не найден). Если хотя бы одно из этих условий нарушается (достигнут конец файла или найден фрагмент), выполнение цикла прекращается.

1.10.2.3 Оператор while

Оператор **while** используется для организации циклического выполнения тела цикла, пока выполняется некоторое условие. Синтаксис структуры **while:**

```
while (условие) оператор;
```

Структура работает следующим образом. Сначала вычисляется условие, которое должно возвращать результат булева типа. Если выражение возвращает **true** (ненулевое значение), то выполняется оператор тела цикла, после чего опять вычисляется выражение, определяющее условие. Такое циклическое повторение выполнения оператора и проверки условия продолжается до тех пор, пока условие не вернет false (нуль). После этого цикл завершается, и управление передается оператору, следующему за структурой **while.**

Поскольку проверка выражения осуществляется перед выполнением оператора тела цикла, то, если условие сразу ложно, оператор не будет выполнен ни одного раза.

Программист должен быть уверен, что выражение рано или поздно вернет false. Если этого не произойдет, то программа «зациклится», т.е. цикл будет выполняться бесконечно. Иногда такие бесконечные циклы используются. Но в этом случае внутри тела цикла должно быть предусмотрено его прерывание в какой-то момент, например, оператором **break,** прерывающим цикл, или другими способами, рассмотренными в разд. 1.10.2.4.

Часто оператор **while** используется для организации поиска среди множества объектов такого, который обладает каким-то определенным свойством. Причем не исключается, что множество объектов может быть пустым, т.е. не содержащим ни одного объекта. К тому же должен быть критерий, позволяющий проверить, не является ли текущий объект последним. Тогда тело цикла включает операторы перехода к новому объекту и какой-то его обработки, а условие **while** включает проверку, является ли объект не последним и не обладает ли он искомым свойством. Если

одно из этих условий нарушается (объект последний или имеет искомое свойство), выполнение цикла прерывается.

Ниже повторен приведенный в предыдущем разделе пример поиска в файле **Filel.txt** фрагмента текста, указанного пользователем в окне редактирования **Edit1.** Но если в предыдущем разделе для организации цикла использовался оператор **do...while,** то в данном случае использован оператор **while.** Этот оператор здесь более уместен, поскольку проверка конца файла осуществляется до начала цикла, т.е. до чтения из него строки. Поэтому все будет нормально работать даже в случае, если файл окажется пустым и в нем не будет ни одной строки.

```
FILE *F;
char S[256] = "";
AnsiString SKey = Edit1->Text;
if((F = fopen("File1.txt","r")) == NULL)
(
  ShowMessage("Файл не найден");
  return;
}

while(!feof(F)&&(strstr(S,SKey.c_str()) == NULL))
  fgets(S,256,F);

fclose (F);
if (strstr(S,SKey.c_str()) == NULL)
  ...
```

В данном случае можно использовать и цикл **for:**

```
for(; !feof(F) && (strstr (S,SKey.c_str() ) == NULL) ;
      fgets (S,256,F));
```

но цикл **while** выглядит наиболее естественным.

1.10.2.4 Прерывание цикла: операторы break, Continue, return, функция Abort

В некоторых случаях желательно прервать повторение цикла, проанализировав какие-то условия внутри него. Это может потребоваться в тех случаях, когда проверки условия окончания цикла громоздкие, требуют многоэтапного сравнения и сопоставления каких-то данных и все эти проверки просто невозможно разместить в выражении условия операторов **for, do** или **while.**

Один из возможных вариантов решения этой задачи — ввести в код какой-то флаг окончания (переменную). При выполнении всех условий окончания этой переменной присваивается некоторое условное значение. Тогда условие в операторах **for, do** или **while** сводится к проверке, не равно ли значение этого флага принятому условному значению.

Другой способ решения задачи — использование оператора **break.** Он используется как в операторах цикла, так и в структурах **switch.** Оператор **break** прерывает выполнение тела любого цикла **for, do** или **while** и передает управление следующему за циклом выполняемому оператору.

Например, цикл в рассмотренном в предыдущих разделах примере поиска текста в файле мог бы быть организован следующим образом:

```
while(!feof(F))
{
 fgets(S,256,F);
 if(strstr(S,SKey.c_str()) != NULL) break;
}
```

Еще один способ прерывания цикла — использование оператора **goto,** передающего управление какому-то оператору, расположенному вне тела цикла.

Для прерывания циклов, размещенных в функциях, можно воспользоваться оператором **return.** В отличие от оператора **break,** оператор **return** прервет не только выполнение цикла, но и выполнение той функции, в которой расположен цикл.

Прервать выполнение цикла, а заодно — и блока, в котором расположен цикл, можно также генерацией какого-то исключения (см. разд. 1.12). Наиболее часто в этих целях используется процедура **Abort,** генерирующая «молчаливое» исключение, не связанное с каким-то сообщением об ошибке.

Описанные способы прерывали выполнение цикла. Имеется еще процедура **Continue,** которая прерывает только выполнение текущей итерации, текущего выполнения тела цикла и передает управление на следующую итерацию.

Чтобы продемонстрировать применение **Continue,** усложним рассмотренный ранее пример поиска заданного фрагмента в текстовом файле. Пусть, например, мы хотим найти заданный фрагмент не в любой строке файла, а только в такой, которая начинается с символа "*". Тогда поиск можно было бы организовать следующим образом:

```
while(!feof(F))
{
 fgets(S,256,F);
 if(S[0] != '*') continue;
 if(strstr(S,SKey.c_str()) != NULL) break;
}
```

В этом варианте при первом символе в строке, отличном от "*", текущая итерация прерывается и поиск в такой строке не производится. Таким образом, не тратится время на выполнение функции strstr для строк, в которых искать фрагмент не нужно.

1.11 Динамическое распределение памяти

Динамическое распределение памяти широко используется для экономии вычислительных ресурсов. Те переменные или объекты, которые становятся ненужными, уничтожаются, а освобожденное место используется для новых переменных или объектов. Это особенно эффективно в задачах, в которых число необходимых объектов зависит от обрабатываемых данных или от действий пользователя, т.е. заранее не известно. В этих ситуациях остается только два выхода: заранее с запасом отвести место под множество объектов или использовать динамическое распределение памяти, создавая новые объекты по мере надобности. Первый путь, конечно, неудовлетворительный, поскольку связан с излишними затратами памяти и в то же время накладывает на размерность задачи необоснованные ограничения.

Для динамического распределения выделяется специальная область памяти — heap. Динамическое распределение памяти в этой области может производиться несколькими способами: с помощью библиотечных функций **malloc, calloc, realloc, free** или с помощью операций **new** и **delete.**

Указанные функции объявлены в файле **stdlib.h** или **alloc.h.** Объявление функции **malloc** следующее:

```
void *malloc(size_t size);
```

Функция выделяет в heap блок размером в size байтов. В случае успешного выделения памяти функция возвращает указатель на выделенный блок. Если не хватило места для блока требуемого размера или если size = 0, возвращается NULL.

Другая функция — **calloc** объявлена следующим образом:

```
void *calloc(size_t nitems, size_t size);
```

Функция выделяет память под **nitems** объектов, размер каждого из которых равен **size.** Таким образом общий объем выделяемой памяти составляет **nitems * size.**

Выделенная память инициализируется нулями. В случае успешного выделения памяти функция возвращает указатель на выделенный блок. Если не хватило места для блока требуемого размера или если **size** = 0 или **nitems** = 0, возвращается **NULL.**

Еще одна функция — **realloc** позволяет изменить размер ранее выделенного блока памяти. Функция объявлена следующим образом:

```
void *realloc(void *block, size_t size);
```

Она изменяет размер блока в heap, на который указывает **block,** до размера **size.** При этом предполагается, что **block** указывает блок памяти, выделенной ранее функциями **malloc, calloc или realloc.** Если же аргумент **block** задан равным **NULL,** то функция **realloc** работает так же, как описанная выше функция **malloc.**

Если размер **size** задан равным нулю, то выделенный ранее блок, на который указывает **block,** освобождается, а функция возвращает **NULL.** Таким образом, функция с size равным 0 может использоваться не для выделения памяти, а для освобождения памяти, выделенной ранее.

Если блок нового размера не может быть выделен, то функция **realloc** возвращает **NULL.** Если же память выделилась успешно, то возвращается адрес выделенного блока. При этом он может отличаться от начального значения **block,** поскольку функция при необходимости осуществляет копирование содержимого блока в новое место.

Функция **free** объявлена следующим образом:

```
void free(void *block);
```

Она освобождает блок памяти, выделенный ранее функциями **malloc, calloc** или **realloc,** на который указывает **block.**

Рассмотрим примеры использования описанных функций. Следующий код динамически выделяет функцией **malloc** память под строку, а затем, после выполнения с ней каких-то операций, освобождает выделенную память.

```
#include <stdio.h>
#include <alloc.h>
char *str;

// str — указатель на строку, под которую выделена память
str = (char *) raalloc(lOO);
...
// освобождение памяти
free(str);
```

В этом примере можно было бы использовать для выделения памяти функцию **calloc:**

```
str = (char *) calloc(100, sizeof(char));
```

Размер выделенной функциями **malloc** или **calloc** памяти можно было бы изменить, например, следующим оператором:

```
str = (char *) realloc(str, 20);
```

Впрочем, к тому же результату привел бы и более простой оператор:

```
realloc(str, 20);
```

Необходимо помнить, что рассмотренные функции возвращают NULL (O), если память не удалось выделить. Поэтому прежде, чем использовать возвращенные ими указатели, надо обязательно проверять, не равны ли они **NULL.** Иначе возможны очень тяжелые ошибки при работе программы.

Теперь рассмотрим другой подход к динамическому распределению памяти: операции **new** и **delete.**

Операция new работает аналогично функции **malloc,** но лучше использовать именно ее, а не **malloc.** Это пожелание становится безусловной необходимостью,

если речь идет о динамическом размещении в памяти объектов библиотеки компонентов C++Builder.

Операция **new** имеет следующий синтаксис:

```
<::> new <размещение> тип <(инициализатор)>
<::> new <размещение> (тип) <(инициализатор)>
```

Операция возвращает указатель на динамически размещенный в памяти объект.

Все элементы, заключенные в описании синтаксиса в угловые скобки, являются необязательными. Операция разрешения области действия "::" позволяет обратиться к глобальной версии **new,** если наряду с ней возможно использование перегруженных операций. Элемент размещение используется (если он предусмотрен перегруженной версией) для дополнительной информации о месте размещения в памяти. Инициализатор задает начальное значение создаваемого объекта.

Таким образом, обязательно должен быть указан только тип данных. Например:

```
double *A = new double;
```

В данном случае в памяти динамически создается объект — действительное число. В дальнейшем доступ к нему осуществляется как *A. Например:

```
*A = 5.1;
Label1->Caption = *A;
```

Если нет желания вводить указатель на объект и в дальнейшем работать с этим указателем, можно динамически разместить объект с помощью следующего оператора:

```
double B = *new double;
```

В этом случае в дальнейшем на объект можно ссылаться просто по имени — B.

Создание динамически размещенного объекта можно совместить с его инициализацией. Например:

```
double *A = new double(5.1);
double B = *new double(5.5);
```

Ниже приведен пример создания и динамического размещения в памяти компонента — окна редактирования типа **TEdit:**

```
TEdit *Edit = new TEdit(this);
Edit->Parent = Form1;
```

Первый оператор выделяет память под объект и создает его, передавая в него указатель **this** как владельца **Owner.** Второй задает для компонента родителя — **Form1.** В этот момент компонент станет виден на форме.

Рассмотрим подробнее выполнение операции **new** при создании и размещении в памяти объекта. Операция определяет объем необходимой памяти, используя неявно операцию **sizeof(тип).** Если в динамически распределяемой области памяти есть место для размещения объекта, то выделяется соответствующий блок памяти и операция new возвращает указатель на объект данного типа. При этом нет необходимости явно приводить тип этого указателя — все делается автоматически. Созданный объект хранится в памяти, пока не будет уничтожен описанной далее операцией **delete** или пока не завершится выполнение программы.

Если в памяти невозможно выделить блок требуемого размера, генерируется исключение **bad_alloc.** Поэтому в программе всегда надо предусматривать блок **catch** (см. разд. 1.12.5), который бы перехватывал это исключение прежде, чем программа попытается получить доступ к создаваемому объекту. Таким образом, динамическое размещение объектов в памяти, как правило, должно оформляться следующим образом:

```
#include <iostream.h>
try
{
```

```
 Операторы динамического распределения памяти с помощью new
}
catch(std::bad_alloc)
 (
 Операторы действий при недостаточной памяти
}
```

Можно отменить генерацию исключения **bad_alloc,** задавая указатель на свой собственный обработчик событий, связанных с невозможностью выделить память. Для этого используется оператор

```
set_new_handler(указатель)
```

который позволяет задать указатель на обработчик. При этом **set_new_handler** возвращает прежний указатель, который был зарегистрирован до этого.

Например, вы можете описать функцию

```
void F1(void)
{
 ShowMessage("Не хватает памяти");
 exit(1);
}
```

которая обрабатывает ситуацию, связанную с нехваткой памяти, и ввести в программу (например, в обработчик события **OnCreate** формы) оператор

```
set_new_handler(F1);
```

Вводимый таким образом обработчик не может ничего возвращать и должен или освободить память для выполнения new, или сгенерировать исключение **bad_alloc,** или завершить программу (это сделано в приведенном примере). Если не выполнено ни одно из этих действий, возникнет бесконечный цикл обращений к обработчику.

Можно отменить генерацию исключения **bad_alloc,** не вводя специального обработчика, а просто записав оператор

```
set_new_handler(0);
```

В этом случае при недостатке памяти операция new будет возвращать **NULL.** Тогда проверку можно строить, проверяя, не равен ли значению **NULL** указатель, возвращенный new.

Приведенные ранее примеры относились к динамическому размещению в памяти одиночных объектов. Аналогичным образом можно размещать и массивы. Например, оператор

```
double *A = new double[100];
```

динамически размещает массив из 100 действительных чисел. К его элементам в дальнейшем можно обращаться как обычно, по индексу: **A[ind].** При использовании для создания массива операции new надо иметь в виду, что в момент создания его нельзя инициализировать, как это делается с одиночными объектами.

Можно создавать и многомерные массивы. Например, оператор

```
double *M = new double[100][100];
```

создает и динамически размещает в памяти двумерный массив. При размещении многомерных массивов надо иметь в виду, что первый размер можно задавать переменной, но остальные размеры задаются только константами. Например:

```
double *M = new double[n][100];
```

Динамически распределенную память надо освобождать, когда отпадает необходимость в размещенных в ней объектах. В противном случае получится неоправданная утечка памяти. Освобождение памяти осуществляется операцией **delete.** Она выполняет то же, что описанная ранее стандартная библиотечная функция **free.** Но использование **delete** предпочтительнее. Во всяком случае все, что размещается в памяти операцией new, должно удаляться операцией **delete.**

Операция может иметь следующие формы записи:

```
<::> delete <выражение>
<::> delete [ ] <выражение>
delete <имя_массива> [ ];
```

Например:

```
double *A = new double (5.1);
...
delete A;
```

или

```
double *A = new double[100];
...
delete [] A;
```

Операция **delete** освобождает память, но сама не задает указателю на эту память значения **NULL.** Поэтому желательно это делать программно, чтобы случайно в дальнейшем не воспользоваться указателем, который уже ни на что не указывает:

```
delete A;
A = NULL;
```

1.12 Исключения

1.12.1 Исключения и их стандартная обработка

При работе программы могут возникать различного рода ошибки: переполнение, деление на нуль, попытка открыть несуществующий файл и т.п. При возникновении таких исключительных ситуаций программа генерирует так называемое *исключение* и выполнение дальнейших вычислений в данном блоке прекращается. Исключение - - это объект специального вида, характеризующий возникшую в программе исключительную ситуацию. Он может также содержать в виде параметров некоторую уточняющую информацию. Особенностью исключений является то, что это сугубо временные объекты. Как только они обработаны каким-то обработчиком, они разрушаются.

Если исключение не перехвачено нигде в программе (как это делать — будет рассказано в последующих разделах), то оно обрабатывается методом **Application->HandleException.** Он обеспечивает стандартную реакцию программы на большинство исключений — выдачу пользователю краткой информации в окне сообщений и уничтожение экземпляра исключения. На рис. 1.3 приведены примеры таких стандартных сообщений для случаев целочисленного деления на нуль и попытки преобразовать функцией **StrToInt** строку "1.5" в целое число.

Если вы работаете в среде разработки C++Builder и отлаживаете свою программу, то при исключениях, помимо указанных на рис. 1.3 сообщений, могут появляться сообщения отладчика C++Builder, которые могут мешать вашей работе. Если хотите, то можете отключить появление этих сообщений. О работе со средой C++Builder см. в [1].

Если не принять соответствующих мер, то к неприятностям прекращения вычислений могут добавиться еще неприятности, связанные с так называемой *утеч-*

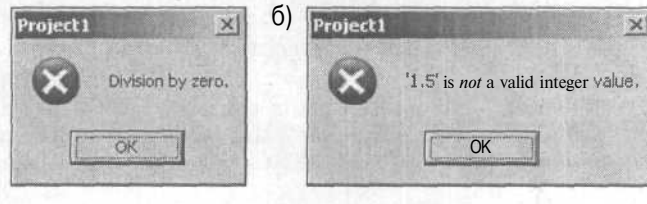


Рис. 1.3 Примеры стандартных сообщений об ошибках деления на нуль (а) и преобразования (б)

кой ресурсов. Под этим подразумеваются потери динамически распределяемой памяти, незакрытые файлы, не уничтоженные временные файлы на диске и прочий «мусор». Например, пусть вы выполняете некоторую программу, в которой имеются следующие операторы:

```
FILE *fp;
int size;
char *str;
...
fp = fopen("a.tmp", "w");
fprintf(fp,"файл a.tmp");
str = (char *) malloc (size);
<операторы, в которых может обнаружиться исключительная ситуация>

remove("a.tmp");
free(str);
```

Вы открываете временный файл (см. разд. 2.10.2) с именем **a.tmp,** чтобы хранить в нем какие-то промежуточные данные вычислений. В конце работы вы намерены уничтожить его процедурой **remove.** Вы динамически выделяете (см. разд. 1.11) некоторую память процедурой **malloc,** намереваясь освободить ее, когда она вам больше не будет нужна, процедурой **free.** Но если в промежуточных операторах возникнет исключение, то вычисления прервутся и процедуры **remove и free** не будут выполнены. В результате память, выделенная процедурой **malloc,** останется *недоступной,* а на диске сохранится временный и уже ненужный файл **a.tmp.**

Помимо указанного, стандартная обработка исключений программой имеет еще один недостаток — пользователь остается в полном недоумении, что же ему дальше делать? И не только не очень квалифицированный пользователь, которого приведенные на рис. 1.3 сообщения на английском языке могут повергнуть в шок. Даже опытному человеку невозможно порой догадаться, что же в вашей программе делится на нуль и как этого можно избежать. Наверное, каждый попадал в подобные ситуации, даже применяя профессионально сделанные программы, включая Windows.

Конечно, программист всегда должен принять все мыслимые меры, чтобы ни при каких ошибках пользователя и ни при каких сочетаниях данных приложение не заканчивалось бы аварийно. Но если все-таки аварийное завершение происходит, необходима полная зачистка «мусора» — удаление временных файлов, освобождение памяти, разрыв связей с базами данных и т.д.

1.12.2 Способы защиты кодов зачистки — блоки try ...__finally и функции exit

Рассмотрим способы защиты кодов зачистки «мусора». Первый из них — использование блока **try ...___finally.** Блок, содержащий совокупность операторов, способных привести к исключению, можно оформить следующим образом:

```
try
{
   // операторы, способные привести к исключению
}
__finally
(
   // операторы, выполняемые в любом случае
}
```

В этом случае операторы в разделе___**finally** будут выполняться всегда, независимо от того, было или не было исключение. Если было исключение, то после выполнения этих операторов вычисления, как и ранее, прерываются и возникает

сообщение об исключении; в противном случае управление передается операторам, следующим за разделом __finally.

В качестве примера рассмотрим приведенный ранее фрагмент кода с временным файлом и динамическим распределением памяти, оформленный следующим образом:

```
FILE *fp;
int size;
char *str;
...
try
{
 fp = fopen("a.tmp", "w");
 fprintf(fp,"Файл a.tmp");
 str = (char *) malloc(size);
// операторы, в которых может обнаружиться
// исключительная ситуация
)
__finally
{
 remove("a.tmp");
 free(str);
}
```

В этом случае процедуры **remove и free** будут выполнены независимо от того, сгенерировано ли исключение в операторах блока try, или все вычисления в них закончились благополучно. Таким образом проблема зачистки «мусора» снимается — память в любом случае будет освобождена, а временный файл будет удален. Причем, это достигается ничтожным дополнительным кодом по сравнению с глобальной предварительной проверкой всех операций.

К сожалению, остаются другие из рассмотренных проблем: необходимость принять какие-то меры для дальнейшей нормальной работы программы при генерации исключения, а также необходимость уведомить пользователя о желательных действиях с его стороны (сообщения типа приведенных на рис. 1.3 в этом случае отображаются на экране, но они мало информативны для пользователя). Решить эти проблемы в данном случае невозможно, поскольку при выполнении операторов раздела **__finally** программа не знает, произошло ли исключение, и если произошло, то какое именно. Проверки наличия исключения с помощью функций **ExceptAddr и ExceptObject,** специально предназначенных для этого, внутри раздела **__finally** ни к чему не приводят, так как исключение генерируется после выполнения этих операторов.

Рассмотренные выше меры направлены на защиту кода зачистки в блоке. Однако не все можно сделать на уровне блока. Поэтому полезно предусмотреть зачистку при завершении приложения.

Один из способов завершения приложения — вызов функции **exit:**

```
#include <stdlib.h>
void exit(int status);
```

Параметр **status** определяет код завершения. Обычно 0 соответствует нормальному завершению, а значение, отличное от нуля — аварийному при наличии ошибки выполнения. Можно, но не обязательно, использовать для задания значения status предопределенные константы: **EXIT_FAILURE** — аварийное завершение, **EXIT_SUCCESS** — нормальное.

Например:

```
exit(0);              // нормальное завершение
exit(EXIT_SUCCESS);   // нормальное завершение
exit(1);              // аварийное завершение
exit(EXIT_SUCCESS);   // аварийное завершение
```

При завершении приложения с помощью exit перед прекращением работы закрываются все открытые файлы и очищаются все буфера вывода (печатается находящийся в них текст). Эти операции совершаются по умолчанию. Если же вам надо произвести еще какие-то действия (например, уничтожить временные файлы на диске), то вы можете зарегистрировать одну или несколько собственных функций, которые всегда автоматически будут выполняться перед действиями по умолчанию.

Регистрируются собственные функции завершения с помощью функции **atexit:**

```
#include <stdlib.h>
int atexit(void (_USERENTRY * func)(void));
```

Здесь **func** — имя регистрируемой функции. Можно выполнить несколько вызовов **atexit,** зарегистрировав таким образом несколько функций завершения. При завершении приложения выполняться эти функции будут в обратной последовательности: сначала — последняя из зарегистрированных, а в конце — зарегистрированная первой.

Например, следующий код определяет две функции завершения — **myexit1 и myexit2:**

```
void myexit1 (void)
{
// операторы зачистки
}
void myexit2 (void)
{
// операторы зачистки
}
```

Эти функции могут не объявляться в заголовочном файле, а просто включаться в текст модуля.

Следующие операторы регистрируют эти функции:

```
atexit(myexit1);
atexit(myexit2);
```

Они могут быть включены, например, в обработчик события **OnCreate** главной формы приложения. Тогда при выполнении в любой точке программы вызова функции **exit** выполнятся операторы функции **myexit2,** затем операторы функции **myexitl,** затем выполнится зачистка по умолчанию (закрытие файлов и буферов), после чего произойдет завершение приложения.

Приведем пример функции **myexitl,** удаляющей в рабочем каталоге все временные файлы с расширением .tmp (использованные в примере функции **findfirst, findnext и remove см. в гл. 4):**

```
void myexit1 (void)
{
 struct ffblk ffblk;
 int D;
 D = findfirst("*.tmp", &ffblk, 0);
 while (!D)
 {
  remove(ffblk.ff_name);
  D = findnext(&ffblk);
 }
}
```

Возможность ввести в процесс собственные функции зачистки делает завершение приложения вызовом **exit** «мягким» по сравнению с некоторыми другими способами.

Если приложение завершается закрытием главной формы методом **Close,** то коды зачистки можно вставить в обработчики событий, происходящих при выполнении этого метода. Таким образом, это тоже «мягкий» способ завершения приложения. Причем, он имеет дополнительные преимущества, так как позволяет проанализировать ситуацию и в зависимости от каких-то условий завершить приложение, или не завершать его.

1.12.3 Иерархия классов исключений VCL

Для дальнейшего рассмотрения работы с исключениями надо представлять, хотя бы в первом приближении, иерархию классов объектов исключений и свойства этих объектов. Ниже приведена таблица иерархии большинства предопределенных в C++Builder классов исключений с краткими пояснениями. Создаваемые пользователем новые классы должны быть производными от одного из классов этой иерархии. Следует отметить, что помимо исключений, наследующих базовому классу **Exception** и используемых в объектах (компонентах) библиотеки VCL, имеются еще исключения, наследующие классу **exception** — базовому классу исключений стандартной библиотеки C++. Эти исключения рассмотрены в разд. 1.12.5.

Exception	Базовый класс исключений VCL		
	EAbort	«Молчаливое» исключение, предназначенное для намеренного прерывания вычислений и быстрого выхода из глубоко вложенных процедур и функций	
	EAbstractError	Попытка вызвать абстрактный метод	
	EArrayError	Ошибка манипулирования с потомками класса **TBaseArray:** использование ошибочного индекса элемента массива, добавление слишком большого числа элементов в массив фиксированной длины, попытка вставки элемента в отсортированный массив	
	EAssertion-Failed	Ложное выражение, проверяемое процедурой Assert в объектах VCL или в модулях Pascal	
	EBitsError	Ошибка доступа к массиву булевых величин TBits	
	ECacheError	Ошибка построения кэша в кубе решений	
	ECommon-CalendarError	Ошибки ввода в компоненты, наследующие классу **TCommonCalendar**	
		EDateTimeError	Ошибка ввода даты или времени в компоненте **TDateTimePicker**
	EComponent-Error	Ошибка регистрации или переименования компонентов	
	EConvertError	Ошибка преобразования строк или объектов (в частности, в функциях StrToInt, **StrToFloat, StrToDate**)	
	EDatabaseError	Ошибка работы с базами данных	
		EDBClient	Ошибка в наборе данных клиента. Свойство **ErrorCode** содержит код ошибки, возвращаемый BDE

			EReconcile-Error	Ошибка обновления данных компонента **TClientDataset;** свойство **Context** содержит информацию в виде сообщения об ошибке, а свойство **ErrorCode** содержит код ошибки, возвращаемый BDE
		EDBEngineError	Ошибка в BDE. Свойство **Errors** содержит информацию об ошибке — объект типа **TDBErrors. Свойство ErrorCount** хранит число ошибок	
		ENoResultSet	Генерируется компонентом **TQuery** при попытке открыть запрос без оператора SELECT	
		EUpdateError	Ошибка при обновлении в **TProvider**	
	EDBEditError	Ошибка при попытке приложения использовать данные, не соответствующие заданной маске для поля		
	EDimension-MapError	Ошибка формата данных в кубе решений		
	EDimIndex-Error	Ошибочный индекс в задании размерности в кубе решений		
	EExternal	Класс, перехватывающий исключения Windows		
		EAccess-**Violation**	Ошибочный доступ к памяти; генерируется при попытке разыменования нулевого указателя **NULL,** попытке записи в кодовую страницу, попытке доступа к адресу вне памяти, распределенной приложению	
		EControlC	Нажатие пользователем клавиш Ctrl+C при выполнении консольного приложения. При обработке этого исключения можно выдать запрос пользователю, действительно ли он хочет прервать работу, и предпринять действия в зависимости от его ответа	
		EIntError	Базовый класс исключений целочисленных математических операций	
			EDivByZero	Попытка целочисленного деления на нуль

			ERange-Error	Целочисленное значение или индекс вне допустимого диапазона; используется только в Object Pascal
			EInt-Overflow	Переполнение при операции с целыми числами
		EMathError	Базовый класс исключений операций с плавающей запятой; всегда генерируются только потомки этого исключения; обработка исключения EMathError может использоваться для перехвата всех исключений операций с плавающей запятой	
			EInvalid-Argument	Недопустимое значение параметра при обращении к математической функции
			EInvalidOp	Неопределенная операция с плавающей запятой: процессор наталкивается на неопределенную инструкцию, ошибочную операцию или переполняется стек процессора с плавающей запятой
			EOverflow	Переполнение регистра при операциях с плавающей запятой
			EUnderflow	Потеря значащих разрядов при выполнении операции с плавающей запятой
			EZeroDivide	Деление на нуль числа с плавающей запятой
		EPrivilege	Попытка приложения выполнить инструкцию процессора, которая недоступна для текущего уровня привилегий	
		EStackOverflow	Переполнение стека	

	EExternal- Exception	Неизвестный код исключения	
	EHeapException	Ошибка динамического распределения памяти	
		EInvalidPointer	Ошибочная операция с указателем, например, попытка дважды освободить один и тот же блок памяти
		EOutOfMemory	Неудачная попытка динамически выделить память; может генерироваться процедурой OutOfMeшогуЕггог
		EOutOf- Resources	Генерируется при попытке приложения создать дескриптор Windows, когда Windows не имеет места для размещения дополнительных дескрипторов; возможно и при выделении других ресурсов Windows
	EInOutError	Ошибка ввода-вывода из файла; исключение генерируется, если включена опция I/O checking на странице Pascal окна опций проекта; информация о конкретном виде ошибки содержится в локальной переменной ErrorCode	
	EIntfCastError	Ошибочное применение операции преобразования типов интерфейса	
	EInvalidCast	Ошибка преобразования типа объекта	
	EInvalid- Graphic	Нераспознаваемый графический файл	
	EInvalid- Graphic- Operation	Ошибочная операция с графикой, например, попытка изменить размер пиктограммы или копирование пиктограммы в буфер Clipboard	
	EInvalidGrid- Operation	Ошибочная операция с таблицей	
	EInvalid- Operation	Ошибочная операция с компонентом; генерируется при попытке выполнить операцию, которая требует обработчика окна, над компонентом, не имеющим родителя (свойство Parent = NULL). Это исключение также генерируется при выполнении операций перетаскивания над формой (например, при попытке выполнить операцию **Form1::BeginDrag**).	

	EListError	Ошибка работы с объектом типа списка TStringList и **TStrings:** попытке сослаться на элемент с индексом вне допустимых пределов, попытке добавления дубликата строки в объект TStringList, в котором значение свойства Duplicates равно dupErrог, попытке вставить элемент в сортированный список, так как это может нарушить правильную последовательность элементов		
	ELowCapacity-Error	Попытка выделить памяти больше, чем доступно кубу решений; надо или увеличить значение Capacity, или уменьшить размерность куба		
	EMCIDevice-Error	Ошибка доступа к устройствам мультимедиа через драйвер Media Control Interface (MCI)		
	EMenuError	Ошибка, связанная с элементами меню		
	EOleCtrlError	Генерируется при невозможности связать приложение с компонентом ActiveX		
	EOleError	Низкоуровневая ошибка OLE; C++Builder проверяет это исключение, но не генерирует его		
		EOleSysError	Ошибка OLE, специфическая для интерфейса OLE IDispatch; свойство **ErrorCode** содержит номер ошибки.	
			EOle-Exception	Ошибка OLE, связанная с методом или свойством
	EOutlineError	Ошибка при работе с компонентом Outline		
		EOutOf-Resources	Генерируется при попытке приложения создать дескриптор Windows, когда Windows не имеет места для размещения дополнительных дескрипторов; возможно и при выделении других ресурсов Windows	
	EPackageError	Исключение времени проектирования, генерируемое при загрузке или использовании пакета		
	EParserError	Ошибка преобразования текста описания формы в двоичное представление, происходящая обычно из-за синтаксической ошибки исходного текста (часто из-за исправления текста вручную)		
	EPrinter	Ошибка печати; например, приложение пытается использовать принтер, которого нет, или задание по какой-то причине не может быть послано на принтер		
	EPropReadOnly	Попытка записать с помощью автоматизации OLE значение свойства, которое предназначено только для чтения		
	EPropWrite-Only	Попытка прочитать с помощью автоматизации OLE значение свойства, которое предназначено только для записи		

	EPropertyError	Ошибка при задании значения свойства	
	ERegistry-Exception	Ошибка при обращении к реестру	
	EResNotFound	Ошибка при загрузке файла ресурсов **.DFM** или **.RES** в процессе проектирования.	
	EStreamError	Базовый класс исключений ошибок потоков	
		EFCreateError	Ошибка создания файла; например, пользователь указал недопустимое имя файла или указанный файл уже существует и не может быть перезаписан, так как пользователь не обладает соответствующим уровнем доступа
		EFOpenError	Ошибка открытия файла
		EFilerError	Базовый класс исключений файловых потоков
			EReadError — Невозможно прочесть заданное число байтов
			EWrite-Error — Невозможно записать заданное число байтов
			EClassNot-Found — Компонент не связан с приложением
			EInvalid-Image — Невозможно прочесть файл ресурсов
	EStringList-Error	Ошибочный доступ к окну списка с неверным индексом	
	EThread	Конфликт в многопоточном приложении (например, вызов метода Synchronize объекта **Tthread** до успешного завершения его предыдущего вызова)	
	ETreeView-Error	Ошибка индекса при работе с компонентом TTree-View	
	EUnsupported-TypeError	Ошибка выбора типа поля в качестве размерности куба решений	
	EVariantError	Ошибка, связанная с типом данных Variant	
	EWin32Error	Ошибка Windows, генерируется процедурой **RaiseLastWin32Error**, если Windows возвращает ошибку	

1.12.4 Базовый класс исключений VCL Exception

Все предопределенные в C++Builder классы исключений, как видно из их иерархии, приведенной в разд. 1.12.3, являются прямыми или косвенными наследниками класса Exception, объявленного в модуле SysUtils и наследующего непосредственно TObject.

1.12.4.1 Свойства исключений

В классе Exception объявлено два свойства:

Свойство	Тип	Описание
Help-Context	**int**	Целый идентификатор экрана контекстно-зависимой справки. Этот экран справки отображается, если пользователь, находясь в окне с сообщением об ошибке, нажимает клавишу F1. По умолчанию значение равно 0
Message	**System::AnsiString**	Строка сообщения, которая в дальнейшем при обработке исключения системным обработчиком отображается в окне сообщений; устанавливается конструктором с умолчанием

Свойство **Message** имеет значение по умолчанию, которое присваивается при автоматической генерации исключения. При преднамеренной генерации исключений их конструкторы, описанные в следующем разделе, могут задавать значение свойства **Message** в виде переменной типа **string** или литеральной константы.

Свойство **HelpContext** хранит целый идентификатор экрана контекстно-зависимой справки. Этот экран справки отображается, если пользователь, находясь в окне с сообщением об ошибке, нажимает клавишу F1.

По умолчанию значение свойства **HelpContext** равно 0. Это значение может изменяться некоторыми конструкторами (см. следующий раздел). Например, оператор

```
throw Exception("Не хватает исходных данных", 4);
```

генерирует исключение со значением свойства **Message,** равным тексту "Не хватает исходных данных", и значением свойства **HelpContext,** равным 4. При получении сообщения об этом исключении пользователь сможет нажать клавишу F1 и получить пояснения, что ему делать в этом случае.

Конечно, чтобы это работало, надо создать соответствующий файл справки и связать его с приложением, установив соответствующую опцию Help file (файл справки) в окне Project Options (опции проекта) на странице Application (приложение).

1.12.4.2 Конструкторы исключений

Класс **Exception** наследует все функции своего базового класса **TObject,** в частности, полезную для идентификации неизвестного исключения функцию **ClassName.**

Кроме того, в интерфейсе класса **Exception** описано 8 конструкторов, наследуемых всеми исключениями:

Конструктор	Описание
Exception(const **System::AnsiString** Msg)	Конструктор, передает строку сообщения Msg свойству Message
Exception(const System::AnsiString Msg, const System::TVarRec * Args, const int Args_Size)	Конструктор формирует строку свойства Message, исходя из строки описания формата Msg и массива аргументов **Args** размером **Args_Size**

Конструктор	Описание
Exception(int Ident)	Конструктор задает строку свойства **Message** идентификатором **Ident** строки сообщения в ресурсах проекта
Exception(int Ident, const System::TVarRec * Args, const int Args_Size)	Конструктор задает строку свойства Message идентификатором **Ident** строки описания формата в ресурсах проекта и массивом аргументов **Args**
Exception(const System::AnsiString Msg, int AHelpContext)	Конструктор передает строку сообщения Msg свойству **Message;** передает свойству **HelpContext** идентификатор **HelpContext** экрана контекстно-зависимой справки по этому исключению
Exception(const System::AnsiString Msg, const System::TVarRec * Args, const int Args_Size, int AHelpContext)	Конструктор формирует строку свойства **Message,** исходя из строки описания формата Msg и массива аргументов **Args;** передает свойству **HelpContext** идентификатор **HelpContext** экрана контекстно-зависимой справки по этому исключению
Exception(int Ident, int AHelpContext)	Конструктор задает строку свойства **Message** идентификатором **Ident** строки сообщения в ресурсах проекта; передает свойству **HelpContext** идентификатор **HelpContext** экрана контекстно-зависимой справки по этому исключению
Exception(int Ident, const System::TVarRec * Args, const int Args_Size, int AHelpContext)	Конструктор формирует строку свойства **Message** исходя из строки описания формата в ресурсах проекта, указываемой **идентификатором Ident,** и массива аргументов **Args;** передает свойству **HelpContext** идентификатор **HelpContext** экрана контекстно-зависимой справки по этому исключению

Рассмотрим примеры использования различных конструкторов:

```
throw Exception("Не хватает исходных данных");

throw Exception(Format("Задано %d параметров из %d",
                OPENARRAY(TVarRec, (N1, N2))));
```

Последний пример использует функцию **Format** (см. гл. 4) для форматированного вывода информации о значениях переменных N1 и N2. При этом для передачи в конструктор массива используется макрос **OPENARRAY** (о передаче в функции открытых массивов см. в гл. 2 в разд. 2.11.3). В результате, например, при значениях переменных N1 = 5 и N2 = 7 будет сгенерировано исключение, в диалоговом окне которого появится текст: "Задано 5 параметров из 7".

Следующий пример:

```
throw Exception("Задано %d параметров из %d",
                OPENARRAY(TVarRec,(N1, N2)));
```

Этот пример аналогичен предыдущему, но использует конструктор с непосредственным заданием строки форматирования в качестве первого параметра. Поэтому запись получается несколько короче, чем в предыдущем примере.

Следующий пример генерирует исключение с указанием темы контекстно-зависимой справки:

```
throw Exception("Не хватает исходных данных", 4);
```

Этот оператор сгенерирует исключение с тем же текстом, что и в одном из приведенных выше примеров, но если в диалоговом окне с сообщением об этом исключении пользователь нажмет клавишу F1, ему будет предъявлена контекстная справка с идентификатором 4.

Пример применения конструктора, использующего строку ресурсов:

```
throw Exception(65369);
```

Этот оператор передает в свойство Message строку с номером 65369 из файла ресурсов. Оператор

```
raise EMy.CreateResFmt(65369, OPENARRAY(TVarRec,(N1, N2)));
```

берет из файла ресурсов строку с номером 65369 как строку описания формата и передает в свойство Message сформатированные с ее помощью значения переменных N1 и N2.

1.12.5 Обработка исключений в блоках try ... catch

1.12.5.1 Синтаксис блоков try ... catch

Наиболее кардинальный путь борьбы с исключениями — отлавливание и обработка их с помощью блоков **try ... catch.** Синтаксис этих блоков следующий:

```
try
{
  Исполняемый код
}
catch ( TypeToCatch )
{
   Код, исполняемый в случае ошибки
}
```

Операторы блока **catch** представляют собой обработчик исключения. Параметр **TypeToCatch** может быть или одним из целых типов (**int, char** и т.п.), или ссылкой на класс исключения, или многоточием, что означает обработку любых исключений. Смысл параметров целого типа будет рассмотрен ниже в разд. 1.12.6.1. А пока остановимся на случае, когда параметр является ссылкой на класс исключений.

Операторы обработчика выполняются только в случае генерации в операторах блока **try** исключения типа, указанного в заголовке catch. После блока **try** может следовать несколько блоков **catch** для разных типов исключений. Таким образом, в обработчиках **catch** вы можете предпринять какие-то действия: известить пользователя о возникшей проблеме и подсказать ему пути ее решения, принять какие-то меры к исправлению ошибки (например, при переполнении заслать в результат очень большое число соответствующего знака) и т.д. Наиболее ценным является то, что вы можете определить тип сгенерированного исключения и дифференцированно реагировать на различные исключительные ситуации. Причем перехват исключения блоком **catch** приводит к тому, что это исключение далее не обрабатывается стандартным образом, т.е. пользователю не предъявляется окно с непонятными ему английскими текстами.

Приведем пример обработки исключений. Пусть в вашем приложении имеется два окна редактирования **Edit1 и Edit2,** в которых пользователь вводит действительные числа типа **float.** Приложение должно разделить их одно на другое. При этом возможен ряд ошибок: пользователь может ввести в окно символы, не преобразуемые в целое число, может ввести слишком большое число, может ввести вместо делителя нуль, результат деления может быть слишком большим для типа **float.** Следующий код отлавливает все эти ошибки:

```
float A;
try
{
 A = StrToFloat(Edit1->Text) / StrToFloat(Edit2->Text);
}
catch(EConvertError&)
{
 Application->MessageBox("Вы ввели ошибочное число",
                         "Повторите ввод",MB_OK);
}
catch(EZeroDivide&)
{
 Application->MessageBox("Вы ввели нуль",
                         "Повторите ввод",MB_OK);
}
catch(EOverflow&)
{
 Application->MessageBox("Переполнение",
                         "Ошибка вычислений",MB_OK);
 if (StrToFloat(Edit1->Text) * StrToFloat(Edit2->Text) >= 0)
      A = 3.4E38;
 else A = -3.4E38;
}
```

Если пользователь ввел неверное число (например, по ошибке нажал не цифру, а какой-то буквенный символ), то при выполнении функции **StrToFloat** возникнет исключение класса **EConvertError.** Соответствующий обработчик исключения сообщит пользователю о сделанной ошибке и посоветует повторить ввод. Аналогичная реакция последует на ввод пользователем в качестве делителя нуля (класс исключения **EZeroDivide).** Если возникает переполнение, то соответствующий блок **catch** перехватывает исключение, сообщает о нем пользователю и исправляет ошибку: заносит в результат максимально возможное значение соответствующего знака.

Поскольку исключения образуют иерархию, рассмотренную в разд. 1.12.3, можно обрабатывать сразу некоторую совокупность исключений, производных от одного базового исключения. Для этого надо в заголовке блока **catch** указать имя этого базового исключения. Например, исключения **EZeroDivide** (целочисленное деление на нуль), **EOverflow** (переполнение при целочисленных операциях), **EInvalidArgument** (выход числа за допустимый диапазон) и некоторые другие являются производными от класса исключений **EMathError.** Поэтому все их можно отлавливать с помощью одного блока **catch,** например, такого:

```
catch(EMathError&)
{
 Application->MessageBox("Ошибка вычислений",
                         "Повторите ввод", MB_OK);
}
```

Правда, в этом случае не конкретизируется причина прерывания исключений. Однако такая конкретизация возможна, если воспользоваться свойствами исключений. Все исключения имеют свойство **Message,** которое представляет собой строку, отображаемую пользователю при стандартной обработке исключений.

Чтобы воспользоваться свойствами исключений, надо в заголовке блока **catch** не только указать тип исключения, но и создать временный указатель на объект этого типа. Тогда через имя этого объекта вы получаете доступ к его свойствам. Ниже приведен пример использования свойств исключений при перехвате исключений, наследующих классу **EMathError**:

```
catch(EMathError& E)
{
AnsiString S = "Ошибка вычислений : ";
if(E.Message == "EZeroDivide") S += "деление на нуль";
if(E.Message == "EOverflow") S += "переполнение";
if(E.Message == "EInvalidArgument") S += "недопустимое число";
Application->MessageBox(S.c_str(), "Повторите ввод", MB_OK);
}
```

Вводимое в этом операторе имя ссылки на исключение E носит сугубо локальный характер и вводится только для того, чтобы можно было сослаться на свойство **Message** по имени объекта исключения.

Как уже говорилось выше, если в заголовке блока **catch** указано многоточие, то этот блок перехватит любые исключения:

```
catch(...)
(
ShowMessage("Призошла ошибка.");
}
```

Блок **catch(...)** может сочетаться и с другими блоками **catch,** но в этом случае он должен, конечно, располагаться последним. Поскольку этот блок перехватит все исключения, то все блоки, следующие за ним, окажутся недоступными. C++Builder следит за этим. Если блок **catch(...)** оказался не последним, вам будет выдано компилятором сообщение об ошибке с текстом: "The handler must be last" ("Обработчик должен быть последним").

Следует отметить некоторую опасность применения блока **catch(...)**. Перехват всех исключений способен замаскировать какие-то непредвиденные ошибки в программе, что затруднит их поиск и снизит надежность работы.

1.12.5.2 Последовательность обработки исключений, обработка на уровне приложения

Блоки **try...catch** могут быть вложенными явным или неявным образом. Примером неявной вложенности является блок **try...catch,** в котором среди операторов раздела **try** имеются вызовы функций, которые имеют свои собственные блоки **try...catch.** Рассмотрим последовательность обработки исключений в этих случаях. При генерации исключения сначала ищется соответствующий ему обработчик в том блоке **try...catch,** в котором создалась исключительная ситуация. Если соответствующий обработчик не найден, поиск ведется в обрамляющем блоке **try...catch** (при наличии явным образом вложенных блоков) и т.д. Если в данной функции обработчик не найден или вообще в ней отсутствуют блоки **try...catch,** то поиск переходит на следующий уровень — в блок, из которого была вызвана данная функция. Этот поиск продолжается по всем уровням. И только если он закончился безрезультатно, выполняется стандартная обработка исключения, заключающаяся, как уже было сказано, в выдаче пользователю сообщения о типе исключения.

Как только блок **catch,** соответствующий данному исключению, найден и выполнен, объект исключения разрушается и управление передается оператору, следующему за соответствующим блоком **try...catch.**

Возможен также вариант, когда в самом обработчике исключения в процессе обработки возникла исключительная ситуация. В этом случае обработка прерывается, прежнее исключение разрушается и генерируется новое исключение. Его об-

работчик ищется в блоке **try...catch,** внешнем по отношению к тому, в котором возникло новое исключение.

Если исключение не перехвачено ни одним обработчиком в функциях, вы можете обработать его на уровне приложения. Для этого предусмотрены события **OnException** компонента **Application** — самого приложения. Обработчик этих событий можно ввести в ваше приложение следующим образом. Пусть вы решили назвать этот обработчик **MyException.** Тогда в заголовочный файл приложения надо добавить его объявление:

```
void__fastcall MyException(TObject *Sender, Exception *E);
```

В файл вашего модуля надо внести реализацию обработчика:

```
void__fastcall TForm1::MyException(TObject *Sender, Exception *E)
{
// операторы обработки
}
```

Осталось указать приложению на вашу функцию **MyException** как на обработчик события **OnException.** Вы можете это сделать, включив, например, в обработку события формы **OnCreate** оператор:

```
Application->OnException = MyException;
```

Ваш обработчик не перехваченных ранее исключений готов. Осталось только наполнить его операторами, сообщающими пользователю о возникших неполадках и обеспечивающими дальнейшую работу программы. К вашей функции **MyException** приложение будет обращаться, если было сгенерировано исключение и ни один блок **catch** его не перехватил. В функцию передается указатель E на объект класса **Exception.** Этот объект является сгенерированным исключением, а класс Exception — базовый класс всех исключений.

Простейшая обработка исключения могла бы производиться функцией **ShowException,** обеспечивающей отображение информации об исключении:

```
Application->ShowException(E);
```

Примеры сообщений, выдаваемых этой функцией, были приведены ранее на рис. 1.3. В заголовке окна пишется имя приложения, а текст содержит описание причины генерации исключения. Основным недостатком функции являются сообщения на английском языке, что вряд ли -порадует пользователей вашего приложения. Поэтому лучше сделать собственные сообщения. При этом для определения истинного класса сгенерированного исключения можно воспользоваться методом **ClassName.** Тогда обработчик события **OnException** может иметь, например, следующий вид:

```
void__fastcall TForm1::MyException(TObject *Sender,
                                   Exception *E)
{
 AnsiString S = "Ошибка вычислений : ";
 if ((String(E->ClassName()) == "EZeroDivide")
     || (String(E->ClassName()) == "EDivByZero"))
    S += "деление на нуль";
 if (String(E->ClassName()) == "EOverflow")
    S += "переполнение";
 if(String(E->ClassName()) == "EInvalidArgument")
    S += "недопустимое число";
 if(String(E->ClassName()) == "EConvertError")
    S += "ввели недопустимое число";
 Application->MessageBox(S.c_str(),"Повторите ввод",MB_OK);
}
```

На рис. 1.4 приведены примеры сообщений, выдаваемых эти обработчиком. Вероятно, пользователям более понравятся сообщения рис. 1.4, чем сообщения рис. 1.3.

Рис. 1.4 Сообщения, выдаваемые вашим обработчиком при делении на нуль (а) и при неверной записи вводимого числа (б)

1.12.6 Преднамеренная генерация исключений

1.12.6.1 Оператор throw

В ряде случаев возникает потребность сгенерировать исключение искусственно. Например, вы обработали какое-то исключение, но хотите, чтобы его обработка была завершена обработчиком внешнего по отношению к данному блока **try...catch**. В этом случае вам надо повторно сгенерировать исключение того же типа, что и прежнее, поскольку прежнее разрушено данным обработчиком.

Повторная генерация исключения осуществляется ключевым словом **throw.** Общая схема такой двухэтапной обработки исключений может иметь вид:

```
try
{
// операторы внешнего блока
try // начало внутреннего блока
  {
// операторы внутреннего блока,
    // способные привести к генерации исключения
}
catch (тип исключения &)
  {
// обработка исключения, сгенерированного во внутреннем блоке
    throw;  // повторная генерация того же исключения
  }
// операторы внешнего блока;
  // при генерации исключения не выполняются
} // завершение внешнего блока try
catch (тип исключения &)
{
// обработка исключений и внутреннего, и внешнего блоков
}
```

При такой организации программы Исключения, сгенерированные во внутреннем блоке, обрабатываются в два этапа: сначала обработчиком внутреннего блока, а затем обработчиком внешнего блока. При этом операторы внешнего блока, следующие за внутренним, при генерации исключения во внутреннем блоке выполняться не будут. Исключения, сгенерированные во внешнем блоке, будут обрабатываться только обработчиками этого внешнего блока.

С помощью ключевого слова **throw** можно сгенерировать не только повторное исключение, но и исключение любого типа в любом месте программы. Такая необходимость, в частности, возникает, когда пользователь что-то не так сделал и не имеет смысла продолжать выполнение приложения. Например, пользователь должен был задать какую-то информацию в окнах редактирования, но забыл это сделать. В этом случае прежде, чем продолжать работу, надо указать пользователю на его ошибку. Это можно сделать, сгенерировав соответствующее исключение.

Генерация нестандартного исключения производится ключевым словом **throw,** после которого указывается генерируемый объект любого типа. Это исключение может в дальнейшем перехватываться блоком **catch,** в заголовке которого указан то же тип, что у сгенерированного объекта. Например, вы можете написать оператор, который проверяет, задана ли информация в окне редактирования **Edit1,** и, если не задана, то генерируется исключение:

```
if(Edit1->Text == "") throw "Не задана требуемая информация";
```

В данном случае объект генерируемого исключения имеет тип **char *.** Подобных операторов с различными текстами в разных местах кода может быть много. И все они могут быть перехвачены, например, следующим блоком **catch:**

```
catch(char * s)
{
 Application->MessageBox(s, "Ошибка ввода", MB_ICONHAND|MB_OK);
}
```

Заголовок этого блока обеспечивает перехват любых исключений типа **char** * и отображение их текстов в диалоговом окне, пример которого показан на рис. 1.5.

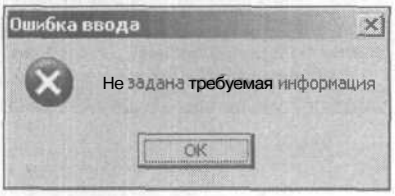


Рис. 1.5
Окно, отображающее текст перехваченного исключения типа char *

Вы можете в качестве параметра **throw** задавать целые числа, отображающие некие номера ошибок. Например:

```
if (...)throw 1;
```

В данном случае генерируется объект исключения типа int. Поэтому подобные исключения могут быть перехвачены и обработаны блоком вида:

```
catch(int& i)
{
 switch (i)
 {
  case 1: ...
          break;
  case 2: ...
          break;
...
 }
}
```

Можно генерировать объекты исключений и более сложных типов, например, структуры с полями, которые анализируются в обработчике исключения. Пусть, например, пользователь перед занесением в базу данных новой записи, относящейся к некоторому объекту, должен задать некоторый минимум параметров (характеристик) этого объекта. В приведенном ниже коде создается структура st типа Pers и в ходе диалога с пользователем в нее заносится число заданных параметров (st.il) объекта. Тогда перед занесением в базу данных программа может сверить требуемое (st.il) и действительно заданное (st.i2) число параметров. Если параметров задано недостаточно, то генерируется исключение, объектом которого является структура st. Обработчик этого исключения в свою очередь может проанализировать все поля структуры и выдать пользователю соответствующее сообщение.

```
struct Pers
{
 int i1, i2;
} st = {0,5};

try
{
 if(...) st.i1++;
 ...
 if(st.i1 < st.i2) throw st;
}
catch (Pers&)
{
 Application->MessageBox(("Требуется параметров - " +
                IntToStr(st.i2) + "\nЗадано параметров — " +
                IntToStr (st. i1)) .c_str() ,
                "Не хватает информации",MB_ICONHAND | MB_OK);
}
```

Пример выдачи информации таким обработчиком исключения приведен на рис. 1.6.

Рис. 1.6
Пример выдачи сообщения об объекте исключения в виде структуры

1.12.6.2 Исключение EAbort и функция Abort

В C++Builder имеется исключение **EAbort,** несколько отличающееся от рассмотренных ранее. Генерация этого исключения, как и любых других, прерывает процесс вычисления. Но если приложение не отлавливает соответствующим блоком **catch** исключений этого класса, то они попадают в обработчик **TApplication::HandleException** и там, **в** отличие от других исключений, разрушаются без всяких сообщений. Таким образом, это «молчаливое» прерывание процесса вычисления, при котором не должно отображаться диалоговое окно с сообщением об ошибке.

Простейший путь генерации исключения **EAbort** — вызов функции **Abort.** Например:

```
if (...) Abort () ;
```

Только нельзя путать две похожие внешне функции: **Abort** - - генерация «молчаливого» исключения, и **abort** — аварийное завершение программы.

Обычное применение **EAbort** — прерывание вычислений при выполнении некоторого условия окончания или условия прерывания пользователем (например, при нажатии клавиши Esc или какого-то оговоренного сочетания клавиш). Функция **Abort** прерывает текущую процедуру и все вызвавшие ее процедуры, передавая управление на самый верх. Таким образом, это наиболее простой выход из глубоко вложенных процедур. Впрочем, можно при необходимости перехватить исключение на каком-то промежуточном уровне, предусмотрев на нем блок **try...catch** и вставив соответствующий оператор обработки:

```
catch(EAbort&)
{
 ...
}
```

1.12.7 Стандартные исключения C++

В предыдущих разделах была рассмотрена обработка исключений, используемая в C++Builder. Разработчики этой системы позаботились об автоматической генерации множества исключений, и пользователю остается только вовремя перехватывать и обрабатывать их. В стандартном C++ все обстоит несколько иначе.

При использовании исключений стандартного C++ учтите, что помимо подключения директивами **#include** соответствующих заголовочных файлов, надо обеспечивать пространство имен стандартной библиотеки (см. гл. 5), вставляя в ваш модуль оператор:

```
using namespace std;
```

Альтернативный подход — ссылка на соответствующее пространство имен непосредственно перед применением типа исключения. Например, **std::bad_alloc.**

Базовым классом исключений в C++ является класс **exception,** объявленный в модуле *stdexcept.* Он имеет, кроме, естественно, конструкторов и деструктора, одну виртуальную функцию-элемент: **what ().** Она возвращает текст сообщения, заданный в том или ином исключении.

Классу **exception** наследуют следующие классы исключений:

Исключение	Описание	Заголовочный файл
bad_alloc	Генерируется операцией new при отсутствии необходимого объема памяти (см. подробное описание в разд. 1.11)	<new>
bad_cast	Генерируется операцией dynamic_cast, примененной к ссылке, если операнд не принадлежит ожидаемому типу	<typeinfo>
bad_exception	Неожиданное исключение, которое может быть обработано особым образом (см. далее)	<exception>
bad_typeid	Генерируется операцией typeid при ее применении к нулевому указателю или ссылке	<typeinfo>
ios_base::failure	Генерируется при ошибочных операциях со стандартными потоками ввода и вывода	<ios>
logic_error	Базовый класс исключений логических ошибок в приложении	<exception>
runtime_error	Базовый класс исключений различных ошибок времени выполнения	<exception>

Первые пять из перечисленных исключений генерируются автоматически средствами языка при возникновении ошибок в процессе выполнения соответствующих операций. А два последних являются базовыми для ряда предопределенных в библиотеке исключений, а также основой для объявления своих собственных исключений. Ниже приведены определенные в заголовочном файле **<stdexcept>** наследники класса **logic_error:**

domain_error	Используется в функциях для сообщения о выходе значения за допустимые пределы
invalid_argument	Используется для сообщения об ошибочном значении аргумента функции
length error	Используется в функциях для сообщения о превышении какого-то размера, например, при занесении лишних символов в строку ограниченной длины
out_of_range	Используется в функциях для сообщения о неверном значении индекса массива и т.п.

Конечно, все эти определения достаточно условны, и пользуясь конкретными библиотечными функциями надо смотреть, какие исключения и в каких случаях они генерируют.

Класс **runtime_error** имеет следующих наследников, объявленных в заголовочном файле **<stdexcept>** и оповещающих о возникновении чисто вычислительных проблем:

range_error	Используется в функциях для сообщения о выходе значения за допустимые пределы
overflow_error	Используется для сообщения о переполнении
underflow_error	Используется в функциях для сообщения о потере порядка

В C++ предполагается, что программист при написании тех или иных функций, прежде всего, библиотечных, проверяет возможные ошибочные ситуации и генерирует в нужных случаях оператором throw соответствующие исключения. В конструкторы исключений передается строка текста, который в дальнейшем может быть прочитан в обработчике исключения функцией **what()**.

Пусть, например, вы пишете некоторую функцию F, в которой вам надо разделить 1 на некоторое число R. Тогда перед делением вы можете проверить, не равно ли R нулю, и если равно, то сгенерировать исключение, передав в него сообщение для пользователя. Аналогичным образом можно предусмотреть в функции другие случаи, требующие генерации исключений. А в том блоке, который вызывает эту функцию, можете установить универсальный перехватчик всех событий (перехватчик базового класса **exception)** и в нем проинформировать пользователя о возникших проблемах или сделать что-то более полезное. Все это может выглядеть так:

```
#include <stdexcept>
using namespace std;
...
double F(double R)
{
...
 if (R == 0)
  throw overflow_error("Переполнение из-за деления на нуль");
 else
  A = 1 / R;
 ...
}
...

try
{
 ...
```

```
// вызов функции F
 double B = F(B);
}
catch(exception & e)
{
// перехват исключения
 ShowMessage(e.what());
}
```

Для облегчения контроля того, какие исключения могут генерироваться в функции, можно в ее заголовок добавить *спецификацию исключений* — ключевое слово **throw,** после которого в скобках записывается список исключений, разделяемых запятыми. Например:

```
double F(double R) throw (overflow_error)
```

Если спецификации нет, как в приведенном ранее примере, то могут генерироваться любые исключения. Если список в спецификации пуст:

```
double F(double R) throw ()
```

то не ожидается никаких исключений.

При наличии спецификации, генерация в функции непредусмотренного исключения все-таки возможна. В этом случае вызывается функция **unexpected,** которая аварийно завершает приложение вызовом функции **terminate.** Например, если функция F имеет следующий вид:

```
double F(double R)  throw (overflow_error)
{
...
  throw overflow_error("Переполнение из-за деления на нуль");
...
  throw range_error("Ошибка диапазона");
...
}
```

то генерация исключения **overflow_error** может быть обработана во внешнем блоке **catch,** а генерация исключения **range_error** приведет к аварийному завершению программы.

В документации на C++ указывается, что аварийного завершения можно избежать, если включить в спецификацию исключение **bad_exception.** Тогда при появлении непредусмотренного исключения функция **terminate,** вместо аварийного завершения программы, будет генерировать исключение **bad_exception,** которое может быть перехвачено обычным образом. Но C++Builder 6 этой возможности не поддерживает. Тем не менее, аварийное завершение программы при генерации непредусмотренного исключения можно предотвратить. Имеется возможность указать собственную функцию **unexpected.** Для этого используется функция **set_unexpected,** объявленная следующим образом:

```
typedef void ( * unexpected_handler )();
unexpected_handler
          set_unexpected(unexpected_handler unexpected_func);
```

В качестве аргумента в эту функцию передается указатель на вашу собственную функцию, заменяющую стандартную **unexpected.** А возвращается указатель на текущую функцию **unexpected.**

Например, вы можете ввести в свою программу собственную функцию обработки непредусмотренного исключения:

```
void MyUnexpected()
(
 throw;
)
```

В этом примере функция просто осуществляет повторную генерацию исключения. Сослаться на эту функцию можно, например, выполнив в нужный момент оператор:

```
set_unexpected(MyUnexpected);
```

Если вы хотите, чтобы всегда работала эта функция, то указанный оператор можно включить в обработчик события **OnCreate** вашей формы. А если вы хотите включать вашу функцию, заменяющую **unexpected,** только в каких-то фрагментах программы, возвращаясь потом к стандартной или другой вашей функции, то это можно сделать следующим образом:

```
unexpected_handler UH;
...
// Задание новой функции и запоминание предыдущей
UH = set_unexpected(MyUnexpected);

// Восстановление прежней функции и запоминание текущей
UH = set_unexpected(UH);
```

Если вы тем или иным способом ввели приведенную выше замену функции **unexpected,** вы сможете обрабатывать любые непредусмотренные исключения. Например, вызов описанного выше варианта функции F с возможностью генерации непредвиденного исключения, можно оформить так:

```
try
{
 ...
// вызов функции F
 double B = F(StrToFloat(B));
...
}
catch(overflow_error & e)
{
// перехват исключения overflow_error
 ShowMessage(e.what());
}
catch(exception & e)
{
// перехват всех прочих исключений
ShowMessage("Неожиданное исключение: "  +
             AnsiString(e.what()));
}
```

Другой способ предотвратить аварийное завершение программы при непредвиденных исключениях — заменить с помощью функции **set_terminate** стандартную функцию **terminate** точно так же, как это рассмотрено для **unexpected.**

1.13 Сигналы

Сигнал — это некоторое непредвиденное событие (прерывание), которое может вызвать преждевременное завершение программы. Перечислим некоторые из таких непредвиденных событий: прерывание программы, вызванное нажатием пользователем клавиш Ctrl-C, появление недопустимой команды, ошибочный доступ к памяти (нарушение сегментации), запрос от операционной системы о завершении работы, ошибка операций с вещественными числами (деление на нуль или перемножение слишком больших действительных чисел).

Ниже перечислены некоторые стандартные сигналы, определенные в заголовочном файле *signal.h* (полный список приведен в гл. 4, в описании функции **signal**).

SIGABRT	Аварийное завершение программы (например, в результате вызова функции abort). Действие по умолчанию — вызов _exit(3)
SIGFPE	Ошибка арифметической операции, например, деление на нуль или операция, вызвавшая переполнение. Действие по умолчанию — вызов _exit(l)
SIGINT	Получение интерактивного сигнала (например, прерывание Ctrl-C). Действие по умолчанию — прерывание INT 23h
SIGUSR1, SIGUSR2, SIGUSR3	Определенные пользователем (только в Win32) сигналы пользователя, генерируемые функцией raise. Действие по умолчанию — игнорирование сигнала

Библиотека обработки сигналов содержит функцию **signal,** перехватывающую сигналы. В функцию **signal** передаются два параметра: целочисленный номер сигнала и указатель на функцию обработки сигнала.

Обычно сигналы автоматически генерируются при возникновении соответствующих событий. Но программа может целенаправленно генерировать сигналы функцией raise, в которую передаются целочисленное значение номера сигнала.

Например, вы можете предусмотреть в своей программе обработчик некоторого вводимого вами сигнала **SIGUSR1.** Пусть вы дали ему имя **Handl_SIGUSR1.** Тогда где-то в программе (например, при обработке события **OnCreate** формы) вам надо установить в системе этот обработчик с помощью оператора

```
signal(SIGUSR1, Handl_SIGUSR1);
```

При этом не забудьте вставить в файл директиву

```
#include <signal.h>
```

Сам обработчик сигнала может иметь вид:

```
void Handl_SIGUSR1(int N)
{
 ...
 if(MessageDlg("Продолжать?", mtConfirmation,
              TMsgDlgButtons() << mbYes << mbNo,0) == mrYes)
// повторная установка обработчика:
  signal(SIGUSR1, Handl_SIGUSR1);
 else exit(EXIT_SUCCESS);
}
```

Этот обработчик принимает одно целое значение, соответствующее номеру сигнала. В обработчике предусматриваются некоторые действия, необходимые при появлении данного сигнала. Затем, если выполнение программы должно продолжаться, надо повторно установить обработчик сигнала с помощью функции **signal,** как показано в приведенном примере. Если этого не сделать, то последующие события **SIGUSR1** не будут вызывать этот обработчик. После выполнения команды повторной установки обработчика сигнала управление автоматически передается в точку программы, в которой сигнал был обнаружен. В этом, в частности, коренное отличие сигналов от исключений.

Генерация сигнала **SIGUSR1** в необходимых местах программы осуществляется оператором

```
raise(SIGUSR1);
```

Рассмотренный вариант функции сейчас считается несколько устаревшим. Более современный вариант (подробнее о нем см. в гл. 4, в описании функции **signal)** выглядит следующим образом. Определите в программе тип указателя на функцию **fptr:**

```
typedef void  (*fptr)(int);
```

Этот указатель используется в вызове функции **signal:**

```
signal(SIGFPE, (fptr)Handl_SIGFPE);
```

Весь остальной приведенный выше текст примера может не изменяться.

1.14 Сообщения Windows и их обработка

1.14.1 Обработка сообщений в приложениях C++Builder

Приложения Windows состоят из множества объектов, которые взаимодействуют друг с другом, обмениваясь сообщениями (messages). Источниками этих сообщений могут быть: пользователь, оперирующий с клавиатурой и мышью, среда Windows, посылающая сообщения приложениям, другие приложения, обменивающиеся информацией с вашим приложением и, наконец, ваше приложение, посылающее сообщения компонентам.

Большинство сообщений, которые вам могут потребоваться, C++Builder обрабатывает сам, так что вам достаточно использовать обработчики стандартных событий компонентов. Но иногда вам может потребоваться самому обрабатывать сообщения Windows. Такая необходимость возникает, если нужное вам сообщение пока еще компонентами C++Builder не обрабатывается, или если вы определили свое собственное сообщение.

Сообщение Windows представляет собой структуру **TMessage,** содержащую поля. Наиболее важное из них — Msg содержит целое значение, идентифицирующее данное сообщение. В модуле *Messages.hpp* в C++Builder содержатся объявления множества идентификаторов, позволяющие оперировать с мнемоническими именами сообщений, а не с какими-то целыми значениями. Важная информация о сообщении содержится также в двух полях параметров - - **wParam (word parameter** — параметр типа **word) и lParam (long parameter** — параметр типа **long)** и в поле результата **Result.** Впрочем, реально в C++Builder все эти поля имеют тип **int.** Каждое из них может быть также представлено как комбинация двух полей типа **Word: WParamLo, WParamHi, LParamLo, LParamHi, ResultLo, ResultHi.** В этих именах окончание "Hi" относится к старшим разрядам соответствующего параметра, а окончание "Lo" - к младшим разрядам.

В API Windows и в C++Builder для большинства сообщений введены мнемонические имена параметров. Так что теперь, например, при обработке сообщения от мыши можно ссылаться на понятные параметры **XPos и YPos,** а не на стандартные и ни о чем не говорящие имена **lParamLo и lParamHi.**

В Windows предусмотрено множество сообщений. Для дальнейших экспериментов нам потребуется только два сообщения. Первое из них — **WM_CLOSE,** сигнализирующее, что окно или приложение закрывается. Это сообщение не имеет параметров. По умолчанию оно уничтожает окно, которому послано. Если приложение обрабатывает это сообщение, то оно должно возвращать нуль.

При обработке данного сообщения приложение может запросить пользователя о необходимости закрывать окно и вызвать функцию закрытия окна только при положительном ответе.

Второе сообщение, которое мы будем использовать — **WM_ACTIVATE.** Оно посылается, когда окно переводится в активное или неактивное состояние. Сначала сообщение посылается окну, переходящему в неактивное состояние, а потом — активируемому.

Это сообщение определено следующим образом:

```
WM_ACTIVATE
  fActive = LOWORD(wParam);
  fMinimized = (BOOL) HIWORD(wParam);
  hwndPrevious = (HWND) lParam;
```

Параметры этого сообщения означают следующее.

Параметр **fActive** показывает, как активируется или деактивируется окно. Возможные значения параметра:

WA_ACTIVE	Окно активируется не щелчком мыши (например, функцией Set Active Window или клавиатурой).
WA_CLICKACTIVE	Окно активируется щелчком мыши.
WA_INACTIVE	Окно деактивируется.

Параметр **fMinimized** показывает, свернуто окно, или нет. Ненулевое значение соответствует свернутому окну.

Параметр **hwndPrevious** — это дескриптор, который указывает на окно, из которого фокус переключился на данное окно, если оно активируется, или на окно, в которое передается управление, если данное окно деактивируется.

По умолчанию, если активируемое окно не свернуто, то оно получает фокус. Если приложение обрабатывает это сообщение, то оно должно возвращать нуль.

1.14.2 Посылка сообщений

В API Windows определен ряд функций, позволяющих послать сообщение.

1.14.2.1 Функции SendMessage, PostMessage и Perform

Функция **SendMessage** посылает указанное в ней сообщение окну или множеству окон и не возвращается, пока это сообщение обрабатывается. Этим она отличается от функции **PostMessage,** которая возвращается сразу после передачи сообщения.

Объявление функции SendMessage:

```
Int SendMessage(HWnd hWnd, unsigned int Msg,
                WPARAM wParam, LPARAM lParam);
```

Параметр **hWnd** — дескриптор окна, которому передается сообщение. Если этот параметр равен **HWND_BROADCAST,** то сообщение передается всем окнам верхнего уровня в системе, включая недоступные и невидимые, кроме дочерних.

Параметр Msg определяет передаваемое сообщение. Параметры **wParam** и **lParam** могут содержать дополнительную информацию. Значение, возвращаемое функцией, зависит от вида сообщения.

Функция **PostMessage** объявлена следующим образом:

```
bool PostMessage(HWND hWnd, unsigned int Msg,
                 WPARAM wParam, LPARAM lParam);
```

Эта функция похожа на **SendMessage,** но в отличие от нее она помещает сообщение в очередь и сразу возвращается. Таким образом, **PostMessage** не годится для передачи срочных сообщений, но зато она не блокирует вызвавшее приложение на время обработки сообщения приемником.

Параметры **hWnd** и Msg аналогичны рассмотренным для функции **SendMessage.** Если **hWnd = NULL,** то сообщение ставится в очередь сообщений (если она есть) текущего процесса.

Функция **PostMessage** возвращает ненулевое значение при успешном завершении и нуль в случае аварийного завершения. В этом случае причину ошибки можно установить вызовом функции **GetLastError.**

Имеется еще один метод, который может посылать сообщение непосредственно оконному компоненту. Это метод **Perform,** объявление которого имеет вид:

```
Perform(unsigned int Msg, WPARAM wParam, LPARAM lParam);
```

Есть еще ряд функций, позволяющих передавать сообщения, но мы на них не будем останавливаться, так как они реже используются в приложениях C++Builder.

1.14.2.2 Пример посылки сообщений

Давайте построим простую программу, демонстрирующую посылку сообщений. Начните новый проект, создайте в нем две формы **Forml и Form2,** сохраните проект, назвав модули форм **UlMessl и U2Messl** соответственно, а файл проекта — **PMessl.** Форма **Forml** будет у нас главной, и она будет управлять видимостью формы **Form2.** Поэтому в ее модуль введите директиву препроцессора

```
#include "U2Mess1.h"
```

а свойство Visible формы Form2 должно быть равно false.

Перенесите на форму Forml две кнопки, дав им надписи "Show Form2" и "Close Form2". В обработчике щелчка первой кнопки напишите оператор

```
Form2->Show();
```

а в обработчике щелчка второй кнопки — оператор

```
SendMessage(Form2->Handle,WM_CLOSE,0,0);
```

Этот оператор посылает сообщение **WM_CLOSE** (второй параметр функции **SendMessage)** форме **Form2.** Первый параметр функции **SendMessage** содержит дескриптор окна этой формы, полученный с помощью ее свойства **Handle.** Сообщение **WM_CLOSE** не имеет параметров; поэтому параметры **wParam и lParam** заданы равными нулю.

Приведенный выше оператор можно заменить следующим:

```
Form2->Perform(WM_CLOSE,0,0);
```

Результат будет тем же самым.

Сохраните и запустите приложение. Вы увидите, что нажатие пользователем кнопки Show Form2 приводит к появлению на экране формы **Form2,** а нажатие кнопки Close Form2 — к ее закрытию. Так что посылаемое формой **Forml** сообщение достигает своего адресата.

Конечно, этот пример не очень интересный, поскольку сделать невидимой форму **Form2** мы могли бы, не посылая никаких сообщений Windows, а просто используя метод **Hide.** Поэтому пойдем дальше. Откройте новый проект, задайте заголовок **Caption** ее формы равным "Приложение Pmess2" (этот текст мы будем далее использовать для идентификации окна этого приложения) и сохраните проект под именем **PMess2,** а модуль формы — под именем **UlMess2.** Откомпилируйте новый проект и сохраните. В дальнейшем мы еще к нему вернемся.

Теперь давайте попробуем управлять этим проектом из формы **Forml** проекта **PMessl.** Откройте опять проект **PMessl** и добавьте на форму **Forml** две кнопки, сделав на них надписи "Exec Pmess2" и "Close Pmess2". В обработчике щелчка первой кнопки напишите оператор

```
WinExec("Pmess2.exe",SW_RESTORE);
```

который, как вы уже знаете, запускает приложение **PMess2** на выполнение. А теперь давайте попробуем его закрыть. Для этого в обработчике щелчка кнопки Close Pmess2 напишите оператор

```
SendMessage(FindWindow("TForm1","Приложение Pmess2"), WM_CLOSE,0,0);
```

Этот оператор использует функцию **FindWindow** для получения дескриптора окна приложения, которому надо послать сообщение, а затем функцией **SendMessage** посылает сообщение **WM_CLOSE.**

Сохраните и запустите приложение. Теперь, нажимая кнопку Exec Pmess2, вы можете выполнять приложение **PMess2,** причем можете создать несколько экземп-

ляров этого приложения. А кнопкой Close Pmess2 можете закрывать приложение PMess2.

Вы получили возможность управлять из своего приложения другими. Но в данном случае вы знали класс окна **(TForml)** внешнего приложения, поскольку вы сами его создавали. Поэтому вы смогли применить функцию **FindWindow**, передав в нее и имя класса, и заголовок окна. А как быть, если вы хотите закрыть какое-то чужое приложение? Например, вы из своего приложения выполнили стандартное приложение Windows «Калькулятор», пользователь посчитал, что ему было надо, а теперь вы хотите закрыть «Калькулятор», послав ему сообщение **WM_CLOSE** (например, пользователь уже не работает с ним, а закрыть забыл).

При посылке сообщения другому приложению возникает задача определить дескриптор нужного окна. Если вы определили имя класса необходимого вам приложение (например, SciCalc для приложения «Калькулятор») с помощью WinSight 32 и хотите послать из своего приложения сообщение о закрытии калькулятора, вы можете выполнить оператор:

```
SendMessage(FindWindow("SciCalc","Калькулятор"),WM_CLOSE,0,0);
```

1.14.3 Обработка сообщений

Во всех оконных компонентах предусмотрены обработчики сообщений Windows по умолчанию. До сих пор вы пользовались именно стандартными обработчиками сообщений. Однако вы можете определить и свои собственные обработчики, заменив ими обработчики по умолчанию, или дополнив их.

Для введения собственного обработчика сообщения Windows надо сделать следующее:

1. Создать в объявлении класса карту (тар) сообщений и ввести в нее те сообщения, которые вы хотите обрабатывать сами.
2. Добавить в объявление класса объявления вводимых вами обработчиков.
3. Описать эти обработчики в вашем модуле.

Первый шаг — объявление карты сообщений, легко осуществляется с помощью следующих макросов:

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(parameter1, parameter2, parameter3)
  ...
END_MESSAGE_MAP
```

Макрос **BEGIN_MESSAGE_MAP** открывает объявление карты сообщений, макрос **END_MESSAGE_MAP** завершает это объявление, а один или несколько макросов **MESSAGE_HANDLER** вводят в карту соответствующие сообщения. В макросе **MESSAGE_HANDLER** первый параметр указывает имя сообщения, второй — тип структуры сообщения, а третий — имя функции-обработчика.

Имя сообщения пишется заглавными буквами и должно соответствовать предопределенному в Windows сообщению. Например, **WM_CLOSE.** Имя типа структуры сообщения может быть любым, но обычно принято делать его тождественным имени обрабатываемого сообщения с исключенным из него символом подчеркивания и добавленным префиксом "T". Например, **TWMClose.** Передаваемый в обработчик параметр этого типа представляет собой структуру, через которую в обработчик передаются параметры сообщения, а из обработчика возвращается значение поля **Result,** фиксирующее результат обработки. Имя функции-обработчика сообщения также может быть любым, но обычно оно образуется из имени сообщения исключением первых символов "WM_" и добавлением префикса "On". Например, **OnClose.**

Давайте введем в рассмотренное ранее наше приложение **PMess2** и в форму **Form2** приложения **PMessl** обработку тех сообщений **WM_CLOSE,** которые мы посылаем им из формы **Form1.** Для этого поместим на форму **Form1** приложения **PMess2** и на форму **Form2** приложения **PMessl** метки **Labell,** чтобы отображать в них текст, свидетельствующий о том, что обработка сообщения действительно происходит. Введем в модулях этих форм следующие обработчики (текст приведен для **Form2;** для **Forml** все то же самое с заменой идентификатора **Form2 на Forml):**

```
// модуль U2Mess1.h
class TForm2 : public TForm
{
__published:      // IDE-managed Components
   TLabel *Label1;
private:   // User declarations
  void___fastcall OnClose(TWMClose& Message);
public:           // User declarations
   __fastcall TForm2(TComponent* Owner);
  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_CLOSE, TWMClose, OnClose)
  END_MESSAGE_MAP(TComponent)
};
//---------------------------------------------------------------------
// модуль U2Mess1.cpp
...
void___fastcall TForm2::OnClose(TWMClose& a)
 (
 Label2->Caption = "Караул! Закрывают!";
if (MessageDlgPos("Меня хотят закрыть. Согласны?",
 mtConfirmation, TMsgDlgButtons() << mbYes<< mbNo << mbCancel,
 0,BoundsRect.Left,BoundsRect.Bottom) == mrYes)
  Close ();
else Label2->Caption = "Не закроюсь!";
 a.Result = 0;
}
```

В этом коде в файле **U2Mess1.h** в разделе **public** объявлена карта сообщений, в которую включено сообщение **WM_CLOSE.** Для этого сообщения объявлен обработчик **OnClose** и тип передаваемого в него параметра — **TWMClose.** В разделе **private** объявлена функция этого обработчика. В нее передается параметр, названный message — структура сообщения.

В обработчике **OnClose** сообщения **WM_CLOSE** с параметром, которому дано имя a, производится запрос подтверждения пользователя о закрытии окна. Для запроса использована процедура **MessageDlgPos,** позволяющая указать позицию окна запроса вблизи окна формы, к которой относится запрос. При положительном ответе пользователя окно закрывается методом **Close.** В заключение оператором **a.Result = O** возвращается нуль, так как это действие оговорено в приведенном ранее описании сообщения **WM_CLOSE.**

Откомпилируйте оба ваших приложения и выполните приложение **PMessl.** После того как вы сделаете кнопкой Show Form2 видимой форму **Form2** и запустите кнопкой Exec Pmess2 приложение **PMess2,** вы увидите, что окна формы **Form2** и **PMess2** оказывают сопротивление попыткам их зарыть, независимо от того, как это делается: кнопками Close Form2 и Close Pmess2, или кнопками в полосах заголовков этих окон. В любом случае они закрываются только после подтверждения пользователя.

Кстати, вы можете наглядно посмотреть и очередь сообщений. Щелкните на кнопке Close Pmess2 и, пока на экране отображено окно запроса, вернитесь, ничего не отвечая, в окно формы **Forml** первого приложения и щелкните еще несколько раз на кнопке Close Pmess2. После этого ответьте на запрос отрицательно. Вы уви-

дите, что окно запроса без всяких дополнительных действий с вашей стороны будет отображаться еще столько раз, сколько раз вы щелкнули перед этим на Close Pmess2. Все сообщения, связанные с этими щелчками, запомнились в очереди и теперь очередь разгружается.

Приведенный пример обработки сообщений не очень показательный, поскольку сообщение **WM_CLOSE** не имеет параметров. Давайте, усовершенствуем наши приложения так, чтобы поработать с сообщениями, имеющими параметры. Воспользуемся для этого описанным ранее (разд. 1.14.1) сообщением **WM_ACTIVATE.** Это сообщение получает любое окно при его активации или деактивации. Сообщение имеет, в частности, параметр **f Active.** Значение **WA_INACTIVE** этого параметра показывает, что окно деактивируется. Иные значения параметра показывают, что окно активируется.

Давайте введем во все наши формы **(Form1 и Form2** приложения **PMessl, и Forml** приложения **PMess2)** обработчики сообщения **WM_ACTIVATE.** Чтобы видеть, что эти обработчики работают правильно, поместим на эти формы дополнительно метки **Label2,** в которых будем отображать результаты обработки. И во все три формы введем обработчики вида (приводится текст для формы **Form2):**

```
// модуль U2Mess1.h
class TForm2 : public TForm
{
__published:       // IDE-managed Components
    TLabel *Label1;
    TLabel *Label2;
private:   // User declarations
  void__fastcall OnClose(TWMClose& Message);
  void__fastcall OnActivate(TWMActivate& Message);
public:           // User declarations
    __fastcall TForm2(TComponent* Owner);
  BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_CLOSE, TWMClose, OnClose)
    MESSAGE_HANDLER(WM_ACTIVATE, TWMActivate, OnActivate)
  END_MESSAGE_MAP(TComponent)
};
//---------------------------------------------------------
// модуль U2Mess1.cpp
...
void__fastcall TForm2::OnActivate(TWMActivate& a)
{
 if (a.Active == WA_INACTIVE)
  Label2->Caption = "Меня покинули!";
 else Label2->Caption = "Ура! Я работаю!";
 a.Result = 0;
}
```

Обработчик **OnActivate** строится по тем же принципам, что и предыдущий. В нем анализируется значение поля **Active** структуры a, передаваемой в процедуру в качестве параметра и содержащей параметры сообщения. В зависимости от значения этого параметра производятся те или иные действия.

Откомпилируйте оба ваших приложения и выполните приложение **PMessl.** Вы увидите, как при каждом переключении фокуса между окнами в них появляются сообщения, отражающие потерю и обретение ими активности.

1.14.4 Определение собственных сообщений

В предыдущих разделах посылались и обрабатывались сообщения, предопределенные API Windows. Однако вы можете описать свои собственные сообщения и работать с ними так же, как с предопределенными.

Номера своих собственных сообщений вы должна отсчитывать от константы **WM_USER,** которая соответствует первому номеру сообщения пользователя.

Например, вы можете определить в своем приложении константы

```
#define WM_MyMess1 WM_USER
#define WM_MyMess2 WM_USER + 1
```

и затем оперировать с сообщениями **WM_MyMessl и WM_MyMess2** как с предопределенными в Windows, Например, можете вставить в карту сообщений объявление:

```
MESSAGE_HANDLER(WM_MyMess1, TMessage, OnMyMess1)
```

В данном объявлении в качестве типа параметра использован тип **TMessage.** Этот тип определяет следующую структуру:

```
struct TMessage
{
 unsigned int Msg;
 long WParam;
 long LParam;
 long Result;
};
```

Вы можете при посылке сообщения передавать параметрами **WParam и LParam** любую необходимую информацию.

В качестве примера давайте добавим в нашем тестовом приложении **PMessl** на форму **Form1** компонент **CSpinEdit** и кнопку, задав ей надпись Послание. В обработчик щелчка на этой кнопке мы хотим вставить посылку сообщения второму нашему приложению — **PMess2,** в котором в качестве кода послания передать число, установленное пользователем в компоненте **CSpinEdit.**

Для того чтобы сделать это, объявите в заголовочном файле **U1Mess1.h** номер вашего сообщения с именем, например, **WM_MyPost:**

```
#define WM_MyPost WM_USER
```

а в обработчик события щелчка на кнопке Послание вставьте оператор

```
SendMessage(FindWindow("TForm1","Приложение Pmess2"),
            WM_MyPost,0,CSpinEdit1->Value);
```

Вот и все! Сообщение будет посылаться. Можно было даже сделать проще: не вводить константы, а просто в функции **SendMessage** указать вместо **WM_MyPost** номер сообщения — **WM_USER.** Но с именем сообщения код читается проще.

Теперь осталось написать обработчик этого сообщения в приложении **PMess2.** Это выглядит так же, как и для других обработчиков сообщений:

```
// модуль U1Mess2.h
...
#define WM_MyPost WM USER
//---------------------------------------------------------
class TForml : public TForm
{
  ...

private:          // User declarations
  void__fastcall OnMyPost(TMessage& Message);
public:           // User declarations
   __fastcall TForm1(TComponent* Owner);
  BEGIN_MESSAGE_MAP
    ...
    MESSAGE_HANDLER(WM_MyPost, TMessage, OnMyPost)
  END_MESSAGE_MAP(TComponent)
);
//---------------------------------------------------------
```

```
// модуль  U1Mess2.cpp
...
void__fastcall TForm1::OnMyPost(TMessage& a)
{
 Label2->Caption = "Получено письмо " + IntToStr(a.LParam);
}
```

В объявлении и реализации обработчика использован тип TMessage — стандартный тип параметра сообщения Windows.

Можете запускать приложения и убедиться, что послание нормально передается и воспринимается. Конечно, это просто демонстрация возможностей обмена собственными сообщениями. В реальных приложениях можно по номеру параметра оператором switch выбирать тот или иной вид реакции приложения на полученное сообщение.

Глава 2

Типы данных в языке C++

В данной главе не затрагивается множество типов, объявленных в стандартной библиотеке шаблонов STL, так как их невозможно рассматривать, пока не изложены основы создания шаблонов (разд. 2.14.7). Все, связанное с STL, рассмотрено в гл. 5.

2.1 Классификация типов данных, объявление типов

Все типы, используемые в C++Builder, можно разбить на четыре группы:

Aggregate		структуры данных
	Array	массивы
	struct	структуры
	union	объединения
	class	классы
Function		функции
Scalar		скалярные
	Arithmetic	арифметические
	Enumeration	перечислимые
	Pointer	указатели
	Reference	ссылки
void		отсутствие значения

Другой способ классификации типов связан с их разбиением на *основные* и *производные* типы. К *основным* относятся: **void, char, int, float и double,** а также их варианты с модификаторами **short** (короткий), **long** (длинный), **signed** (со знаком) и **unsigned** (без знака). Например, **unsigned char, unsigned int, signed int** (модификатор **signed** подразумевается по умолчанию и поэтому обычно не указывается).

Основные типы в C++ следующие:

Тип	Размер в байтах	Диапазон значений
char	**1**	**от -128 до 126**
unsigned char	**1**	**от 0 до 255**
short	**2**	**от -32 768 до 32 767**

Тип	Размер в байтах	Диапазон значений
unsigned short	2	от 0 до 65 535
enum	2	от -2 147 483 648 до 2 147 483 647
long	4	от -2 147 483 648 до 2 147 483 647
unsigned long	4	от 0 до 4 294 967 295
int	4	как в long
unsigned int	4	как в unsigned long
float	4	от $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{38}$
double	8	от $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{308}$
long double	10	от $3.4 \cdot 10^{-4932}$ до $1.1 \cdot 10^{4932}$
bool	1	true или false

Имеются также основные типы __int8, __**int16**, __int32, __**int64**, о которых подробнее см. в разд. 2.3.

Следует отметить, что в C++Builder, в отличие от некоторых других версий C++, булев тип **bool** реализован как отдельный тип, а не как псевдоним целого. Такое определение булева типа в настоящий момент зафиксировано и в новом стандарте C++. Однако это не мешает при желании использовать в логических выражениях целые значения вместо булевых. При этом значение 0 расценивается как **false,** а любое ненулевое значение — как **true.**

Производные типы включают в себя указатели и ссылки на какие-то типы, массивы каких-то типов, типы функций, классы, структуры, объединения. Эти типы считаются производными, поскольку, например, классы, структуры, объединения могут включать в себя объекты различных типов.

Можно выделить еще одну категорию типов — *порядковые,* в которых значения упорядочены и для каждого из них можно указать предшествующее и последующее. К ним относятся целые, символы, перечислимые типы.

Типы данных указываются при объявлении любых переменных и функций (см. разд. 1.6.1 и 1.7.1). Например:

```
double a = 5.4, b = 2;
int c;
void F1(double A);
```

Пользователь может вводить в программу свои собственные типы. Объявления типов могут делаться в различных местах кода. Место объявления влияет на область видимости или область действия так же, как и в случае объявления переменных (см. разд. 1.8).

Синтаксис объявления типа:

```
typedef определение_типа идентификатор;
```

Здесь идентификатор — это вводимое пользователем имя нового типа, а определение_типа — описание этого типа. Например, оператор

```
typedef double Ar[10];
```

объявляет тип пользователя с именем **Ar** как массив из 10 действительных чисел. В дальнейшем на этот тип можно ссылаться при объявлении переменных. Например:

```
Ar A = {1,2,3,4,5,6,7,8,9,10};
```

Объявление типа с помощью **typedef** можно использовать и для создания нового типа, имя которого будет являться псевдонимом стандартного типа C++. Имен-

но так в C++Builder многие встроенные типы компонентов Object Pascal приведены к типам, характерным для C++. Эти переопределения типов содержатся в файле **sysdefs.h**. Например:

```
typedef bool Boolean;
typedef int Integer;
typedef short Smallint;
typedef unsigned char Byte;
```

Ниже дается таблица соответствия типов Delphi (т.е. типов Object Pascal) и типов C++.

Delphi	Размер или значение	Соответствие C++	Реализация
ShortInt	целое 8 бит	signed char	typedef
Smallint	целое 16 бит	short	typedef
LongInt	целое 32 бита	int	typedef
Byte	целое без знака 8 бит	unsigned char	typedef
Word	целое без знака 16 бит	unsigned short	typedef
Integer	целое 32 бита	int	typedef
Cardinal	целое без знака 32 бита	unsigned int	typedef
Boolean	true/false	bool	typedef
ByteBool	true/false или целое без знака 8 бит	unsigned char	typedef
WordBool	true/false или целое без знака 16 бит	unsigned short	typedef
LongBool	true/false или целое без знака 32 бита	BOOL (WinAPI)	typedef
AnsiChar	символ без знака 8 бит	char	typedef
WideChar	символ Unicode размером в слово	wchar_t	typedef
Char	символ без знака 8 бит	char	typedef
AnsiString	AnsiString Delphi	AnsiString	класс
String[n]	прежний стиль строк Delphi, n = 1..255 бит	SmallString<n>	шаблон класса
ShortString	прежний стиль строк Delphi, 255 бит	SmallString<255>	typedef
String	AnsiString Delphi	AnsiString	typedef
Single	число с плавающей запятой 32 бита	float	typedef
Double	чиело с плавающей запятой 64 бита	double	typedef
Extended	число с плавающей запятой 80 бит	long double	typedef
Real	число с плавающей запятой 32 бита	double	typedef

Delphi	Размер или значение	Соответствие C++	Реализация
Pointer	родовой указатель 32 бита	void *	typedef
PChar	указатель на символы 32 бита	unsigned char *	typedef
PAnsiChar	указатель на символы ANSI 32 бита	unsigned char *	typedef
Comp	число с плавающей запятой 64 бита	Comp	класс
OleVariant	значение variant OLE	OleVariant	класс

2.2 Приведение типов

В арифметических выражениях, содержащих элементы различных арифметических типов, C++Builder в процессе вычислений автоматически осуществляет преобразование типов. Это стандартное преобразование всегда осуществляется по принципу: если операция имеет операнды разных типов, то тип операнда «младшего» типа приводится к типу операнда «старшего» типа. Иначе говоря, менее точный тип приводится к более точному. Например, если в операции участвует короткое целое и длинное целое, то короткое приводится к длинному; если участвует целый и действительный операнды, то целый приводится к действительному и т.д. Таким образом, после подобного приведения типов оба операнда оказываются одного типа. И результат применения операции имеет тот же тип.

Все это относится к арифметическим операциям, но не относится к операции присваивания. Присваивание сводится к приведению типа результата выражения к типу левого операнда. Если тип левого операнда «младше», чем тип результата выражения, возможна потеря точности или вообще неправильный результат.

Рассмотрим примеры неявного автоматического преобразования типов. В результате действия следующих операторов

```
double a = 5.4, b = 2;
int c = a * b;
```

переменная с получит значение 10, хотя истинное значение должно быть равно 10.8. Это значение действительно будет вычислено в результате умножения a * b, но затем дробная часть будет отброшена, поскольку с — целая переменная.

Результатом выполнения операторов

```
int m = 1, n = 2;
double A = m / n;
```

будет значение A = 0. Поскольку m и n — целые переменные, то деление m / n сведется к целочисленному делению с отбрасыванием дробной части, результат которого равен нулю.

Результат выполнения похожих на предыдущие операторов

```
int m = 1;
double n = 2;
double A = m / n;
```

даст правильный результат — A = 0.5. Поскольку в данном случае один из операндов операции деления имеет тип **double,** то тип другого, целого операнда будет тоже приведен к **double,** и результат деления будет иметь тип **double.**

Еще один пример, который дает совершенно неверный результат:

```
double a = 300, b = 200;
short c = a * b;
```

Если вы попробуете реализовать этот пример, то увидите, что переменная с получит значение -5536, вместо ожидаемого 60 000. Дело в том, что переменная типа **short** может хранить значение не больше, чем 32 767. Поскольку выражение в правой части приведенного оператора дает результат 60 000, то его присваивание переменной типа **short** дает совершенно неверное значение.

Как было видно из некоторых приведенных примеров, неявное автоматическое приведение типов не всегда дает желаемый результат. Это можно исправить, применив операцию явного приведение типов. Она записывается в виде

```
(тип)
```

перед той величиной, которую вы хотите привести к указанному типу. Вернемся к уже рассмотренному примеру

```
int m = 1, n = 2;
double A = m / n;
```

который давал неверное значение переменной A. Этот результат можно исправить, применив во втором операторе явное приведение типа:

```
double A = (double)m / n;
```

В этом случае переменная т, к которой применяется операция приведения типа, рассматривается как действительная величина типа **double.** Тогда и переменная n неявно приводится к типу **double,** так что деление осуществляется уже не с целыми, а с действительными числами. Результат получается правильным — 0.5.

Есть еще одна ситуация, которая требует явного приведения типов: в некоторых случаях компилятор не может выбрать среди перегруженных функций (о перегрузке функций см. разд. 1.7.7), если под данный тип параметра подходит несколько из них. Если в C++Builder 4 вы напишете код

```
TPoint P;
P.x = 5;
P.y = 1;
Label1->Caption = "Координата x = " + IntToStr(P.x);
```

то получите сообщение компилятора об ошибке: "Ambiguity between '_fastcall Sysutils::IntToStr(__int64)' and '_fastcall Sysutils::IntToStr(int)'" (Неоднозначность применения функции IntToStr к параметрам типов___int64 и int). Компилятор, как Буриданов осел, остановился между двумя (в данном случае идентичными) возможностями и отказывается производить выбор. Помочь компилятору легко, применив в последнем из приведенных операторов явное указание типа **int:**

```
Label1->Caption = "Координата x = " + IntToStr((int)P.x);
```

Подобный текст компилятор обработает без проблем.

В C++Builder 6 и 5 компилятор более «интеллектуальный» и в приведенном примере в подобной помощи не нуждается. Но в некоторых других сложных случаях подобное явное приведение типов может потребоваться. Ряд примеров явного приведения типов вы найдете в разд. 2.8.3.

В новой версии C++ введен ряд новых операций приведения типа, которые, по замыслу, должны заменить традиционную скобочную операцию, рассмотренную выше. Первая из этих операций — **static_cast:**

```
static_cast< T > (arg)
```

В этом выражении **arg** - переменная, а T - тип, к которому приводится тип этой переменной. Например, если R - действительное число, то следующие выражения осуществляют указанное в комментариях приведение типа:

```
// приведение R к целому, аналог (int)R :
static_cast<int>(R)
// приведение R к строке, аналог (AnsiString)R:
static_cast<AnsiString>(R);
```

С помощью **static_cast** можно осуществлять преобразование целого типа в перечислимый, нулевого указателя в нулевой указатель типа T, указателя на объект одного типа в указатель на объект другого типа. Можно также преобразовывать указатель на класс X в указатель на один из наследующих ему классов Y (см. пример в разд. 2.8.3).

Заданное операцией **static_cast** приведение типов осуществляется на этапе компиляции. Имеется также операция явного приведения типов **dynamic_cast,** которая позволяет осуществлять приведение типа и проверку его корректности во время выполнения. Но это преобразование применимо только к указателям и ссылкам на классы. Так что дальнейшее имеет смысл читать, только если вы владеете знаниями по классам, указателям и ссылкам или уже изучили материал разд. 2.8, 2.9, 2.14. Синтаксис операции **dynamic_cast:**

```
dynamic_cast< T > (ptr)
```

Здесь T - указатель (см. разд. 2.8) или ссылка (см. разд. 2.9) на тип класса или **void*,** а **ptr** - выражение, которое может быть приведено к указателю или ссылке. Если T равно **void*,** то **ptr** должен быть указателем. В этом случае результирующий указатель может давать доступ к объектам класса, непосредственно наследующего в иерархии исходному классу. Такой класс не должен быть базовым для каких-то других классов.

Преобразование класса-наследника в базовый класс или одного класса-наследника в другой требует выполнения следующего условия: если T -- указатель и **ptr** — указатель на класс, не являющийся базовым, результат является указателем на уникальный подкласс. Аналогичное условие накладывается на ссылки: если T — ссылка и **ptr** — ссылка на класс, не являющийся базовым, результат является ссылкой на уникальный подкласс.

Преобразование класса-наследника в базовый класс возможно только в случае, если базовый класс полиморфного типа.

Преобразование в базовый класс осуществляется во время компиляции. А преобразования из базового класса в класс-наследник и **преобразования,** не связанные с иерархией, осуществляется во время выполнения. В этом основное отличие **dynamic_cast от static_cast:** в возможности преобразования типов и проверки корректности преобразования во время выполнения. Если преобразование завершилось успешно, возвращается указатель или ссылка заданного типа. В противном случае при преобразовании указателя возвращается 0, а при преобразовании ссылки генерируется исключение **Bad_cast.**

Развернутый пример применения **dynamic_cast** вы можете посмотреть в разд. 2.14.6.

Имеется еще две операции явного приведения типа. Операций **const_cast:**

```
const_cast< T > (arg)
```

позволяет удалить или добавить в тип переменной **arg** спецификаторы **const и volatile** (см. разд. 1.6.2), не изменяя в остальном тип **arg.** Операция **reinterpret_cas:**

```
reinterpret_cast< T > (arg)
```

позволяет преобразовать указатель в целый тип или наоборот, целый тип в указатель. Впрочем, поведение **reinterpret_cas** зависит от конкретной реализации компилятора. Так что лучше не использовать эту операцию, чтобы не ограничивать переносимость вашего приложения.

2.3 Арифметические типы данных

Арифметические типы данных — это целые и действительные типы.

К целым типам относятся **char, short, int и long** вместе с их вариантами **signed** — со знаком и **unsigned** — без знака. Из этих ключевых слов может форми-

роваться множество целых типов данных. Многие из них являются синонимами друг друга, как следует из следующей таблицы.

Синонимы	Примечания
char, signed char	Синонимы, если умолчанием для char задано signed
unsigned char	
char, unsigned char	Синонимы, если умолчанием для char задано unsigned
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

Спецификаторы **signed и unsigned** могут применяться только к **char, short, int, long.** Если тип обозначен просто как **signed** или **unsigned,** то подразумеваются соответственно **signed int и unsigned int.**

При отсутствии в указании типа спецификатора **unsigned** для целых типов подразумевается **signed.** Исключением из этого правила является тип **char.** C++Builder позволяет вам установить в качестве умолчания для **char signed** или **unsigned.** В этом случае, если вы пишете объявление

```
char ch;
```

оно воспринимается как

```
signed char ch;
```

Если же вы хотите объявить переменную типа **char** без знака, вы должны это сделать явно:

```
unsigned char ch;
```

Спецификаторы **long и short** могут использоваться только с int. Если тип обозначен просто как **long** или **short,** то подразумеваются соответственно **long int и short int.**

Объем памяти, занимаемый различными целыми типами, не лимитирован стандартом ANSI C. Указано только, что **short, int и long** должны образовывать неубывающую последовательность, т.е. **short <= int <= long.** Поэтому не исключается, что все три типа требуют одинакового объема памяти. Таким образом, объем памяти может меняться от одной платформы к другой и это надо учитывать, если хотеть создавать переносимые программы.

Объемы памяти, занимаемые целыми типами в данной версии C++Builder для 32 разрядных программ, приведены в таблице в разд. 2.1. В частности, из этой таблицы вы можете увидеть, что **int** и **long** эквивалентны и занимают по 32 бита.

Типы со знаком используют старший бит для хранения знака: 0 — положительный, 1 — отрицательный.

Помимо рассмотренных выше имеются еще целые типы, имена которых начинаются с символов "__int", за которыми следует число бит. При записи констант этих типов можно использовать суффиксы i и ui, как показано в приведенной

ниже таблице. Впрочем, эти же суффиксы можно использовать и при задании значений переменных других целых типов.

Тип	Суффикс	Пример	Память (биты)
_int8	i8	__int8 c = 127i8;	8
__int16	i16	__int16 s = 32767i16;	16
__int32	i32	__int32 i = 123456789i32;	32
_int64	i64	_int64 big = 1234654321i64;	64
unsigned__int64	ui64	unsigned __int64 hugeInt = 1234567887654321ui64;	64

Основными типами данных для представления действительных чисел с плавающей запятой являются типы **float и double.** Первый из них размещается в 32 битах, второй — в 64. К типу **double** может применяться спецификатор **long,** который увеличивает размер памяти до 80 бит.

Диапазоны возможных значений и затраты памяти для действительных типов в данной версии C++Builder для 32 разрядных программ приведены в таблице в разд. 2.1. Стандарт ANSI C не накладывает никаких ограничений на способ реализации действительных чисел. Поэтому, если вы хотите делать переносимые программы, то не ориентируйтесь на тот или иной размер памяти для действительных типов, а используйте для определения этого размера операцию **sizeof** (см. разд. 1.9.10, гл. 1).

2.4 Типы символов

Символы относятся к порядковым типам данных (см. разд. 2.1). Для них определен порядок следования и для любого символа можно сказать, какой символ расположен перед ним или какой расположен после него.

Фундаментальными типами символов для C++Builder являются типы **AnsiChar (char) и WideChar (wchar_t).** Символы **AnsiChar** занимают в памяти 1 байт. Каждому символу соответствует целое число. Тип **AnsiChar** отображает все множество символов ANSI, а их последовательность соответствует локализации, заданной в операционной системе. Это значит, что последовательность символов кириллицы соответствует русскому алфавиту. Заглавные символы кириллицы соответствуют числам от 192 до 223, а строчные буквы - числам от 224 до 255. Исключением является буква "е" (число 168 соответствует заглавной букве, а 184 — строчной). Заглавным латинским символам соответствуют числа от 65 до 90, а строчным — от 97 до 122. Цифрам соответствуют числа от 48 (0) до 57 (1). В разд. 3.1.2 приведен более полный список символов и соответствующих им чисел.

С типами **char и AnsiChar** можно обращаться и как с целыми, и как с символами. Все зависит от контекста. Например, операторы

```
char ch = 'Б';
Label1->Caption = ch;
```

выведут в метку символ "Б". А при том же самом значении **ch** оператор

```
Label1->Caption = (unsigned char)ch;
```

выведет в метку "193" - число, соответствующее символу "Б".

Оператор

```
Label1->Caption = (char)(ch + 1);
```

выведет в метку "В" — символ, следующий за "Б". Т.е. в данном контексте ch воспримется как целое число, к которому добавляется 1. А затем результат сложения явным образом приводится к символьному типу.

Оператор

```
for (int i = 1; i < 256; i++)
  RichEdit1->Lines->Add(IntToStr(i)+' '+(char)i);
```

выведет в окно RichEditl строки вида "193 Б" для всех символов. Правда, видны будут только те символы, которые могут отображаться в окне RichEdit.

Помимо множества символов ANSI, используемого в Windows, иногда, в частности для консольных приложений DOS, требуется множество символов DOS (оно называется множеством OEM). В гл. 4, в разд. «CharToOem — перевод строки в текст DOS» и «OemToChar, OemToCharBuff— перевод текста DOS в строку» описаны функции, обеспечивающие взаимное преобразование этих двух множеств.

Символы типа **WideChar** - аналога стандартного для программ на C++ типа wchar_t, используют для своего хранения более одного байта. Последовательность символов соответствует множеству символов Unicode. В настоящее время символы WideChar и wchar_t в C++Builder используют 2 байта, но в дальнейшем число байтов может быть увеличено. Дело в том, что в Linux символ типа **wchar_t** занимает 4 байта. Впрочем, Linux и библиотеки GNU поддерживают и двухбайтовый стандарт Unicode, а также однобайтный.

Первые 256 символов совпадают в множествах Unicode и ANSI.

Тип **wchar_t** является естественным для Unicode и для технологии XML. Поэтому компоненты библиотеки CLX в C++Builder 6 работают в основном с этим типом. А компоненты VCL работают или с однобайтовыми символами, или с многобайтовыми символами MBCS. Это множество символов, используемое в азиатских языках - японском, китайском и т.п. Вряд ли подобная экзотика потребуется читателям данной книги. Но надо учитывать, что применение типа **wchar_t** при работе с компонентами VCL требует некоторых дополнительных затрат времени на трансляцию в MBCS.

2.5 Типы строк

2.5.1 Массивы символов

В языках C и C++ традиционно отсутствовал специальный тип строк. Впрочем, в стандартной библиотеке шаблонов STL имеется шаблон строк string, который будет рассмотрен в гл. 5. Но традиционно строки в этих языках рассматривались как массивы символов, оканчивающиеся нулевым символом ('\0'). Строка доступна через указатель на первый символ в строке. Значением строки является адрес ее первого символа. Таким образом, можно сказать, что в C++ строка является указателем — указателем на первый символ строки. В этом смысле строки подобны массивам, потому что массив тоже является указателем на свой первый элемент. Подробно о работе с массивами символов см. в разд. 2.11.1, посвященном массивам.

Строка может быть объявлена либо как массив символов, либо как переменная типа **char***. Каждое из двух приведенных ниже эквивалентных объявлений

```
char S[] = "строка";
char *Sp = "строка";
```

присваивает строковой переменной начальное значение «строка». Первое объявление создает массив из 7 элементов S содержащий символы 'с', 'т', 'р', 'о', 'к', 'а' и '\0'. Второе объявление создает переменную указатель Sp, который указывает на строку с текстом «строка», лежащую где-то в памяти. Но в любом случае число

хранимых символов на 1 больше числа значащих символов за счет оконечного нулевого символа.

Доступ к отдельным символам строки осуществляется по индексам, начинающимся с нуля. Например, **S[0]** и **Sp[0]** — первые символы объявленных выше строк, **S[l] и Sp[l]** — вторые и т.д.

В приведенных объявлениях длина строк определялась автоматически компилятором. Можно объявлять строковые переменные заданной длины. Например, оператор

```
char buff[100];
```

объявляет переменную **buff,** которая может содержать строку до 99 значащих символов плюс заключительный нулевой символ.

Для обработки строк имеется ряд библиотечных функций. Основные из них **strcat** — конкатенация (склеивание) двух строк, **strcmp** — сравнение двух строк, **strcpy** — копирование одной строки в другую, strstr — поиск в строке заданной подстроки, **strlen** — определение длины строки, **strupr** — преобразование символов строки к верхнему регистру, **sprintf** — построение строки по заданной строке форматирования и списку аргументов и ряд других функций. Все они подробно рассмотрены в гл. 3 и 4. А пока рассмотрим несколько примеров их применения.

Начнем с самого простого. Выше было приведено объявление массива символов **buff.** Как занести в него какой-то текст? Это можно сделать с помощью функции **strcpy:**

```
strcpy(buff,"Текст, копируемый в buff");
```

Эта функция копирует строку, являющуюся ее вторым параметром, в строку, являющуюся первым параметром, и возвращает указатель на результат копирования.

Теперь решим задачу посложнее. Пусть, например, мы хотим прибавить в конец текста строки S1 текст, хранящийся в строке S2. Это можно сделать с помощью функции **strcat:**

```
char S1[20] = "текст 1", S2[10] = "текст 2";
strcat(S1,S2);
```

Обратите внимание на то, что размер первой строки выбран с запасом, чтобы в ней уместились оба текста. Если не задать в объявлении размер строки, то он определится по присваиваемому ей тексту и в ней не останется места для каких-то добавлений.

Функция **strcat** прибавляет к тексту строки, указанной ее первым параметром, текст строки, указанной вторым параметром, и возвращает указатель на первую строку. Последнее обстоятельство позволяет делать вложенные вызовы **strcat,** если надо склеить несколько текстов. Давайте несколько усложним задачу. Пусть мы хотим оставить в неприкосновенности обе строки, а в третьей строке S хотим получить склеенные тексты строк S1 и S2, разделенные символом пробела. Это можно сделать следующими операторами:

```
char *S1 = "текст 1", *S2 = "текст 2", S[20];
strcat(strcat(strcat(S,S1)," "),S2);
```

Самый внутренний вызов **strcat** склеивает пустую строку S и строку S1. Он возвращает указатель на S и, значит, следующий вызов **strcat** склеивает текст, появившийся в S, со строковой константой, содержащей символ пробела. Функция **strcat** опять возвращает указатель на S и последний внешний вызов **strcat** добавляет к уже сформированной строке текст строки S2.

Приведенный код будет работать, если есть уверенность, что сначала текст в S отсутствует. Чтобы не зависеть от исходного текста в S, лучше вместо внутреннего вызова **strcat** применить функцию

```
strcpy(S,S1)
```

При анализе текстовых строк часто надо найти в одной из строк фрагмент текста, заданный в другой строке. Этот фрагмент, например, может быть некоторым ключевым словом, символом и т.п. Эту задачу позволяет решить функция

```
strstr(S1,S2)
```

которая ищет в строке S1 первое вхождение текста строки S2 и, если поиск прошел удачно, возвращает указатель на первый символ этого вхождения. Если же текст не был найден, возвращается нуль.

Теперь давайте решим более сложную задачу. Пусть нам надо найти в строке S1 первое вхождение текста строки S2 и, если поиск прошел удачно, то заменить найденный фрагмент на текст, содержащийся в строке S3. Иначе говоря, требуется произвести контекстную замену в S1 текста S2 на текст S3. Один из возможных вариантов решения этой задачи приведен ниже.

```
char S1[20], S2[20], S3[20], S[60], *St;
// операторы занесения текста в S1, S2, S3
...
St = strstr(S1,S2);
if (St)
{
 *St = 0;
 St += strlen(S2);
 Label1->Caption = strcat(strcat(strcpy(S,S1),S3),St);
}
else Label1->Caption = "Текст не найден";
```

Помимо строк S1, S2, S3 в этом коде объявлена строка S, являющаяся буфером, в который будет помещаться текст с произведенной в нем заменой. Объявлен также указатель на строку St. Он нам потребуется в качестве вспомогательной переменной.

Первый выполняемый оператор кода ищет с помощью функции strstr вхождение строки S2 в строку S1 и присваивает результат поиска переменной St. Если функция strstr вернула нуль (это эквивалентно false), то печатается сообщение "Текст не найден". Если же поиск прошел успешно, то осуществляются следующие операции. Сначала в символ, на который указывает St, засылается 0 — это эквивалентно нулевому символу. Таким образом выделяется первая часть строки S1, расположенная до заменяемого текста. Затем указатель St сдвигается на длину заменяемого текста, которая определяется функцией strlen. После этой операции St начинает указывать на первый символ в строке S1 после заменяемого текста. Следующий оператор формирует в буфере S текст с заменой и отображает его в метке Labell. Формирование текста осуществляется вложенными вызовами функций strcat и strcpy. Сначала срабатывает вложенный вызов strcpy. Он копирует в S строку, на которую указывает S1. Но поскольку вместо первого символа заменяемого текста мы занесли нулевой символ, то скопирована будет только начальная часть строки S1 до этого символа. Затем срабатывает вложенный вызов strcat и к тексту, сформированному в S, добавляется строка S3. Последний внешний вызов strcat добавляет к сформированному тексту часть строки S1, расположенную после замененного фрагмента. Именно на эту часть строки указывает St.

Чтобы это стало понятнее, разберем пример. Пусть строка S1 содержит текст "Маша ела кашу", строка S2 содержит текст "ела", а строка S3 - "съела". Значит, строка S1 представляет собой массив:

```
'М','а','ш','а',' ','е','л','а',' ','к','а','ш','у','\0'
```

После выполнения функции strstr указатель St будет указывать на шестой символ — букву 'е'. После того, как в этот символ заносится нуль, строка S1 имеет вид:

```
'М','а','ш','а',' ','\0','л','а',' ','к','а','ш','у','\0'
```

. После изменения **Pt** он начинает указывать на девятый символ — пробел после слова "ела". После вызова **strcpy** в строку S копируется первая часть строки S1, завершающаяся нулевым символом:

```
'М','а','ш','а',' ','\0'
```

После вложенного вызова **strcat** к строке S добавляется текст строки S3:

```
'М','а','ш','а',' ','с','ъ','е','л','а','\0'
```

И после внешнего вызова **strcat** к S добавляется строка, на которую указывает St, т.е. часть строки S1, начинающаяся с пробела после "ела":

```
М','а','ш','а',' ','с','ъ','е','л','а',' ','к','а','ш','у','\0'
```

В качестве последнего примера рассмотрим использование функции **sprintf.** Пусть в приложении имеется окно редактирования **Edit1,** в котором пользователь вводит фамилию сотрудника, и компонент **CSpinEditl** типа **TCSpinEdit,** в котором вводится год рождения. Вы хотите сформировать строку вида "Сотрудник ..., ... г.р.", в которой вместо точек должны подставляться введенные данные: фамилия и год. Это можно сделать следующим кодом:

```
#include <stdio.h>
char S[40];
sprintf(S,"Сотрудник %s, %i г.р.",Edit1->Text, CSpinEdit1->Value);
```

Первый аргумент функции **sprintf** — формируемая строка. Второй — строка форматирования (ее полное описание см. в гл. 3, в разд. 3.1.3.1). Она указывает текст формируемой строки и содержит спецификаторы, записываемые после символа "%", которые указывают формат включения в строку аргументов, список которых расположен в вызове **sprintf** после строки форматирования. В данном случае первый из этих параметров — текст в окне **Editl,** вводимый со спецификатором **%s,** что означает строку, а второй параметр — значение года в компоненте **CSpinEditl,** вводимое со спецификатором **%i,** что означает целое число.

Мы рассмотрели применение основных библиотечных функций работы со строками. Более полное изложение этих функций вы найдете в гл. 3 и 4.

C++Builder не ограничивается изложенным выше типичным для C++ подходом, сводящим строки к массивам символов. В нем реализованы в виде классов еще некоторые очень полезные типы. Наиболее интересные из них — **AnsiString,** имеющий множество методов и перегруженных операций, облегчающих работу со строками, и типы списков строк **TStrings и TStringList.** Кроме того, в стандартной библиотеке шаблонов STL имеется шаблон строк string, который будет рассмотрен в гл. 5.

2.5.2 Тип строк AnsiString

В C++Builder тип строк **AnsiString** реализован как класс, объявленный в файле **vcl/dstring.h** и аналогичный типу длинных строк в Delphi. Это строки с нулевым символом в конце. При объявлении переменные типа **AnsiString** инициализируются пустыми строками.

Для **AnsiString** определены операции отношения ==, !=, >, <, >=, <=. Сравнение производится с учетом регистра. Сравниваются коды символов, начиная с первого, и если очередные символы не одинаковы, строка, содержащая символ с меньшим кодом, считается меньше. Если все символы совпали, но одна строка длиннее и в ней имеются еще символы, то она считается больше, чем более короткая.

Для **AnsiString** определены операции присваивания =, += и операция склеивания строк (конкатенации) +. Определена также операция индексации []. Индексы начинаются с 1. Например, если S1 = "Привет", то **Sl[l]** вернет 'П', **S1[2]** вернет 'р' и т.д.

Класс **AnsiString** имеет множество методов, подробно рассмотренных в разд. **3.1.6. Не** останавливаясь сейчас на их перечислении, рассмотрим только некоторые примеры применения типа **AnsiString.**

Тип **AnsiString** используется для ряда свойств компонентов C++Builder. Например, для таких, как свойства **Text** окон редактирования, свойства **Caption** меток и разделов меню и т.д. Этот же тип используется для отображения отдельных строк в списках строк типа **TStrings.** Таким образом, постоянно имея дело с этими свойствами, вы постоянно работаете с **AnsiString.**

Рассмотрим некоторые примеры работы с **AnsiString.** Следующий оператор демонстрирует конкатенацию (склеивание) двух строк:

```
Label1->Caption = Edit1->Text + ' ' + Edit2->Text;
```

В данном случае в свойстве **Label1->Caption** отображается текст, введенный пользователем в окне редактирования **Edit1,** затем записывается символ пробела, а затем — текст, введенный в окне редактирования **Edit2.**

Как видите, склеивание строк типа **AnsiString** легко осуществляется перегруженной операцией сложения "+". Сравните это с теми вложенными вызовами функций **strcat,** которые приходилось делать в предыдущем разделе для тех же операций со строками типа **(char *),** и вы почувствуете преимущества **AnsiString.**

Теперь попробуем повторить рассмотренный в предыдущем разделе поиск в строке S1 фрагмента, заданного строкой S2, и замену его текстом строки S3. Код, осуществляющий эти операции, может иметь вид:

```
AnsiString S1, S2, S3;
// операторы занесения текста в S1, S2, S3
...
int i = S1.Pos(S2);
if (i)
  Label1->Caption = S1.SubString(1,i-1) + S3 +
                    S1.SubString(i+S2.Length(),255);
else Label1->Caption = "Текст не найден";
```

В этом коде использован ряд функций-элементов класса **AnsiString: Pos, Substring, Length.** Обратите внимание на то, что доступ к ним осуществляется операцией точка (.), вместо более привычной в C++Builder операции доступа к методам компонентов стрелка (->). Дело в том, что к методам компонентов доступ осуществляется через указатель на объект, а в данном случае к методам **AnsiString** доступ осуществляется через сами объекты — строки.

Первый выполняемый оператор приведенного кода использует функцию **Pos.** Эта функция ищет в строке, к которой она применена (в нашем случае в S1), первое вхождение подстроки, заданной ее параметром (в нашем случае S2). Если поиск успешный, функция возвращает индекс первого символа найденного вхождения подстроки. Индексы начинаются с 1. Если подстрока не найдена, возвращается 0.

Следующий оператор с помощью структуры **if...else** проверяет, не равно ли нулю **(false)** возвращенное функцией **Pos** значение. Если не равно, то производится формирование строки с заменой найденной подстроки. Строка формируется склеиванием трех строк: начальной части строки **S1,** расположенной до найденного вхождения подстроки, строки S3, заменяющей найденное вхождение, и заключительной части строки S1, расположенной после найденного вхождения. Для получения фрагментов строки S1 использована функция **Substring.** Эта функция возвращает подстроку, начинающуюся с символа в позиции, заданной первым параметром функции, и содержащую число символов, не превышающее значение, заданное вторым параметром функции. Таким образом, выражение **Sl.SubString(l, i — 1)** возвращает подстроку строки S1, начинающуюся с первого символа и содержащую i — 1 символов, т.е. часть строки S1, расположенную до найденного вхождения подстроки S2. Аналогично, выражение **S1.SubString(i +**

S2.Length(), **255)** возвращает подстроку строки S1, расположенную после найденного вхождения подстроки S2. При этом для определения начала этой подстроки использована функция **Length,** возвращающая число символов в строке (в нашем случае — в строке S2, содержащей заменяемый фрагмент). В приведенном выражении в качестве второго параметра функции **Substring** задано число 255, которое, как ожидается, превышает длину подстроки. В действительности будет возвращено менее 255 символов, столько, сколько имеется до завершающего S1 нулевого символа.

Сравнение данного кода с приведенным в предыдущем разделе для типов строк **(char *),** как мне кажется, показывает большую прозрачность действий со строками **AnsiString.**

Если нам надо не отображать измененную строку в виде сообщения, а просто произвести замену фрагмента в исходной строке S1, это еще более упрощает код, который в этом случае сводится всего к двум операторам:

```
int i = S1.Pos(S2);
S1 = S1.SubString(1,i-1) + S3 + S1.SubString(i+S2.Length(),255);
```

Подобная задача для строк **(char** *) была бы более сложной и потребовала бы объявления дополнительного буфера для временного хранения формируемой строки.

Давайте еще более усложним задачу: пусть в строке S1 надо заменить все вхождения S2 на строку S3. Эту задачу можно было бы решить следующим кодом:

```
int i0 = 0, i = S1.Pos(S2);
while(i)
{
 S1 = S1.SubString(1,i + iO - 1) + S3 +
      S1.SubString(i + iO + S2.Length(),255);
 iO += i - 1 + S3.Length();
 i = S1.SubString(i0 + 1, 255).Pos(S2);
}
```

Приведенный код мало отличается от рассмотренного ранее и не содержит каких-то новых функций. Основные отличия заключаются в следующем. Во-первых, вводится переменная **i0** — индекс, предшествующий первому символу еще не обработанной части строки **S1.** Значение **i0** изменяется после обработки очередной части строки. Во-вторых, очередное вхождение строки S2 в S1 определяется не по всей строке S1, а только по ее еще не обработанной части: **S1.SubString(i0 + 1, 255).**

Рассмотренную задачу контекстного поиска и замены в строке можно было бы решить иначе, воспользовавшись функциями **Delete и Insert** класса **AnsiString.** Функция **Delete** удаляет из строки, начиная с позиции, заданной первым параметром функции, число символов, заданное вторым параметром функции. Функция **Insert** вставляет в строку подстроку, заданную первым параметром функции, в позицию, заданную вторым параметром функции.

Применение этих функций позволяет выполнить контекстную замену с помощью, например, следующего кода:

```
int i0 = 1, i = S1.Pos(S2);
while(i > iO)
{
 S1.Delete(i,S2.Length());      // удаление вхождения S2
 S1.Insert(S3,i);               // вставка S3
 iO = i + S3.Length();
 i = iO - 1 + S1.SubString(i0, 255).Pos(S2);
}
```

Мы проиллюстрировали применение только малой части методов, имеющихся в классе **AnsiString.** Полный перечень этих методов вы найдете в соответствую-

щем разд. 3.1.6. В заключение отметим только метод, позволяющий переходить от типа **AnsiString** к типу **(char** *). Несмотря на то, что применение **AnsiString** практически всегда удобнее **(char** *), такие переходы приходится делать при передаче параметров в некоторые функции, требующие тип параметра **(char** *). Чаще всего это связано с вызовом функций API Windows или функций C++Builder, инкапсулирующих такие функции. Например, на протяжении этой книги многократно использовалась функция **Application->MessageBox,** требующая в качестве двух своих первых параметров (сообщения и заголовка окна) тип **(char** *). Аналогичные преобразования требуются для функции **PlaySound** для передачи в нее имени файла и для многих других функций.

Преобразование строки **AnsiString** в строку **(char** *) осуществляется функцией **c_str()** без параметров, возвращающей строку с нулевым символом в конце, содержащую текст той строки **AnsiString,** к которой она применена. Например, если вы имеете строки S1 и S2 типа **AnsiString,** которые хотите передать в функцию **Application->MessageBox** в качестве сообщения и заголовка окна, то вызов **Application->MessageBox** может иметь вид:

```
Application->MessageBox(S1.c_str(),S2.c_str(), MB_OK);
```

Возможно и обратное преобразование строки **(char** *) **в** строку **AnsiString.** Для этого используется функция

```
AnsiString(char *S)
```

которая возвращает строку типа **AnsiString,** содержащую текст, записанной в строке **S,** являющейся аргументом функции.

2.6 Перечислимые типы

Перечислимые типы определяют упорядоченное множество идентификаторов, представляющих собой возможные значения переменных этого типа. Вводятся эти типы для того, чтобы сделать код более понятным. В частности, многие типы C++Builder являются перечислимыми, что упрощает работу с ними, поскольку дает возможность работать не с абстрактными числами, а с осмысленными значениями.

Приведем пример, который покажет смысл введения пользователем своего перечислимого типа. Пусть, например, в программе должна быть переменная **Mode,** в которой зафиксирован один из возможных режимов работы приложения: чтение данных, их редактирование, запись данных. Можно, конечно, дать переменной **Mode** тип **int** и присваивать этой переменной в нужные моменты времени одно из трех условных чисел: 0 — режим чтения, 1 — режим редактирования, 2 — режим записи. Тогда программа будет содержать операторы вида

```
if (Mode == 1) ...
```

Через некоторое время уже забудется, что означает значение **Mode,** равное 1, и разбираться в таком коде будет очень сложно. А можно поступить иначе: определить переменную **Mode** как переменную перечислимого типа и обозначить ее возможные значения как **mRead, mEdit, mWrite.** Тогда приведенный выше оператор изменится следующим образом:

```
if (Mode == mEdit) ...
```

Конечно, такой оператор понятнее, чем предыдущий.

Переменные перечислимого типа могут определяться предложением вида:

```
enum {<константа 1>, ..., <константа n>} <имена переменных>;
```

Например

```
enum {mRead, mEdit, mWrite} Mode;
```

Этот оператор вводит именованные константы **mRead, mEdit, mWrite** и переменную **Mode,** которая может принимать значения этих констант. В момент объявления переменная инициализируется значением первой константы, в нашем примере — **mRead.** В дальнейшем вы можете присваивать ей любые допустимые значения. Например:

```
Mode = mEdit;
```

Значение переменной перечислимого типа можно проверять, сравнивая ее величину с возможными значениями. Кроме того, надо учитывать, что перечислимые типы относятся к целым порядковым типам и к ним применимы любые операции сравнения: >, < и т.п. Например, вы можете писать операторы:

```
if(Mode > mRead)  ...;
if(Mode < mWrite)  ...;
if(Mode == mEdit) ...;
```

Вы можете также использовать **Mode** в структуре **switch:**

```
switch (Mode)
 {
  case mRead:  ...
               break;
  case mEdit:  ...
               break;
  case mWrite: ...
}
```

По умолчанию перечислимые значения, указанные в объявлении **enum,** интерпретируются как целые числа, причем первое значение эквивалентно 0, второе — 1 и т.д. Именно эти значения рассматриваются в операциях отношения >, < и др. Значения по умолчанию можно изменить, если после имени константы указать знак равенства (=) и задать присваиваемое целое значение, как положительное, так и отрицательное. Например:

```
enum {mRead = -1, mEdit, mWrite = 2} Mode;
```

Если после каких-то констант не задано их целое значение, оно считается на 1 больше предыдущего. Поэтому в приведенном примере **mRead** эквивалентно **−1, mEdit** эквивалентно 0, **mWrite** эквивалентно 2.

После ключевого слова **enum** может следовать тег — имя объявляемого типа. Например:

```
enum regim {mRead = -1, mEdit, mWrite = 2} Mode, Model;
```

Этот оператор объявляет две переменные **Mode и Model** перечислимого типа, и кроме того определяет тип **regim.** В дальнейшем вы можете воспользоваться именем **regim** для объявления каких-то новых переменных, например:

```
regim Mode3;
```

2.7 Множества

Множество — это группа элементов, которая ассоциируется с ее именем и с которой можно сравнивать другие величины, чтобы определить, принадлежат ли они этому множеству. Как частный случай, множество может быть пустым.

Множество реализовано в C++Builder как шаблон класса, определенный в головном файле **vcl/sysdefs.h.**

Объявляется множество оператором:

```
Set <type, minval, maxval> переменные;
```

Параметр **type** определяет тип элементов множества. Обычно это порядковые типы **int, char** или перечислимый. Параметры **minval и maxval** типа **unsigned char** определяют минимальное и максимальное значения элементов множества. Минимальное значение должно быть не меньше 0, максимальное — не более 255.

Приведем примеры объявления множеств.

Объявление переменной **s1** как множества всех заглавных латинских букв имеет вид:

```
Set <char, 'A', 'Z'> s1;
```

Следующий оператор объявляет множество **Ch,** содержащее все символы:

```
Set <char, 0, 255> Ch;
```

Следующие операторы объявляют тип **UPPERCASESet** множества всех заглавных латинских букв и объявляют переменные s2 и s3 этого типа:

```
typedef Set <char, 'A','Z'> UPPERCASESet;
UPPERCASESet s1, s2;
```

Следующие операторы определяют множество S, элементами которого являются данные перечислимого типа E: **red, yellow, green:**

```
enum E { white, red, yellow, green };
Set <E, red, green> S;
```

Объявление переменной типа множества **Set** не инициализирует ее какими-то значениями. Инициализацию можно делать с помощью описанной ниже операции << — добавление элемента в множество.

Для множества определены следующие операции (в описании операций словами «данное множество» обозначается левый операнд):

Операция	Определение	Описание
–	Set___**fast**call operator –(**const** Set& **rhs**) const;	данное множество равно разности двух множеств: данного и **rhs** (операция **xor** с их элементами)
–=	Set&___fastcall operator –=(**const** Set& rhs);	создание нового множества, определенного разностью двух множеств: данного и rhs (операция **xor** с их элементами)
*	Set& fastcall operator *=(**const** Set& rhs);	создание нового множества, определенного пересечением двух множеств: данного и rhs (операция and с их элементами)
*–	Set___fastcall operator *(const Set& rhs) const;	данное множество равно пересечению двух множеств: данного и **rhs** (операция and с их элементами)
+	Set___fastcall operator +(const Set& **rhs**) const;	создание нового множества, определенного объединением двух множеств: данного и rhs (операция or с их элементами)
+=	Set&___fastcall operator +=(const Set& rhs);	данное множество равно объединению двух множеств: данного и **rhs** (операция or с их элементами)

Операция	Определение	Описание
<<	Set&__fastcall operator <<(const T el);	добавление элемента el в данное множество
<<	friend ostream& operator <<(ostream& os, const Set& arg);	поместить множество arg в поток ostream (выводится 0 или 1 для каждого элемента в зависимости от его наличия в множестве)
>>	Set&__fastcall operator >>(const T el);	удаление элемента el из данного множества
>>	friend istream& operator >> (istream& is, Set& arg);	извлечь множество arg из потока istream (вводится 0 или 1 для каждого элемента в зависимости от его наличия в множестве)
=	Set& __fastcall operator =(const Set& rhs);	присваивание данному множеству содержимого множества rhs
==	bool__fastcall operator ==(const Set& rhs) const;	эквивалентность двух множеств: данного и rhs (совпадение всех элементов)
!=	bool__fastcall operator !=(const Set& rhs) const ;	неэквивалентность двух множеств: данного и rhs

Все операции можно применять только к множествам одного типа, то есть к таким, при объявлении которых все аргументы объявления **(type, minval и maxval)** совпадают. В операциях, создающих новое множество (операции +, — и *), переменная, в которую заносится результат, также должна быть того же типа, что и операнды. Операция эквивалентности возвращает **true** в случае, когда оба операнда содержат только совпадающие элементы. Соответственно только в этом случае операция неэквивалентности возвращает **false.**

Для множеств Set определены также два метода:

Метод	Определение	Описание
Clear	**Set&__fastcall Clear();**	**очистка множества**
Contains	**bool__fastcall Contains(const T el) const;**	**проверка наличия в множестве элемента el**

Рассмотрим примеры работы с множествами. Пусть вы задаете пользователю в программе некоторый вопрос, подразумевающий ответ типа "Yes/No". Тогда возможные символы, вводимые пользователем в качестве ответа, являются множеством, содержащим символы "y", "Y", "n" и "N". Сформировать такое множество можно операторами:

```
Set <char, 0, 255> TrueKey;
...
TrueKey << 'y' << 'Y' << 'n' << 'N';
```

Тогда проверить, принадлежит ли введенный пользователем символ Key множеству допустимых ответов, можно с помощью метода **Contains:**

```
if (!TrueKey.Contains(Key))
   ShowMessge("Вы ввели ошибочный ответ");
else . . .
```

Рассмотрим еще один пример. Пусть вы хотите, чтобы в окне редактирования **Edit1** пользователь мог вводить только число, т.е. только цифры от 0 до 9. Это можно сделать, включив в обработчик события **OnKeyPress** этого окна операторы:

```
Set <char, '0', '9'> Dig;
Dig << '0' << '1' << '2' << '3' << '4' << '5'
    << '6' << '7' << '8' << '9';
if (!Dig.Contains(Key))
   {Key = 0; Beep();}
```

При попытке пользователя ввести символ, отличный от цифры, раздастся звук (его обеспечит функция **Beep)** и символ не появится в окне.

2.8 Указатели

2.8.1 Общие сведения

Указатель — это переменная, значение которой равно значению адреса памяти, по которому лежит значение некоторой другой переменной. В этом смысле имя этой другой переменной отсылает к ее значению *прямо,* а указатель — *косвенно.* Ссылка на значение посредством указателя называется *косвенной адресацией.*

Указатели, подобно любым другим переменным, перед своим использованием должны быть объявлены. Объявление указателя имеет вид:

```
type *ptr;
```

где type — один из предопределенных или определенных пользователем типов, а **ptr** — указатель. Читается это объявление так: «**ptr** является указателем на значение типа **type**».

Например,

```
int *countPtr, count;
```

объявляет переменную **countPtr** типа int * (т.е. указатель на целое число) и переменную **count** целого типа. Символ * в объявлении относится только к **countPtr.** Каждая переменная, объявляемая как указатель, должна иметь перед собой знак звездочки (*). Если в приведенном примере желательно, чтобы и переменная **count** была указателем, надо записать:

```
int *countPtr, *count;
```

Символ * в этих записях обозначает *операцию косвенной адресации.*

Может быть объявлен и указатель на **void:**

```
void *Pv;
```

Это универсальный указатель на любой тип данных. Но прежде, чем его использовать, ему надо в процессе работы присвоить значение указателя на какой-то конкретный тип данных. Например:

```
Pv = countPtr;
```

Указатели должны инициализироваться либо при своем объявлении, либо с помощью оператора присваивания. Указатель может получить в качестве начального значения О, **NULL** или адрес. Указатель с начальным значением 0 или **NULL** ни на что не указывает. **NULL** — это символическая константа, определенная специально для цели показать, что данный указатель ни на что не указывает. Пример объявления указателя с его инициализацией:

```
int *countPtr = NULL;
```

Как правило, при объявлении указателей им желательно присваивать значение **NULL** или определенный адрес. Учтите, что неинициализированный указатель может при работе с ним приводить к самым неожиданным результатам.

Для присваивания указателю адреса некоторой переменной используется *операция адресации* (&), которая возвращает адрес своего операнда. Например, если имеются объявления

```
int y = 5;
int *yPtr, x;
```

то оператор

```
yPtr = &y;
```

присваивает адрес переменной y указателю **yPtr.**

Для того чтобы получить значение, на которое указывает указатель, используется операция (*), обычно называемая *операцией косвенной адресации* или *операцией разыменования.* Она возвращает значение объекта, на который указывает ее операнд (т.е. указатель). Например, если продолжить приведенный выше пример, то оператор

```
x = *yPtr
```

присвоит переменной x значение *5,* т.е. значение переменной y, на которую указывает **yPtr.**

Операцию разыменования нельзя применять к указателю на **void,** поскольку для него неизвестно, какой размер памяти надо разыменовывать.

Массивы и указатели в C++ тесно связаны и могут быть использованы почти эквивалентно. Имя массива можно понимать как константный указатель на первый элемент массива. Его отличие от обычного указателя только в том, что его нельзя модифицировать.

Указатели можно использовать для выполнения любой операции, включая индексирование массива. Пусть вы сделали следующее объявление:

```
int b[5] = {1,2,3,4,5}, *Pt;
```

Тем самым вы объявили массив целых чисел **b[5]** и указатель на целое Pt. Поскольку имя массива является указателем на первый элемент массива, вы можете задать указателю Pt адрес первого элемента массива b с помощью оператора

```
Pt = b;
```

Это эквивалентно присваиванию адреса первого элемента массива следующим образом

```
Pt = &b[0];
```

Теперь можно сослаться на элемент массива **b[3]** с помощью выражения *(Pt + 3).

Указатели можно индексировать точно так же, как и массивы. Например, выражение **Pt[3]** ссылается на элемент массива **b[3].**

Таким образом манипуляции, определенные для массивов, определены и для указателей на массивы. Но с точки зрения понятности программы лучше это без крайней необходимости не использовать.

Указатели могут применяться как операнды в арифметических выражениях, выражениях присваивания и выражениях сравнения. Однако не все операции, обычно используемые в этих выражениях, разрешены применительно к переменным указателям.

С указателями может выполняться ограниченное количество арифметических операций. Указатель можно увеличивать (++), уменьшать (--), складывать с указателем целые числа (+ или +=), вычитать из него целые числа (— или -=) или вычитать один указатель из другого.

Сложение указателей с целыми числами отличается от обычной арифметики. Прибавить к указателю 1 означает сдвинуть его на число байтов, содержащихся в переменной, на которую он указывал. Обычно подобные операции применяются

к указателям на массивы. Если продолжить приведенный выше пример, в котором указателю **Pt** было присвоено значение b — указателя на первый элемент массива, то после выполнения оператора

```
Pt += 2;
```

Pt будет указывать на третий элемент массива b. Истинное же значение указателя **Pt** изменится на число байтов, занимаемых одним элементом массива, умноженное на 2. Например, если каждый элемент массива b занимает 2 байта, то значение **Pt** (т.е. адрес в памяти, на который указывает **Pt)** увеличится на 4.

Аналогичные правила действуют и при вычитании из указателя целого значения.

Переменные указатели можно вычитать один из другого. Например, если **Pt** указывает на первый элемент массива b, а указатель Pt1 — на третий, то результат выражения Ptl — Pt будет равен 2 — разности индексов элементов, на которые указывают эти указатели. И так будет, несмотря на то, что адреса, содержащиеся в этих указателях, различаются на 4 (если элемент массива занимает 2 байта).

Арифметика указателей теряет всякий смысл, если она выполняется не над указателями на массив. Нельзя полагать, чтоб две переменные одинакового типа хранятся в памяти вплотную друг к другу, если только они не соседствуют в массиве. Сравнение указателей операциями >, <, >=, <= также имеют смысл только для указателей на один и тот же массив. Однако операции отношения == и != имеют смысл для любых указателей. При этом указатели равны, если они указывают на один и тот же адрес в памяти.

Указатель можно присваивать другому указателю, если оба указателя имеют одинаковый тип. В противном случае нужно использовать операцию приведения типа, чтобы преобразовать значение указателя в правой части присваивания к типу указателя в левой части присваивания. Исключением из этого правила является указатель **на void (т.е.** void*), который является общим указателем, способным представлять указатели любого типа. Указателю на **void** можно присваивать все типы указателей без приведения типа. Однако указатель на **void** не может быть присвоен непосредственно указателю другого типа — указатель на **void** сначала должен быть приведен к типу соответствующего указателя.

Мы рассмотрели ранее указатели на массивы. Однако соотношение между массивами и указателями может быть и обратным — могут использоваться массивы указателей. Подобные структуры часто используются в массивах строк или в массивах указателей на различные объекты.

Например, вы можете сделать следующее объявление:

```
char *Sa[2] = ("Это первая строка", "Вторая");
```

Вы объявили массив размером 2 элементов типа **(char *).** Каждый элемент такого массива — строка. Но в C++ строка является, по существу, указателем на ее первый символ. Таким образом, каждый элемент в массиве строк в действительности является указателем на первый символ строки. Каждая строка хранится в памяти как строка, завершающаяся нулевым символом. Число символов в каждой из строк может быть различным. Таким образом, массив указателей на строки позволяет обеспечить доступ к строкам символов любой длины.

Указатели широко используются при передаче параметров в функции. Особенности использования указателей в этих целях см. в разд. 1.7.2.

2.8.2 Указатели на объекты классов

В C++Builder указатели используются очень широко. В частности, все компоненты, формы и т.д. объявляются именно как указатели на соответствующий объект, тип которого описан некоторым классом (см. разд. 2.14). Посмотрев заголовочный файл любого приложения, вы увидите в нем объявления вида:

```
TForm1 *Form1;
TLabel *Label1;
```

Это объявления указателей на форму **Form1** и на размещенные на ней компоненты **(Label1).** Можно создать и абстрактный указатель, не привязанный к какому-то конкретному объекту. Например, следующее объявление создает указатель на объект класса **TLabel** (метку):

```
TLabel *Lab;
```

Создается не сам объект, а только указатель на любой объект данного типа. В момент его создания указатель инициируется нулем. Нуль не может ассоциироваться ни с каким объектом в памяти. Поэтому определить, занесена в указатель ссылка на конкретный объект, или нет, можно, например, оператором:

```
if (Lab == 0) ... ;
```

Здесь многоточием обозначены некие действия, которые надо делать при отсутствии ссылки.

В C++ предопределена константа **NULL,** которая эквивалентна нулевому указателю. Поэтому приведенный выше оператор эквивалентен следующему:

```
if (Lab == NULL) ...;
```

В дальнейшем этому указателю Lab можно присвоить ссылку на любой объект соответствующего класса простым присваиванием. Например:

```
Lab = Label1;
```

Тогда указатель **Lab** становится как бы псевдонимом объекта **Label1.** Оба указателя: и **Lab,** и **Label1** ссылаются на один и тот же объект. Например, **Label1->Caption и Lab->Caption** ссылаются на надпись одной и той же метки.

В качестве типа объекта, на который ссылается указатель, можно задать **void.** Например:

```
void *Lab;
```

Такой указатель на **void** можно рассматривать как указатель на объект любого типа. В дальнейшем этому указателю можно задать простым присваиванием ссылку на объект любого типа. Но разыменование такого указателя требует применения явного приведения типов, поскольку компилятор не знает, на объект какого типа в действительности ссылается указатель. Поэтому для него нельзя, например, после присваивания ему ссылки на метку **Label1** (как в приведенном ранее примере) написать просто **Lab->Caption.** Для ссылки на надпись **Caption** через этот указатель надо писать **((TLabel *)Lab)->Caption,** то есть явным образом приводить тип указателя Lab к типу «указатель на объект класса **TLabel».**

Можно создавать ссылку на объект с помощью указателя не на истинный класс объекта, а на один из классов, которым наследует класс данного объекта. Дело в том, что любой объект может рассматриваться не только как объект своего класса, но и как объект любого класса-предка. Это в общем достаточно естественно для обычного понимания объектов в реальном мире. Так любой автомобиль может рассматриваться не только как объект автомобилей данной марки, например, «Жигули», но и как один из объектов более общих классов — автомобили, средства передвижения и т.д. Так же и объект в C++ может рассматриваться как объект любого из классов предков.

Например, универсальным указателем на любой компонент может быть указатель на класс **TControl** — базовый класс всех компонентов:

```
TControl *Contr;
```

Такому указателю можно непосредственно присваивать ссылку на любой компонент. Например,

```
Contr = Label1;
```

При таком присваивании компилятор сам производит необходимое приведение типов.

Но с этим указателем **Contr** уже нельзя работать непосредственно как с указателем на метку. Для него известны только свойства, объявленные в классе **TControl.** Это такие общие свойства всех компонентов, как, например, **Name** – имя. Непосредственная ссылка на специфические свойства классов — наследников невозможна. Например, попытка написать код **Contr->Caption** вызовет сообщение компилятора об ошибке с текстом: « 'Controls::TControl::Caption' is not accessible. », смысл которого заключается в том, что свойство **Caption** в классе **TControl** недоступно. Поэтому для доступа к методам и свойствам, отсутствующим в классе **TControl,** надо осуществлять явное приведение типа указателя, например:

```
((TLabel *)Contr)->Caption
```

Этот код как бы говорит компилятору: «Рассматривай Contr как ссылку на класс **TLabel**». И тогда никаких сообщений об ошибках не возникает.

Причина, по которой в ряде случаев для хранения ссылок на объекты используются указатели на объекты базовых классов, заключается в удобстве групповой обработки объектов разных классов с помощью общих для них методов или для задания значений общих для них свойств. Этот вопрос будет подробнее рассмотрен в следующем разделе.

2.8.3 Идентификация объекта неизвестного класса

В предыдущем разделе было рассмотрено объявление указателей на объекты и было показано, что тип такого указателя может определяться не обязательно классом конкретного объекта, но и любым классом-предком. В C++Builder это используется достаточно широко. Например, во все обработчики событий передается в качестве параметра **Sender** — указатель на объект, в котором произошло событие. Зачем нужен этот параметр? Конечно, если вы пишете обработчик какого-то события в конкретном компоненте, например, пишете для кнопки **Button1** обработчик события **OnClick,** которое наступает при щелчке на ней мыши, то параметр **Sender** вам не нужен. Вы и без этого параметра знаете, что событие произошло именно в кнопке **Button1.** Но часто для разных компонентов нужна идентичная реакция на идентичные события. В этих случаях писать отдельные одинаковые обработчики для разных компонентов нерационально. Можно ограничиться одним обработчиком для всех этих компонентов. Это обеспечит существенно более компактный код, его будет проще отлаживать, да и размер загрузочного модуля вашей программы будет меньше.

Во все обработчики событий в C++Builder передается по ссылке параметр **Sender,** объявленный как **TObject *Sender,** т.е. как указатель на объект типа **TObject.** Класс **TObject** является базовым классом всех компонентов в C++Builder. Но в нем не объявлено никаких свойств, которые можно было бы использовать в обработчике события. Поэтому при обращении к каким-то свойствам объектов вам надо явным образом осуществлять приведение типа параметра **Sender** к тому классу, в котором требуемые свойства объявлены. Пусть, например, вы пишете обработчик, который должен в качестве надписи (свойство **Caption)** метки **Labell** выводить текст: "Произошло событие в компоненте ...". Имя компонента, которое вам надо включать в эту надпись, содержится в свойстве **Name.** Но это свойство появляется только начиная с класса **TComponent.** Значит, именно к этому классу вам надо привести тип параметра **Sender** (о приведении типов см. в разд. 2.2). Тогда соответствующий оператор будет иметь вид:

```
Labell->Caption = "Произошло событие в компоненте " +
                  ((TComponent *)Sender)->Name;
```

Выражение **((TComponent *)Sender)** является приведением типа параметра **Sender** к типу указателя на объект класса **TComponent.** Только в этом классе и в его потомках появляется свойство **Name,** которое вам нужно. Приведенный оператор будет работать для любых компонентов: окон, меток, кнопок и т.д., поскольку классы всех компонентов являются производными от **TComponent.**

Аналогичное приведение типов можно осуществить с помощью описанной в разд. 2.2 операции **static_cast:**

```
Labell->Caption = " Произошло событие в компоненте " +
                  static_cast<TComponent *>(Sender)->Name;
```

Если описанное приведение типов требуется во многих операторах вашей функции, то для сокращения записи можно один раз определить указатель на объект **Sender** как указатель на требуемый класс, а затем во всех операторах использовать его. Это сделано, например, в следующем коде:

```
TComponent *0bj = (TComponent *)Sender;
...
Labell->Caption="Произошло событие в компоненте " + Obj->Name;
```

Первый из этих операторов объявляет переменную **Obj** как указатель на объект класса **TComponent** и с помощью явного приведения типа присваивает этому указателю ссылку на тот объект, на который указывает **Sender.** После этого переменную **Obj** можно везде использовать как указатель на этот объект.

Аналогично, если вы хотите применить к параметру **Sender** некоторый метод, объявленный в классах-наследниках, вы должны привести тип указателя к тому классу, где этот метод имеется. Например, если вы хотите увеличить масштаб оконного компонента, указателем на который является **Sender,** вы должны привести его тип к указателю на объект класса **TWinControl** (или одного из производных от него классов), так как только начиная с базового класса всех оконных компонентов **TWinControl** объявлен требуемый вам метод **ScaleBy.** Соответствующий оператор будет иметь вид:

```
((TWinControl *)Sender)->ScaleBy(11,10);
```

В ряде случаев требуется определить истинный класс объекта, на который указывает параметр **Sender.** Это можно сделать с помощью метода **ClassName,** объявленного в классе **TObject как**

```
ShortString__fastcall ClassName();
```

Функция **ClassName** возвращает строку типа **ShortString,** содержащую истинный класс объекта. Например, оператор

```
Labell->Caption = Sender->ClassName();
```

может выдать текст "TButton", если **Sender** указывает на кнопку типа TButton.

Функцию **ClassName** можно использовать для выполнения каких-то действий с объектами только одного конкретного класса. Например, оператор

```
if (String(Sender->ClassName()) == "TLabel")
   ...;
```

обеспечивает выполнение неких действий (обозначенных многоточием) только для объектов класса **TLabel.**

Для тех же целей может использоваться еще одна функция — **ClassNameIs,** объявленная в классе **TObject как:**

```
bool__fastcall ClassNameIs(const AnsiString string);
```

Эта функция возвращает **true,** если класс объекта совпадает с заданным параметром **string.** При использовании этой функции приведенный выше пример приобретает вид:

```
if (Sender->ClassNameIs("TLabel"))
   ...;
```

В приведенных ранее примерах использования свойств и методов класса, к которому приводится указатель на класс-предшественник, вас может подстерегать некая опасность. Выше был приведен пример масштабирования компонента с использованием метода **ScaleBy,** объявленного в классе **TWinControl.** Но если истинный класс объекта, на который указывает **Sender,** окажется не потомком класса **TWinControl** (например, меткой **TLabel,** которая не наследует **TWinControl),** то метод **ScaleBy** не сработает. Еще более неприятные и непредсказуемые результаты получатся, если вы обратитесь к свойству, отсутствующему у компонента.

Поэтому, если нет уверенности, что применяемый метод или свойство имеется в обрабатываемом объекте, надо предварительно проверить, является ли класс объекта потомком того класса, в котором требуемый метод или свойство объявлены. Например, прежде, чем применять метод **ScaleBy,** надо убедиться, что класс объекта является потомком **TWinControl.**

Для этих целей можно воспользоваться методом **InheritsFrom,** объявленным в классе **TObject** и, следовательно, имеющимся в любых компонентах. Объявление этого метода:

```
bool __fastcall InheritsFrom(TClass aClass);
```

Метод возвращает **true,** если класс данного объекта является потомком класса aClass, указываемого как параметр метода. Этот параметр имеет тип **TClass,** который может создаваться операцией **__classid:**

```
__classid(classType)
```

Аргументом этой операции является обычное имя класса, например, **TWinControl.**

Таким образом, проверка, является ли класс объекта, на который указывает **Sender,** потомком **TWinControl,** может осуществляться оператором:

```
if (Sender->InheritsFrom(__classid(TWinControl)))
      ...;
```

С учетом этого приведенный ранее пример масштабирования методом **ScaleBy** оконных компонентов более грамотно должен осуществляться следующим оператором:

```
if (Sender->InheritsFrom(__classid(TWinControl)))
   ((TWinControl *)Sender)->ScaleBy(11,10);
```

Еще одна функция — **ClassParent,** объявленная в классе **TObject,** возвращает класс, являющийся непосредственным предком класса данного объекта. Функция объявлена как:

```
TClass __fastcall ClassParent()
```

Если данный класс не имеет предшественников (т.е. это класс **TObject),** то возвращается **NULL.**

Функция **ClassParent,** используемая в цикле, позволяет восстановить всю иерархию класса объекта. Следующий код заносит в список строки, перечисляющие все классы, встречающиеся на пути по дереву классов от **TObject** до класса, на который указывает параметр **Sender.**

```
TClass ClassRef= Sender->ClassType();
ListBox1->Clear();
while(ClassRef != NULL)
 {
   ListBox1->Items->Add(ClassRef->ClassName());
   ClassRef = ClassRef->ClassParent();
 }
```

Так, если **Sender** указывает на объект типа **TButton,** то в списке **ListBoxl** окажется текст:

```
TButton
TButtonControl
TWinControl
TControl
TComponent
TPersistent
TObject
```

2.9 Ссылки

Ссылки — это специальный тип указателя, который позволяет работать с указателем как с объектом. Объявление ссылки делается с помощью операции ссылки, обозначаемой амперсандом (&) — тем же символом, который используется для адресации. Если в вашей программе имеется указатель на объект какого-то типа **MyObject:**

```
MyObject *P = new MyObject;
```

то вы можете создать ссылку на этот объект оператором:

```
MyObject & Ref = *P;
```

Объявленная таким образом переменная **Ref** является ссылкой на объект **MyObject.** Она может рассматриваться как псевдоним объекта. Эта переменная реально является указателем, а не самим объектом. Но работа с ней производится как с объектом. Например, если вы хотите получить доступ к некоторому свойству объекта x, то через указатель на объект вы обеспечиваете доступ выражением **P->x,** т.е. через операцию стрелка. А через ссылку вы обеспечиваете доступ к свойству x выражением **Ref.x,** т.е. через операцию точка.

Аналогичным образом вы можете получить доступ по ссылке и к любым компонентам. Например, если в вашем приложении имеется метка **Labell,** то вы может обращаться к его свойству **Caption** оператором

```
Label1->Caption = "Это обращение по указателю";
```

А можете ввести соответствующую ссылку и обращаться через нее:

```
TLabel & ref = *Label1;
ref.Caption = "Это обращение по ссылке";
```

Чаще всего ссылки используются при передаче в функции параметров по ссылке. Этот вопрос подробно рассмотрен в разд. 1.7.2.

2.10 Файлы и потоки

Работа с файлами в C++Builder может производиться несколькими принципиально различными (с точки зрения пользователя) способами:

- использование библиотечных компонентов
- работа с файлами как с потоками в стиле C
- работа с файлами как с потоками в стиле C++

Рассмотрим эти возможности.

2.10.1 Файловый ввод/вывод с помощью компонентов

Работа с текстовыми файлами может осуществляться с помощью методов **LoadFromFile и SaveToFile,** имеющихся у классов TStrings и **TStringList.** Эти

классы описывают списки строк и обладают множеством методов, позволяющих манипулировать строками.

Если вы хотите в своем приложении прочитать содержимое некоторого текстового файла, обработать текст и сохранить его в файле, вы можете сделать это следующим образом. Объявите и создайте две глобальные переменные: список типа **TStringList,** в котором будет храниться текст файла, и строковую переменную типа **AnsiString,** в которой можете сформировать имя файла. Например:

```
TStringList *List = new TStringList;
AnsiString SFile = "Test.txt";
```

Не забудьте только, что если требуемый файл расположен не в текущем каталоге и вам надо указать путь к файлу, то обратные слэши в записи пути должны быть сдвоенные (см. разд. 1.5.1). Например, если вам требуется файл "c:\MyTest\Test.txt", то вы должны записать его как "c:\\MyTest\\Test.txt".

В момент, когда вы хотите загрузить в свой список файл, надо выполнить оператор

```
List->LoadFromFile(SFile);
```

Впрочем, ограничиться таким оператором можно, если есть уверенность, что требуемый файл существует. В противном случае код надо несколько усложнить, чтобы можно было перехватить сгенерированное исключение. Например:

```
try{
 List->LoadFromFile(SFile);
}
catch(...){
 ShowMessage("Файл \"" + SFile +"\" не найден");
}
```

Если файл нормально загрузился в список **List,** вы можете работать с его текстом. Текст расположен в свойстве списка **Strings[int Index],** в котором каждая строка имеет тип AnsiString. Индексы начинаются с нуля. Для нашего примера **List->Strings[0]** — это первая строка, **List->Strings[l]** — вторая и т.д.

Для списков типа **TStringList** предусмотрено множество методов. При обработке отдельных строк вы можете использовать операции и методы, предусмотренные для строк типа **AnsiString** (см. разд. 2.5.2 и соответствующие разделы гл. 3).

Если вы хотите сохранить файл после проведенного редактирования, можно выполнить оператор

```
List->SaveToFile(SFile);
```

где SFile содержит прежнее или новое имя файла.

При открытии и сохранении файла вы можете воспользоваться стандартными диалогами Windows, вызываемыми через соответствующие компоненты C++Builder.

Если вы открываете файл для того, чтобы пользователь мог его просмотреть, что-то в нем отредактировать и сохранить, вы можете обойтись без описанного выше объекта типа **TStringList.** Для этих целей проще воспользоваться многострочными окнами редактирования типов **TMemo** или **TRichEdit.** В последнем случае вы можете работать не только с обычными текстовыми файлами, но и с файлами в обогащенном формате RTF. Свойства **Lines** этих компонентов имеют тип **TStrings,** что позволяет применять к ним непосредственно методы **LoadFromFile и SaveToFile.** Например:

```
Memo1->Lines->LoadFromFile(SFile);
RichEdit1->Lines->LoadFromFile(SFile);
```

Через компоненты C++Builder можно работать не только с текстовыми файлами, но и с файлами изображений и мультимедиа.

2.10.2 Файловый ввод/вывод с помощью потоков в стиле С

2.10.2.1 Общие сведения

В языках С и С++ файл рассматривается как поток (stream), представляющий собой последовательность считываемых или записываемых байтов. При этом поток «не знает», что и в какой последовательности в него записано. Расшифровка смысла записанных последовательностей байтов лежит на программе.

Классический подход, принятый в С, заключается в том, что информация о потоке (файле) заносится в структуру типа **FILE,** определенную в файле **stdio.h.** Файл открывается с помощью функции **fopen,** которая возвращает указатель на структуру типа FILE. Этот указатель потока используется далее во всех операциях с файлами.

Синтаксис функции **fopen:**

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Функция **fopen** открывает файл с именем в виде строки, на которую ссылается указатель **filename,** и связывает с ним поток. Аргумент **mode** указывает на строку, которая определяет режим открытия. Она может содержать спецификаторы:

r	открыть файл только для чтения
r+	открыть существующий файл для чтения и записи
a	открыть или создать файл для записи данных в конец файла
a+	открыть или создать файл для чтения или записи в конец файла
w	создать файл для записи
w+	создать файл для чтения и записи

К указанным спецификаторам в конце или перед символом "+" может добавляться символ "t" — текстовый файл, или "b" — бинарный, двоичный файл. Например, **rt, rb,** r+t, r+b и т.д. Если ни символ "t", ни символ "b" не указаны, то тип открываемого файла определяется значением глобальной переменной **_fmode,** определенной в файле **fcntl.h.** Она может принимать значения O_TEXT — текстовый файл (по умолчанию) или **O_BINARY** — двоичный файл. Более подробное пояснение режимов открытия файлов вы найдете в гл. 3, в разд. 3.5.2.

Открываемый функцией **fopen** поток буферизуется, т.е. обмен информацией происходит не непосредственно с файлом, "а с промежуточным буфером, расположенным в оперативной памяти. Информация переписывается из буфера в файл только при переполнении буфера или при закрытии файла. В гл. 3, в разд. 3.5.2 вы можете посмотреть функции, управляющие процессом буферизации.

Функция **fopen** возвращает указатель на объект, управляющий потоком. Если попытка открыть файл закончилась неудачей, **fopen** возвращает нулевой указатель.

После того, как необходимая работа с файлом (чтение или запись) завершена, файл должен быть закрыт функцией **fclose(FILE *),** в которую передается указатель потока.

2.10.2.2 Текстовые файлы

Рассмотрим сначала работу с текстовыми файлами. Открытие текстового файла «Test.txt» может иметь вид:

```
#include <stdio.h>
...
FILE *F;
if ((F = fopen("Test.txt", "rt")) == NOLL)
{
 ShowMessage("Файл не удается открыть");
 return;
}
...          // чтение из файла
fclose(F);   // закрытие файла
```

Здесь объявляется переменная F — указатель потока и связывается с файлом «Test.txt», открываемым как текстовый только для чтения. Если открыть файл не удалось (например, он не существует), появляется сообщение об ошибке.

Из открытого таким образом файла можно читать информацию. После окончания чтения файл должен быть закрыт функцией fclose(F).

Если бы файл открывался функцией

```
fopen("Test.txt", "rt+")
```

то из такого файла можно было бы не только читать информацию, но и записывать в него новые строки.

Из текстового файла можно читать информацию по строкам или по символам. Чтение строки осуществляется функцией **fgets**:

```
char *fgets(char *s, int n, FILE *stream);
```

В вызове функции s — указатель на буфер, в который читается строка, n – число читаемых символов. Чтение символов в строку происходит или до появления символа конца строки "\n" (этот символ записывается в строку), или читается n — 1 символ. В конце прочитанной строки записывается нулевой символ.

Например, чтение и отображение в компоненте **Memo1** всех строк файла может быть организовано следующим образом:

```
char s[80];
Memo1->Clear();
do
{
 fgets(s,80,F);
 if(feof(F)) break;
 if(s[strlen(s)-1] == '\n') s[strlen(s)-1] = 0;
 Memo1->Lines->Add(s);
}
while(true);
fclose(F);   // закрытие файла
```

Функция fgets читает очередную строку. Функция feof проверяет, не прочитан ли символ конца файла. При чтении этого символа feof возвращает ненулевое значение и цикл прерывается. Если признака конца файла нет, то оператор

```
if(s[strlen(s)-1] == '\n') s[strlen(s)-1] = 0;
```

убирает из строки последний символ, если он оказывается символом перевода строки. Эта операция не обязательна, но наличие символа "\n" испортит вид строки в окне Memo1. Затем прочитанная строка заносится в окно редактирования.

Чтение из текстового файла форматированных данных может осуществляться функцией **fscanf**.

```
int fscanf(FILE *stream, const char *format[, address, ...]);
```

Ее параметр format определяет строку форматирования аргументов, заданных своими адресами. Подробно строка форматирования рассмотрена в гл. 3, в разд. 3.1.3.2. Сейчас отметим только, что эта строка при чтении обычно состоит из последовательности символов "%", после которых следует символ типа читаемых

данных. Ниже приведены некоторые наиболее часто используемые символы (подробнее см. в разд. 3.1.3.2):

Символ	Вводимое значение	Тип аргумента функции
i	Десятичное, восьмеричное или шестнадцатеричное целое	int *arg
I	Десятичное, восьмеричное или шестнадцатеричное целое	long *arg
d	Десятичное целое	int *arg
D	Десятичное целое	long *arg
u	Десятичное целое без знака	unsigned int *arg
U	Десятичное целое без знака	unsigned long *arg
e, E	Действительное с плавающей запятой	float *arg
s	Строка символов	char arg[]
c	Символ	char *arg

Перед символом типа могут добавляться модификаторы. В частности, модификатор l расширяет тип целого до **long int,** а тип действительного до **double.**

Пусть, например, вы знаете, что, начиная с текущей позиции файла, в нем записаны, разделенные пробелами, два целых и одно действительное число. Тогда прочитать эти числа можно операторами:

```
int i1, i2;
double r;
fscanf(F, "%d%d%le", &i1, &i2, &r);
```

Обратите внимание, что в качестве аргументов, в которые заносятся читаемые функцией **fscanf** данные, всегда указываются адреса переменных, а не сами переменные. Отсутствие операции адресации (&) — очень распространенная ошибка, которая приводит к самым неожиданным результатам.

При форматированном чтении могут возникать ошибки из-за достижения конца файла или из-за неверного формата записанного в файле числа. Проверить, успешно ли прошло чтение, можно по значению, возвращаемому функцией **fscanf.** При успешном чтении она возвращает число прочитанных полей. Поэтому в нашем примере лучше организовать чтение следующим образом:

```
if (fscanf(F, "%d%d%le", &i1, &i2, &r) != 3)
{
 ShowMessage("Ошибка чтения");
 ...
}
```

Символ типа s позволяет читать из файла отдельное слово, точнее — так называемую лексему -- последовательность символов, завершающуюся пробельным символом. Пусть, например, мы хотим просмотреть файл, чтобы узнать, не встречается ли в нем слово, которое пользователь ввел в окне редактирования **Edit1.** Для решения этой задачи после того, как файл открыт, можно выполнить, например, следующий код:

```
char s[80], key[10];
strcpy(key,Edit1->Text.c_str()); // загрузка ключевой строки
do
{
 fscanf(F, "%s", &s);
```

```
 if(feof(F)|| !strcmp(s,key)) break;
}
while(true);
fclose(F);
if  (!strcmp(s,key))
  ShowMessage("Слово найдено");
```

В этом коде вводится рабочая строка s и строка key, *в* которую функцией **strcpy** загружается ключевое слово. Далее в цикле функцией **fscanf** в строку s читается по одной лексеме из файла. Функция **strcmp** сравнивает эту лексему с ключом. Она возвращает 0, если строки s и key совпадают. В этом случае, а также при достижении конца файла цикл прерывается.

Мы рассмотрели вопросы чтения из текстового файла. Имеется также ряд функций записи в текстовый файл. Наиболее часто используемая из них — функция **fprintf:**

```
int fprintf(FILE *stream, const char *format[, argument, ...]);
```

Эта функция подобна рассмотренной выше функции **fscanf,** только строка форматирования строится несколько иначе. В ней используются аналогичные рассмотренным ранее символы типа, помещаемые после символа "%". Но имеется много возможностей по выбору формата печати данных в файл. Кроме того, все символы строки форматирования, не предваряемые символом "%", просто помещаются в выходной поток. Подробнее о функции **fprintf** и других функциях записи вы можете посмотреть в гл. 3 и 4. А пока приведем только короткий пример.

Пусть у вас имеется строка s типа **(char** *), содержащая фамилию сотрудника, и целое число **year,** содержащее год его рождения. Вы хотите создать текстовый файл и занести в него запись, первая строка которой содержит слово «ХАРАКТЕРИСТИКА», а вторая — текст «сотрудник ..., ... г.р.». Вместо точек в этом тексте подразумевается фамилия и год рождения. Это можно сделать следующим кодом:

```
FILE *F;
if ((F = fopen("Test.txt", "wt")) == NULL)
{
 ShowMessage("Файл не удается создать");
 return;
}
char S[40];
int year = 1960;
strcpy(s,"Иванов");
fprintf(F, "ХАРАКТЕРИСТИКА\nсотрудник %s, %i г.р.\n", &S, year);
fclose(F);
```

Файл открывается функцией **fopen** как текстовый файл для записи. Если файла с указанным именем не было, он создается. Если такой файл был, все его содержимое уничтожается. Затем функцией **fprintf** в файл записывается требуемый текст. В строке форматирования этой функции записан текст первой строки, затем указан символ перехода на новую строку "\n"; далее следует начало второй строки — "сотрудник ", затем символы "%s", задающие тип первого аргумента — указателя на S, затем символы "%i", задающие тип второго аргумента — числа **year,** и наконец — заключительная часть второй строки. В результате в файл будут записаны строки:

```
ХАРАКТЕРИСТИКА
сотрудник Иванов, 1960 г.р.
```

Рассмотренными функциями не ограничиваются возможности работы с текстовыми файлами. Подробнее все эти функции вы можете посмотреть в гл. 3 и 4.

2.10.2.3 Двоичные файлы

Двоичный файл представляет собой просто последовательность символов, в которой без каких-либо разделителей — пробелов, символов конца строки и т.п. хранятся символы, отображающие самые различные объекты. Они совпадают с тем, как хранятся соответствующие объекты в оперативной памяти. Что именно и в какой последовательности лежит в двоичном файле — должна знать программа.

Двоичные файлы имеют немало преимуществ перед текстовыми при хранении каких-то числовых данных. Операции чтения и записи с такими файлами производятся намного быстрее, чем с текстовыми, поскольку отсутствует необходимость форматирования: перевода в текстовое представление и обратно. Двоичные файлы, как правило, имеют существенно меньший объем, чем аналогичные текстовые файлы. В двоичных файлах вы можете перемещаться в любую позицию и читать или записывать данные в произвольной последовательности, в то время как в текстовых фалах практически всегда производится последовательная обработка информации. Пожалуй, недостаток двоичного файла с точки зрения программиста только один — просматривая его с помощью какого-то текстового редактора, трудно понять, где что в нем находится, и это в ряде случаев затрудняет отладку.

О том, как открываются двоичные файлы, уже рассказывалось в разд. 2.10.2.1. Запись и чтение в двоичные файлы чаще всего производятся соответственно функциями fwrite и **fread**:

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

В обе функции передается указатель ptr на выводимые или вводимые данные. Параметр size задает размер в байтах передаваемых данных, а параметр n определяет число передаваемых данных. Применение этих функций иллюстрируется приведенным ниже примером.

```
int i = 1, j = 25, i1, j1;
double a = 25e6, a1;
char s[10], s1[10];
strcpy(s,"Иванов");

FILE *F;
// запись в файл
if ( (F = fopen("Test.dat", "wb")) == NULL)
{
 ShowMessage("Файл не удается создать");
 return;
}
fwrite(&i,sizeof(int),1,F);              // запись i
fwrite(&j,sizeof(int),1,F);              // запись j
fwrite(&a,sizeof(double),1,F);           // запись a
fwrite(s,sizeof(char),strlen(s)+1,F); // запись строки s
fclose (F);

// чтение из файла
if ((F = fopen ("Test.dat", "rb")) == NULL)
{
 ShowMessage("Файл не удается открыть");
 return;
}
fread(&i1,sizeof(int),1,F);              // чтение i
fread(&j1,sizeof(int),1,F);              // чтение j
fread(&a1,sizeof(double),1,F);           // чтение a
fread(s1,sizeof(char),strlen(s)+1,F);  // чтение строки s
fclose (F);
```

В данном примере создается двоичный файл "Test.dat" и в него записывается два целых числа i и j, действительное число a и строка s. Затем этот файл закрывается, открывается для чтения, и данные из него читаются в переменные **i1, j1, a1** и **s1.**

В отношении записи и чтения чисел, вероятно, все понятно. А вопрос записи и чтения строк имеет смысл обсудить подробнее.

В приведенном примере запись строки производится оператором

```
fwrite(s,sizeof(char),strlen(s)+1,F); // запись строки s
```

Запись ведется по символам и указано число записываемых символов — **strlen(s)+1** (единица добавляется на нулевой символ в конце). Читается строка аналогично:

```
fread(s1,sizeof(char),strlen(s)+1,F); // чтение строки s
```

При этом чтение тоже идет по символам и читается **strlen(s)+1** символов.

Тут внимательный читатель может увидеть некоторую подтасовку. В данном учебном примере мы знаем длину строки, которую записали в файл, и можем прочитать требуемое число символов. Но как быть в реальных задачах, когда мы, скорее всего, не будем знать длину записанной строки? Эту проблему можно решить несколькими путями. Проще всего записывать и читать весь массив символов как единое целое:

```
fwrite(s,sizeof(s),1,F);
fread(s1,sizeof(s),1,F);
```

Этот путь **простой,** но имеет один **недостаток:** записывается всегда весь массив символов s, даже если содержащаяся в нем строка много короче размера массива. Приведенные операторы в нашем примере эквивалентны операторам

```
fwrite(s,sizeof(char)*10,1,F);
fread(s1,sizeof(char)*10,1,F);
```

Таким образом, при частичном заполнении массива в файле будут храниться лишние байты. Если в файле много строк разной длины, а все они будут храниться как максимальная из них, то размер файла будет значительно больше действительно необходимого.

Другой путь — записывать по-прежнему по символам, но при чтении проверять каждый символ, чтобы при появлении нулевого символа закончить чтение строки. Это может быть реализовано следующим образом:

```
// запись строки
fwrite(s,sizeof(char),strlen(s)+1,F);
...
// чтение строки
for(int ind = 0; ind < 10; ind++)
{
 fread(s1+ind,sizeof(char),1,F);
 if(s1[ind] == '\0') break;
}
```

Здесь в цикле for читается за раз по одному символу и при обнаружении нулевого символа цикл прерывается. Обратите внимание, что адрес чтения очередного символа в данном случае задается выражением **sl+ind.** Нельзя было бы вместо этого использовать выражение **s1[ind],** так как функция **fread** требует указания именно адреса, а не значения переменной, в которую осуществляется чтение.

Функцию **fread** в этом примере можно было бы заменить на **fgetc,** которая читает один символ из потока:

```
s1[ind] = fgetc(F);
```

И, наконец, еще один вариант чтения строк неизвестной длины из двоичного файла. Можно перед строкой записывать в файл целое число, равное числу символов в строке. Тогда чтение строки не встретит затруднений:

```
// запись строки
int it = strlen(s)+1;
fwrite(&it,sizeof(int),1,F);
fwrite(s,sizeof(char),it,F);
...
// чтение строки
fread(&it,sizeof(int),1,F);
fread(s1,sizeof(char),it,F);
```

В приведенных примерах чтение происходило последовательно. Но, работая с двоичными файлами, можно организовать произвольное чтение данных. Для этого служит указатель (курсор) файла, который определяет текущую позицию в файле для чтения и записи. При чтении или записи указатель автоматически смещается на число обработанных байтов. Узнать позицию указателя можно функцией **ftell,** которая возвращает текущую позицию:

```
long int ftell(FILE *stream);
```

Изменить позицию указателя можно функцией **fseek:**

```
int fseek(FILE *stream, long offset, int whence);
```

Эта функция задает сдвиг на число байтов **offset** относительно точки отсчета, определяемой параметром **whence.** Параметр **whence** может принимать значения:

Константа	whence	Точка отсчета
SEEK_SET	**0**	**Начало файла**
SEEK_CUR	**1**	**Текущая позиция**
SEEK_END	***2***	**Конец файла**

Если задано значение **whence = 1, то offset** может быть положительным (сдвиг вперед) или отрицательным (сдвиг назад).

Функция **rewind** перемещает указатель на начало файла (позиция 0). Впрочем, то же самое можно сделать оператором

```
fseek(F, OL, 0);
```

Возможность перемещать указатель особенно полезна в файлах, которые состоят из однородных записей одинакового размера. Например, если в файле записаны только действительные числа типа **double,** то для того, чтобы прочитать i-ое число, достаточно выполнить операторы

```
fseek(F, sizeof(double)*(i-1), 0);
fread(&a,sizeof(double),1,F);
```

Таким образом можно читать любые записи в любой последовательности.

С помощью перемещения указателя можно редактировать записи в файле. Пусть, например, вы хотите одно из чисел, записанных в файле, изменить, умножив его на 10. Это можно сделать, если открыть файл в режиме чтения и записи (например, "rb+"), установить позицию, соответствующую изменяемому числу, и выполнить операторы:

```
fread(&a,sizeof(double),1,F);
a *= 10;
fseek(F,-sizeof(double),1);
fwrite(&a,sizeof(double),1,F);
```

Первый из этих операторов читает число в переменную a, второй — умножает его на 10. Третий оператор возвращает текущую позицию на одну запись назад, поскольку после выполнения **fread** позиция сдвинулась вперед. Последний оператор пишет в ту позицию, в которой было прочитано число, новое значение.

Ту же задачу можно решить иначе:

```
long int pos = ftell(F);              // запоминание позиции
fread(&a,sizeof(double),1,F);
a *= 10;
fseek(F,pos,0);                       // восстановление позиции
fwrite(&a,sizeof(double),1,F);
```

Здесь функция **ftell** запоминает позицию, из которой читается число, а функция fseek восстанавливает эту позицию перед записью измененного числа.

С помощью двоичных файлов можно записывать и читать не только числа и строки, но и гораздо более сложные объекты, например, структуры. Ниже приведен пример, в котором определяется тип структуры **spers,** объявляются переменные этого типа **pers и pers1,** поля структуры **pers** заполняются, а затем она целиком записывается в файл. После того, как файл закрывается, он открывается для чтения, и данные из него читаются в структуру **persl.**

```
struct spers
{
  char Name[20];
  int year;
};

struct spers pers, pers1;
strcpy(pers.Name,"Иванов");
pers.year = 1960;

FILE *F;
if ((F = fopen("Test2.dat", "wb")) == NULL)
{
 ShowMessage("Файл не удается создать");
 return;
}
fwrite(&pers,sizeof(spers),1,F);
fclose(F);

if ((F = fopen("Test2.dat", "rb")) == NULL)
{
 ShowMessage("Файл не удается открыть");
 return;
}
fread(&pers1,sizeof(spers),1,F);
fclose(F);
```

2.10.2.4 Ввод/вывод, использующий дескрипторы потоков

В С предусмотрен еще один механизм работы с файлами, основанный не на указателях на структуру типа FILE, а на дескрипторах. Файлы, открываемые подобным образом, не работают с буферами и с форматированными данными.

В начале работы любой программы автоматически открываются три потока со своими дескрипторами:

поток	дескриптор	
stdin	0	стандартный входной поток — обычно клавиатура
stdout	1	стандартный выходной поток — обычно экран
stderr	2	стандартный поток сообщений об ошибках

Но программа может и явным образом открывать любые новые файлы с дескрипторами.

Функции, работающие с дескрипторами файлов, описаны в файле **io.h.** Ряд используемых флагов и констант описан также в файлах **stdio.h, fcntl.h,** sys\types.h и sys\stst.h.

Файл открывается функцией open, которая возвращает дескриптор файла:

```
#include <fcntl.h>
#include<io.h>
int open(const char *path, int access, unsigned mode);
```

Параметр path указывает имя открываемого файла. Параметр access определяет режим доступа к файлу. Параметр mode является не обязательным и задает режим открытия файла.

Параметр access формируется операцией ИЛИ (|) из ряда флагов. Вот некоторые из них (полный список см. в гл. 3, в разд. 3.5.1):

O_RDONLY	только для чтения
O_WRONLY	только для записи
O_RDWR	для чтения и записи
O_CREAT	создание нового файла
O_TRUNC	если файл существует, он урезается до 0
O_BINARY	двоичный файл
O_TEXT	текстовый файл

Параметр mode может принимать значения:

S_IWRITE	разрешение записи
S_IREAD	разрешение чтения
S_IREAD | **S_IWRITE**	разрешение записи и чтения

Например, операторы

```
int handle;
if ((handle = open("Test.txt", O_CREAT | O_TEXT)) == -1)
{
 ShowMessage("Файл не удается создать");
 return;
}
```

пытаются создать новый текстовый файл, а в случае неудачи (функция **open** вернула -1) отображают сообщение об ошибке.

Имеется также функция **_creat,** осуществляющая примерно те же функции, что и **open.**

Закрывается файл функцией **close:**

```
int close(int handle);
```

Запись и чтение при работе с файлами, определяемыми дескрипторами handle, осуществляется функциями write и read:

```
#include <io.h>
int write(int handle, void *buf, unsigned len);
int read(int handle, void *buf, unsigned len);
```

В этих функциях buf — указатель на буфер, из которого записывается в файл или в который читается из файла len байтов.

Чтобы продемонстрировать работу с файлами, определенными своими дескрипторами, давайте воспроизведем пример, приведенный в разд. 2.10.2.3, в котором осуществлялась запись и чтение двух целых числе i и j, действительного числа a и строки s:

```
#include <stdio.h>
#include <stdlib.h>
ttinclude<fcntl.h>
#include <sys\stat.h>
tinclude <io.h>
#include <string.h>

int i = 1, j = 25, i1, j1;
double a = 25e6, a1;
char s[10], s1[10] ;
strcpy(s,"Иванов");

int handle;
// запись в файл
if  ((handle = open ("Test.txt", O_WRONLY | O_CREAT I O_BINARY))
     == -1)
{
 ShowMessage("Файл не удается создать");
 return;
}
write(handle, &i, sizeof (int));          // запись i
write (handle, &j, sizeof (int));         // запись j
write(handle, &a, sizeof(double));        // запись a
write(handle, s, strlen(s)+1);            // запись строки s
close(handle);

// чтение из файла
if  ((handle = open ("Test.txt", O_RDONLY | O_BINARY)) == -1)
{
 ShowMessage("Файл не удается открыть");
 return;
}
read (handle, &i1, sizeof (int) );        // чтение i
read(handle, &j1, sizeof (int));          // чтение j
read(handle, &a1, sizeof(double));        // чтение a
read(handle, s1, strlen (s)+1);           // чтение строки s
close(handle);
```

Если вы сравните этот код с тем, который был приведен в разд. 2.10.2.3, то увидите, что они практически идентичны и различаются только синтаксисом. Соответственно, и все приемы записи и чтения строк произвольной длины, рассмотренные в разд. 2.10.2.3, могут применяться и в данном случае.

Для работы с файлами, имеющими дескрипторы, могут использоваться функции **tell** и **lseek,** аналогичные рассмотренным в разд. 2.10.2.3 функциям **ftell** и **fseek,** производящими операции с указателями файлов. Имеются также функции **dup** и **dup2,** производящие операции непосредственно с дескрипторами, позволяющие создавать дубли дескрипторов или, например, перенаправлять стандартные потоки. Описания этих и иных функций вы найдете в гл. 3, в разд. 3.5.3, 3.5.4.

2.10.3 Файловый ввод/вывод с помощью потоков в стиле C++

2.10.3.1 Ввод и вывод потоков

В C++ определены три класса файлового ввода/вывода:

ifstream	входные файлы для чтения
ofstream	выходные файлы для записи
fstream	файлы для чтения и записи

Чтобы использовать эти классы, надо включить в модуль директиву

```
#include <fstream.h>
```

При работе с файлами этих классов можно использовать ряд присущих им методов, но, пожалуй, основным достоинством использования этих классов является возможность применять очень удобные операции поместить в поток (<<) и взять из потока (>>).

Создаются объекты потоков, связанные с файлами, конструкторами соответствующих классов. Например, операторы

```
ofstream outfile("Test.dat");
if(!outfile)
{
 ShowMessage("Файл не удается создать");
 return;
}
...
```

создают выходной поток **outfile,** связанный с файлом "Test.dat", создавая одновременно сам файл или, если он уже существует, урезая его длину до нуля. Если по каким-то причинам операция не может быть выполнена, значение **outfile** равно О и оператор if прерывает работу.

Аналогично может создаваться входной поток, связанный с файлом:

```
ifstream infile("Test.dat");
if(!infile)
{
 ShowMessage("Файл не удается открыть");
 return;
}
...
```

К созданным таким образом потокам можно применять операции поместить в поток (<<) и взять из потока (>>), подробно рассмотренные в разд. 1.9.15. Преимущество этих операций, работающих с текстовыми файлами, по сравнению с рассмотренными в предыдущих разделах функциями является простота использования и автоматическое распознавание типов данных. Рассмотрим, например, следующий код:

```
int i = 1, j = 25, i1, j1;
double a = 25e6, a1;
char s[40], s1[40];
strcpy(s,"Иванов");

// создание файла как выходного потока
ofstream outfile("Test.dat");
if(!outfile)
{
 ShowMessage("Файл не удается создать");
 return;
}
outfile << i << ' ' << j << ' ' << a << ' ' << s << endl;
// закрытие файла
outfile.close();

// открытие файла как входного потока
ifstream infile("Test.dat");
if(!infile)
{
 ShowMessage("Файл не удается открыть");
 return;
}
infile >> i1 >> j1 >> a1 >> s1;
```

```
// закрытие файла
infile.close();
```

В этом коде создается файл "Test.dat" и в него записываются в текстовом виде два целых числа i и j, действительное число a и строка s, содержащая одно слово, после чего манипулятором потока **endl** (см. разд. 1.9.15) осуществляется перевод строки. Причем, запись всех этих данных осуществляется одним оператором, содержащим сцепленные операции поместить в поток. Если вы сравните это с аналогичными кодами, приведенными в предыдущих разделах, то убедитесь в компактности и простоте применения этой операции.

После того, как файл закроется, в нем будет записан текст "1 25 2.5e+07 Иванов". Дальнейшие операторы создают входной поток, связанный с этим файлом и одним оператором, содержащим сцепленные операции взять из потока читает все эти данные.

Особенности применения операций << и >> детально рассмотрены в гл. 1, в разд. 1.9.15. Отметим только, что возможности операции поместить в поток можно существенно расширить использованием манипуляторов потока, которые будут обсуждаться в следующем разд. 2.10.3.2. Помимо этой операции выводить данные в поток можно еще двумя способами: методом put и методом write.

Метод put выводит в поток один символ. Например, оператор

```
outfile.put('Я');
```

выведет в поток символ "Я". Функции put допускают сцепленный вызов. Например, оператор

```
outfile.put('Я').put('\n');
```

выведет в поток символ "Я" и символ перевода строки.

Метод write выводит в файл из символьного массива, на который указывает его первый параметр, число символов, указанных вторым параметром. Например, оператор

```
outfile.write(s,5);
```

записывает в поток **outfile** 5 символов из массива s. Причем эти символы никак не обрабатываются, а просто выводятся в качестве сырых байтов данных. Среди этих символов, например, может встретиться в любом месте нулевой символ, но он не будет рассматриваться как признак конца строки.

Аналогичный метод **read** может затем прочитать эти символы в какой-то другой символьный массив и тоже без всякой обработки. Функция gcount сообщает о количестве символов, действительно прочитанных последней операцией ввода.

Теперь остановимся подробнее на вводе данных из файлового потока.

Операция взять из потока (>>) обладает особенностью, которую надо учитывать при вводе строк в массивы символов. Она читает не всю строку, а только одну лексему — последовательность символов до первого пробельного или разделительного символа. Иначе говоря, она читает не строку до символа перевода строки, а только одно слово. Это удобно, если надо производить анализ текста или искать в нем какое-то ключевое слово. Но это становится недостатком, если надо просто прочесть строку целиком.

В классе ifstream имеется еще два метода чтения из потока: get и getline. Метод get имеет три модификации: get(), get(char) и get(char *, int **n**, char **delim**).

Функция get без аргументов вводит одиночный символ из указанного потока (даже, если это символ разделитель) и возвращает этот символ в качестве значения вызова функции. Этот вариант функции get возвращает EOF, когда в потоке встречается признак конца файла.

Следующий код использует функцию get без аргумента, чтобы построчно читать и обрабатывать весь текст файла:

```
char s [80], c;
ifstream infile("Test.dat");
if (!infile)
{
 ShowMessage("Файл не удается открыть");
 return;
}
int i = 0;
while((c = infile.get()) != EOF)
 (
  if (c == '\n')
  {
// занесение нулевого символа в конец строки
   s[i] = 0;
// обработка строки
   ...
   i = 0;
  }
// формирование строки
 else s[i++] = c;
 }
// закрытие файла
 infile.close();
```

Здесь символы файла поочередно читаются в символьную переменную c. Если прочитанный символ не является символом перевода строки "\n", то символ добавляется в строку s. Если же символ равен "\n", то в конец строки заносится нулевой символ, строка подвергается какой-то обработке, после чего начинает формироваться следующая строка. Отметим, что этот код имеет один недостаток: если символу конца файла не предшествует символ перевода строки, то последняя строка оказывается без завершающего нулевого символа и остается необработанной. Нетрудно придумать дополнение кода, которое ликвидировало бы этот недостаток.

Функцию **get()** удобно использовать для поиска в файле какого-то ключевого символа. Например, цикл поиска в файле символа "$" можно организовать следующим образом:

```
while((c = infile.get ()) != EOF)
  if(c == '$') break;
if (c == '$') ...
```

Другой вариант функции-элемента get с символьным аргументом вводит очередной символ из входного потока (даже, если это символ разделитель) и сохраняет его в символьном аргументе. Этот вариант функции **get** возвращает ложь, когда встречается признак конца файла; в остальных случаях этот вариант функции **get** возвращает ссылку на тот объект потока, для которого вызывалась функция-элемент get.

При использовании этого варианта функции get приведенные ранее примеры можно оставить практически без изменений, переписав только заголовки структур **while:**

```
while(infile.get(c))
```

Третий вариант функции-элемента get принимает три параметра: символьный массив s, максимальное число символов n и ограничитель **delim** (по умолчанию символ перевода строки '\n'). Этот вариант читает символы из входного потока до тех пор, пока не достигается число символов, на 1 меньшее указанного максимального числа n, или пока не считывается ограничитель. Затем для завершения введенной строки в символьный массив, используемый в качестве буфера программы, помещается нулевой символ. Ограничитель в символьный массив не помещается, а остается во входном потоке (он будет следующим считываемым символом). Таким образом, результатом второго подряд использования функции get явится пустая строка, если только ограничитель не удалить из входного потока.

Приведенный ранее пример чтения всего файла по строкам в данном случае реализуется проще:

```
char s[80];
 ifstreaminfile("Test.dat"),·
 if(!infile)
 {
  ShowMessage("Файл не удается открыть");
  return;
 }
while(!infile.eof())
  (
  infile.get(s,80);
  infile.get();
// обработка строки
  ...
 }
// закрытие файла
 infile.close();
```

В данном случае третий аргумент в вызове get не указан. Значит подразумевается по умолчанию ограничитель "\n" и каждый вызов get читает одну строку (подразумевается, что ее длина не более 80 символов). Обратите внимание на то, что после оператора

```
infile.get(s,80);
```

добавлен оператор

```
infile.get();
```

Этот оператор удаляет из потока ограничитель. Если этого не сделать, программа зациклится.

Функция get с тремя параметрами не всегда удобна, поскольку* оставляет ограничитель в потоке, и для повторного вызова функции его приходится убирать отдельным оператором. Часто более удобна другая функция — **getline**. Эта функция действует подобно третьему варианту функции get и помещает нулевой символ после строки в символьном массиве. Но в отличие от get функция **getline** удаляет символ ограничитель из потока (т.е. читает этот символ и отбрасывает его); этот символ не сохраняется в символьном массиве.

С помощью **getline** рассмотренный выше цикл чтения файла по строкам может быть записан следующим образом:

```
while(!infile.eof())
 {
  infile.getline(s,80);
// обработка строки
  ...
 }
```

2.10.3.2 Манипуляторы потоков

В разд. 2.10.3.1 и в гл. 1 в разд. 1.9.14 рассматривался один из манипуляторов потоков — манипулятор **endl**, переводящий поток на новую строку. Имеется еще много манипуляторов потока, позволяющих форматировать вывод в файл операцией вывода в поток <<.

Чтобы посмотреть возможности манипуляторов, вы можете построить приложение, аналогичное рассмотренным в предыдущих разделах и содержащее окно **Memo1** и кнопку, обработчик события **OnClick** которой имеет следующий вид:

```
#include <fstream.h>
#include <iomanip.h>

char s[40];
```

```
// создание файла как выходного потока
ofstream outfile("Test.dat");
if(!outfile)
{
 ShowMessage("Файл не удается создать");
 return;
}
// операторы, использующие операция вывести в поток
...
// закрытие файла
outfile.close();

ifstream infile("Test.dat");
if(!infile)
{
 ShowMessage("Файл не удается открыть");
 return;
}
Memo1->Clear();
while(!infile.eof())
{
 infile.getline(s,80);
 Memo1->Lines->Add(s);
}
// закрытие файла
infile.close();
```

Манипуляторы **dec, oct, hex** и **setbase** определяют систему счисления, в которой выводятся целые числа — соответственно десятичную, восьмеричную, шестнадцатеричную и с заданным основанием. По умолчанию целые *числа* выводятся как десятичные. Послав в поток один из перечисленных модификаторов, вы можете перейти к другой системе счисления, и она будет действовать до тех пор, пока вы не примените новый модификатор. Модификатор **setbase** относится к параметризированным модификаторам потоков. В качестве параметра в него передается основание системы счисления. Для применения этого и других параметризованных модификаторов надо включить в проект заголовочный файл **<iomanip.h>**. Приведем пример использования рассмотренных модификаторов. Операторы

```
int i = 31;
outfile << i << ' ' << hex << i << ' ' << oct << i << ' '
        << setbase(10) << i << endl;
```

приведут к записи текста "31 1f 37 31". Сначала число 31 отображается в десятичном виде, потом в восьмеричном, затем в шестнадцатеричном, и в заключение опять в десятичном.

Можно управлять точностью выводимых чисел с плавающей запятой, т.е. числом разрядов справа от десятичной точки, используя манипулятор потока **setprecision** или метод **precision.** Вызов любой из этих установок точности действует для всех последующих операций вывода до тех пор, пока не будет произведена следующая установка точности.

Чтобы посмотреть возможности способов управления точностью, вы можете записать оператор:

```
for(int i = 0; i < 10; i++)
  outfile << setprecision(i) << sqrt(3.0) << endl;
```

Этот оператор изменяет в цикле параметр манипулятора **setprecision** от 0 до 9 и тем самым изменяет точность вывода в файл значения корня квадратного из 3. Значение параметра 0 приводит к установке точности по умолчанию, которая равна 6. Результат работы приведенного оператора следующий:

```
1.73205
2
1.7
1.73
1.732
1.7321
1.73205
1.732051
1.7320508
1.73205081
```

Аналогичный результат даст следующий код, использующий метод **precision:**

```
for(int i = 0; i < 10; i++)
{
 outfile.precision(i);
 outfile << sqrt(3.0) << endl;
}
```

Если в функции **precision** не задавать параметров, например,

```
int i = outfile.precision();
```

то она вернет текущую установку точности.

Манипулятор потока **setw** и метод **width** устанавливают ширину поля (т.е. число символьных позиций, в которые значение будет выведено, или число символов, которые будут введены) и возвращает предыдущую ширину поля. Если обрабатываемые значения имеют меньше символов, чем заданная ширина поля, то для заполнения лишних позиций используются заполняющие символы. По умолчанию заполняющими символами являются пробелы и вставляются они перед значащими символами, т.е. происходит выравнивание вправо. Если число символов в обрабатываемом значении больше, чем заданная ширина поля, то лишние символы не отсекаются и число будет напечатано полностью. Установка ширины поля влияет только на следующую операцию поместить в поток; затем ширина поля устанавливается неявным образом на 0, т.е. поле для представления выходных значений будут просто такой ширины, которая необходима. Функция **width,** не имеющая аргументов, возвращает текущую установку ширины поля.

Заполняющие символы могут устанавливаться манипулятором **setfill(char)** или методом **fill.**

Например, следующие операторы демонстрируют влияние ширины поля на результат вывода числа 25:

```
int j = 25;
for(int i = 0; i < 5; i++)
  outfile << setw(i) << j << endl;
```

Результат работы этих операторов следующий:

```
25
25
25
 25
  25
```

Из этого результата видно, что пока ширина поля меньше числа символов в выводимом числе, она ни на что не влияет, а при большой ширине поля происходит выравнивание числа вправо.

Такой же результат дает и следующий цикл, использующий метод **width:**

```
for(int i = 0; i < 5; i++)
{
 outfile.width(i);
 outfile << j << endl;
}
```

Если вывести в поток модификатор **setfill,** то заполняющие символы изменятся. Например, оператор

```
for(int i = 0; i < 5; i++)
  outfile << setfill('*') << setw(i) << j << endl;
```

приведет к результату:

```
25
25
25
*25
**25
```

Мы рассмотрели многие (но еще не все) манипуляторы потоков. Пользователи могут создавать собственные манипуляторы потоков. В качестве примера того, как это делается, ниже приводится код функции, создающей манипулятор, названный tab, который выводит в поток символ табуляции "\t".

```
//Создание манипулятора tab
ostream& tab(ostream& output)
{
   return output << '\t';
}
```

Если вы ввели в приложение такую функцию, то в дальнейшем можете использовать этот манипулятор. Например, оператор

```
outfile << 'A' << tab << 'B' << tab << 'C' << endl;
```

выведет символы "A", "B" и "C", разделенные символами табуляции:

```
A   B   C
```

2.10.3.3 Флаги состояния формата

В классе ios — базовом классе всех потоков ввода/вывода определены следующие флаги формата.

ios::skipws	пропуск символов разделителей во входном потоке
ios::left	выравнивание по левой границе поля
ios::right	выравнивание по правой границе поля
ios::internal	выравнивание знака или основания системы счисления по левой, а числа — по правой границам поля
ios:: dec	десятичная система счисления, устанавливается манипулятором dec
ios::oct	восьмеричная система счисления, устанавливается манипулятором oct
ios::hex	шестнадцатеричная система счисления, устанавливается манипулятором hex
ios::showbase	вывод основания системы счисления
ios::showpoint	обязательная печать десятичной точки и нулевых младших разрядов
ios::uppercase	вывод в верхнем регистре символов "X" и "E" в шестнадцатеричном и экспоненциальном форматах

ios::showpos	вывод символа "+" перед положительным числом
ios::scientific	экспоненциальное представление действительных чисел
ios::fived	формат действительных чисел с фиксированной точкой

Флаги состояния формата управляются методами **flags, setf и unsetf,** или манипуляторами потоков **setw, setiosflags и resetiosflags.**

Метод **flags** используется для задания сразу всех флагов. При этом те флаги, которые должны быть установлены, объединяются операцией поразрядного ИЛИ (|) в одно значение типа **long,** передаваемое методу как параметр. Метод **flags** возвращает значение типа **long,** содержащее предыдущие значения опций. Это значение часто сохраняется с тем, чтобы можно было впоследствии вызвать функцию **flags** с этим сохраненным значением и восстановить предыдущие значения опций.

Метод **setf** и параметризованный манипулятор потока **setiosflags** имеют единственный аргумент, который устанавливает один или более флагов, соединенных операцией |, и может использовать текущие установки флагов для создания нового состояния формата. Например, манипулятор

```
setiosflags(ios::showpos | ios::showpoint)
```

устанавливает флаги **ios::showpos и ios::showpoint.**

Манипулятор потока **resetiosflags** и метод **unsetf** наоборот, сбрасывают флаги, которые указаны их параметром. Чтобы использовать перечисленные параметризованные манипуляторы потока, надо в приложение включить директиву **#include <iomanip.h>.**

Приведем примеры использования перечисленных флагов состояния формата.

Оператор, использующий флаг **showpoint:**

```
outfile << 1. << "  " << 1.1 << "  " <<
      setiosflags(ios::showpoint) << 1. << "  " << 1.1 << endl;
```

дает результат:

```
1   1.1   1.00000   1.10000
```

Оператор, использующий флаги **right, left и internal:**

```
outfile  << setw(6) << -1.1 << endl
         << setw(6) << resetiosflags(ios::right)
         << setiosflags(ios::left) << -1.1 << endl
         << setw(6) << resetiosflags(ios::left)
         << setiosflags(ios::internal) << -1.1 << endl;
```

дает результат:

```
  -1.1
-1.1
-  1.1
```

Обратите внимание на то, что надо сбрасывать модификатором **resetiosflags** ранее установленный флаг, чтобы при каждом выводе только один из флагов **right, left и internal** был установлен.

Оператор, использующий флаг **showbase:**

```
outfile << 63 << oct << " " << 63 << hex << "  " << 63
        << setiosflags(ios::showbase) << dec << endl
        << 63 << oct << " " << 63 << hex << " " << 63 << endl;
```

дает результат:

```
63  77   3f
63  077  Ox3f
```

Обратите внимание, что флаги системы счисления устанавливаются не модификатором **setiosflags,** а модификаторами **dec, oct, hex.**

Оператор, использующий флаги **scientific и fixed:**

```
outfile << "По умолчанию:" << endl
        << 0.0123 << ' ' << 1.23e6 << endl << endl
        << "Флаг scientific:" << setiosflags(ios::scientific)
        << endl
        << 0.0123 << ' ' << 1.23e6 << endl << endl
        << "Флаг fixed:" << resetiosflags(ios::scientific)
        << setiosflags(ios::fixed) << endl
        << 0.0123 << ' ' << 1.23e6 << endl;
```

дает результат:

```
По умолчанию:
0.0123  1.23e+06

Флаг scientific:
1.230000e-02  1.230000e+06

Флаг fixed:
0.012300  1230000.000000
```

Обратите внимание, что по умолчанию значения чисел с плавающей запятой сами выбирают формат представления и он, пожалуй, наиболее привлекателен.

Оператор, использующий флаги **showpos и showpoint:**

```
outfile << setprecision(4) << setw(3) << 60. << endl
        << setiosflags(ios::showpos | ios::showpoint) << 60.
        << endl;
```

дает результат:

```
 60
+60.00
```

В этом примере один манипулятор **setiosflags** устанавливает сразу два флага.

2.11 Массивы

2.11.1 Одномерные массивы

Массив представляет собой структуру данных, позволяющую хранить под одним именем совокупность данных любого, но только одного какого-то типа. Массив характеризуется своим именем, типом хранимых элементов, размером (числом хранимых элементов), нумерацией элементов и размерностью. В данном разделе мы ограничимся одномерными массивами, т.е. массивами с размерностью 1.

Объявление переменной как одномерного массива имеет вид:

```
тип переменная [константное_выражение]
```

Например, оператор

```
int A[10];
```

объявляет массив с именем A, содержащий 10 целых чисел. Доступ к элементам этого массива осуществляется выражением **A[i],** где **i** -- индекс, являющийся в данном примере, как видно из объявления, целым числом в диапазоне 0-9. Например, **A[0]** — значение первого элемента, **A[1]** — второго, **A[9]** — последнего. Обратите внимание, что индекс последнего элемента на 1 меньше размера массива. Это связано с тем, что индексы начинаются с 0.

Приведем примеры использования этого массива. Код

```
A[0] = 1;
A[1] = 1;
for(int i = 2; i < 10; i++) A[i] = A[i-2]+A[i-1];
```

заполняет массив так называемыми числами Фибоначчи, первые 2 из которых равны 1, а каждое последующее равно сумме двух предыдущих.

Элементы массива могут иметь любой тип. Например, предложение

```
char S[10];
```

объявляет массив символов. Массив символов это фактически строка (см. разд. 2.5.1) и с ним можно во многом обращаться как со строкой, хотя можно обращаться и как с массивом. При использовании массива символов как строки надо только иметь в виду, что это строка фиксированной допустимой длины. И число символов, помещаемых в строку, не должно превышает объявленного размера массива n - 1, поскольку строка кончается нулевым символом.

Объявление переменной массива можно совмещать с заданием элементам массива начальных значений. Эти значения перечисляются в списке инициализации после знака равенства, разделяются запятыми и заключаются в фигурные скобки. Например:

```
int A[10] = {1,2,3,4,5,6,7,8,9,10};
char S[10] = {"abcdefghi\0"};
```

Если начальных значений меньше, чем элементов в массиве, оставшиеся элементы автоматически получают нулевые начальные значения. Например, оператор

```
int A[10] = {1,2,3};
```

задает значения первым трем элементам, а остальные будут равны 0. Оператор

```
int A[10] = {0};
```

присваивает нулевые значения всем элементам массива.

Если массив при его объявлении не инициализирован, то его элементы имеют случайные значения. Элементы такого массива нельзя использовать в выражениях, пока им не будут присвоены какие-нибудь значения.

В массивах символов задание нулей элементам, не указанным в списке инициализации, равносильно заданию нулевых символов, означающих конец строки. Поэтому приведенное выше объявление переменной S с ее инициализацией избыточно. Нулевой символ в конце можно не указывать. Например, нормально будут восприняты такие объявления:

```
char S[10] = {"abcdefghi"};
char S1[10] = {"abc"};
```

Последнее объявление выделяет место под массив из 10 элементов, но инициализирует его строкой из трех элементов.

В объявлении со списком инициализации размер массива можно не указывать. Тогда количество элементов массива будет равно количеству элементов в списке начальных значений. Например, объявление

```
int A[ ] = {1, 2, 3, 4, 5};
```

создает массив из пяти элементов. Объявление

```
char S1[ ] = {"abc"};
```

создает массив из четырех элементов — три значащих символа плюс нулевой символ.

В объявлении массива в качестве размера лучше всегда использовать именованные константы. Например, ниже приведено объявление массива и оператор, подсчитывающий сумму его элементов:

```
int A[10];
// операторы заполнения массива
...
// подсчет суммы
int Sum = A[0] ;
for(int i = 1; i < 10; i++) Sum += A[i];
```

Если в дальнейшем вы решите, что вам требуется массив А не из 10 элементов, а, например, из 100, вы должны будете изменить размер массива и в объявлении А, и во всех операторах, работающих с этим массивом (в данном случае в операторе for). А ведь таких операторов в разных частях программы может быть очень много. О такой программе говорят, что она плохо масштабируется.

Грамотнее реализовать этот пример следующим образом:

```
const Amax = 10;
int A[Amax];
// операторы заполнения массива
...
// подсчет суммы
int Sum = A[0] ;
for(int i = 1; i < Amax; i++) Sum += A[i];
```

В этом случае вы вводите именованную константу Amax и используете ее во всех операторах, в которых вам требуется размер массива. Тогда при необходимости изменить размер массива вам достаточно изменить его только в одном операторе, объявляющем Amax. Программа сразу становится масштабируемой. А объявление Amax как константы гарантирует, что объявленное значение не будет случайно изменено где-то в программе.

Аналогичный результат можно получить, если заменить объявление константы директивой компилятора **#define** (см. гл. 1, разд. 1.4.2).

```
#define Amax 10
```

Как правило, все размеры массивов в программе следует определять именованными константами или макросами. Это делает программу *более* понятной и существенно облегчает ее отладку и сопровождение.

В ряде случаев требуются константные массивы, данные из которых программа может только читать. Такие массивы обязательно должны инициироваться в момент объявления. Например:

```
const AnsiString Day[] = {"понедельник", "вторник", "среда",
                          "четверг", "пятница", "суббота",
                          "воскресенье"};
```

2.11.2 Многомерные массивы

Можно объявлять и многомерные массивы, т.е. массивы, элементами которых являются массивы. Например, двумерный массив можно объявить таким образом:

```
int A2[10] [3];
```

Этот оператор описывает двумерный массив, который можно представить себе как таблицу, состоящую из 10 строк и 3 столбцов.

Доступ к значениям элементов многомерного массива обеспечивается через индексы, каждый из которых заключается в квадратные скобки. Например, **A2[3][2]** - - значение элемента, лежащего на пересечении четвертой строки и третьего столбца (помните, что индексы начинаются с 0).

Если многомерный массив инициализируется при его объявлении, список значений по каждой размерности заключается в фигурные скобки. Приведенный ниже оператор объявляет трехмерный массив **A3** размерностью 4 на 3 на 2.

```
int A3[4][3][2] = {{{0,1},{2,3},{4,5}},
                   {{6,7},{8,9},{10,11}},
                   {{12,13},{14,15},{16,17}},
                   {{18,19},{20,21},{22,23}}};
```

Этот оператор создает массив **A3,** четыре строки которого являются матрицами вида

0	1
2	3
4	5

6	7
8	9
10	11

12	13
14	15
16	17

18	19
20	21
22	23

Например, элемент **A3[0][1][0]** равен 2, элемент **A3[3][0][1]** равен 19 и т.д.

Если в списке инициализации в какой-то из размерностей не хватает данных, то все дальнейшие не перечисленные элементы считаются равными нулям.

2.11.3 Операции с массивами, передача массивов как параметров

Имя массива является константным указателем на первый элемент массива. Взаимосвязь массивов и указателей подробно рассмотрена в разд. 2.8. Поскольку имя массива — константный указатель, оно не может модифицироваться, и к нему не применимы все операции присваивания.

К имени массива можно применять операцию **sizeof,** которая в этом случае возвращает значение, равное общему объему памяти, отведенному под все элементы массива. Таким образом, число элементов массива A можно определить выражением

```
sizeof(A) / sizeof(A[0])
```

поскольку под каждый элемент массива отведен одинаковый объем памяти.

Подобное вычисление размера массива выполняет макрос **ARRAYSIZE.** Приведенное выше выражение эквивалентно выражению

```
ARRAYSIZE(A)
```

При передаче массива в функцию в качестве параметра заголовок функции содержит тип и имя массива с последующими пустыми квадратными скобками. Например, если функция F должна принимать массив как параметр, ее прототип может иметь вид:

```
void F(int Ar[]);
```

Обращение к такой функции может быть записано так:

```
const Amax = 10;
int A[Amax];
...
F(A);
```

Как видно, в вызове функции указывается просто имя массива. Внутри функции к элементам этого массива можно обращаться обычным образом, например, **Ar[2].** C++ передает имя массива в функцию по ссылке (см. гл. 1, разд. 1.7.2). Это значит, что если функция изменяет значения элементов массива, то изменяются элементы исходного массива, который передавался в функцию.

В большинстве случаев только имени массива мало, чтобы провести в функции обработку его элементов. Внутри функции требуется знать размер массива, чтобы можно было организовать его циклическую обработку. Поэтому обычно в функцию передается не только массив, но и его размер. При этом заголовок функции может иметь вид:

```
void F(int Ar[], int N);
```

а вызов функции:

```
F(A, Amax);
```

Чаще библиотечные функции требуют в качестве второго параметра не размер массива, а значение его последнего индекса, которое на единицу меньше размера. В этом случае вызов функции может иметь вид:

```
F(A, Amax - 1);
```

В частности, такого вызова требуют все функции Object Pascal, использующие так называемый открытый массив. Поскольку подобные вызовы функции встречаются довольно часто, в файле **sysdefs.h** определен макрос **EXISTINGARRAY,** который позволяет оформить передачу массива более компактно. При использовании этого макроса приведенный выше вызов можно оформить так:

```
F(EXISTINGARRAY(A));
```

При развертывании макрос **EXISTINGARRAY** передаст в функцию имя массива как первый параметр и значение последнего индекса как второй параметр. При этом макрос использует приведенное ранее выражение для подсчета числа элементов массива через операцию **sizeof**.

Некоторые функции Object Pascal, используемые и в C++Builder, могут воспринимать в качестве параметров так называемые открытые массивы констант, в которых могут содержаться элементы разных типов. В файле **sysdefs.h** описан макрос **OPENARRAY,** позволяющий обращаться к таким функциям. Без дополнительных разъяснений приведем форму записи такого макроса:

```
OPENARRAY(TVarRec, (элемент_1, элемент_2, ...))
```

Число передаваемых элементов может достигать 19.

Передача массива по ссылке не гарантирует защиты от несанкционированного изменения программой значений элементов массива. Если необходимо защитить массив от подобных изменений, его надо передать в функцию как константный:

```
void F(const int Ar[], int N);
```

Пусть, например, вы хотите написать функцию, подсчитывающую сумму элементов массива целых. Тогда вы можете оформить ее следующим образом:

```
int Sum(const int A[], int N)
{
// N — размер массива
int S = A[0];
for(int i = 1; i < N; i++) S += A[i];
return S;
}
```

Ниже приведен пример тестирования этой функции.

```
#define Bmax 10
int B[10] = {1,2,3,4,5,6,7,8,9,10};
ShowMessage("Сумма равна " + IntToStr(Sum(B,Bmax)));
```

При вызове функции не обязательно передавать весь массив. Можно передать только какую-то его часть. Например, вы можете передать в функцию параметр размера массива, меньший истинного. Если вы в приведенном примере в качестве второго параметра передадите в функцию не **Bmax, а Bmax** - 2, то функция будет обрабатывать только восемь первых элементов с индексами от 0 до 7. Можно и в качестве начала массива передать в функцию указатель на какой-то элемент массива. Например, если вы обратитесь к функции так:

```
Sum(B + 2, Bmax - 2)
```

то вы передадите в нее указатель не на первый, а на третий элемент. Поэтому, когда функция будет обращаться к элементам массива от 0 до 7, в действительности она будет работать с элементами, индексы которых от 2 до 9. Т.е. сумма будет посчитана по элементам, начиная с третьего.

Если в функцию передается многомерный массив, то в заголовке только квадратные скобки первой размерности остаются пустыми, а в скобках следующих размерностей должны указываться константами их размеры. Например, если функция **F2** должна принимать двумерный массив размером 3 на 3, то ее заголовок может иметь вид:

```
void F(const int Ar[][3]);
```

Вызов этой функции производится обычной передачей в нее имени массива. Например, **F(A).**

2.12 Структуры

2.12.1 Структуры в стиле C

Структуры — это составные типы данных, построенные с использованием других типов. Они представляют собой объединенный общим именем набор данных различных типов. Именно тем, что в них могут храниться данные разных типов, они и отличаются от массивов, хранящих данные одного типа.

Отдельные данные структуры называются элементами или полями. Все это напоминает запись в базе данных, только хранящуюся в оперативной памяти компьютера.

Простейший вариант объявления структуры может выглядеть следующим образом:

```
struct TPers {
  AnsiString Fam,Nam,Par;
  unsigned   Year;
  bool       Sex;
AnsiString Dep;
};
```

Ключевое слово **struct** начинает определение структуры. Идентификатор **TPers** — тег (обозначение, имя-этикетка) структуры. Тег структуры используется при объявлении переменных структур данного типа. В этом примере имя нового типа — **TPers.** Имена, объявленные в фигурных скобках описания структуры — это элементы структуры. Элементы одной и той же структуры должны иметь уникальные имена, но две разные структуры могут содержать не конфликтующие элементы с одинаковыми именами. Каждое определение структуры должно заканчиваться точкой с запятой.

Определение **TPers** содержит шесть элементов. Предполагается, что такая структура может хранить данные о сотруднике некоего учреждения. Типы данных разные: элементы **Fam, Nam,** Par и Dep — строки, хранящие соответственно фамилию, имя, отчество сотрудника и название отдела, в котором он работает. Элемент **Year** целого типа хранит год рождения, элемент Sex булева типа хранит сведения о поле. Элементы структуры могут быть любого типа, но структура не может содержать экземпляры самой себя. Например, элемент типа **TPers** не может быть объявлен в определении структуры **TPers.** Однако может быть включен указатель на другую структуру типа **TPers.** Структура, содержащая элемент, который является указателем на такой же структурный тип, называется структурой с самоадресацией. Такие структуры очень полезны для формирования различных списков (см. разд. 2.12.2).

Само по себе объявление структуры не резервирует никакого пространства в памяти; оно только создает новый тип данных, который может использоваться для объявления переменных. Переменные структуры объявляются так же, как переменные других типов. Объявление

```
TPers Pers, PersArray[10], *Ppers;
```

объявляет переменную **Pers** типа **TPers,** массив **PersArray** — с 10 элементами типа **TPers** и указатель **Ppers** на объект типа **TPers.**

Переменные структуры могут объявляться и непосредственно в объявлении самой структуры после закрывающейся фигурной скобки. В этом случае указание тега не обязательно:

```
struct {
  AnsiString Fam,Nam,Par;
  unsigned   Year;
  bool       Sex;
AnsiString Dep;
}Pers, PersArray[10], *Ppers;
```

Для доступа к элементам структуры используются операции доступа к элементам: операция точка (.) и операция стрелка (->). Операция точка обращается к элементу структуры по имени объекта или по ссылке на объект. Например:

```
Pers.Fam = "Иванов";
Pers.Nam = "Иван";
Pers.Par = "Иванович";
Pers.Year = 1960;
Pers.Sex = true;
Pers.Dep = "Бухгалтерия";
```

Операция стрелка обеспечивает доступ к элементу структуры через указатель на объект. Допустим, что выполнен оператор

```
Ppers = &Pers;
```

который присвоил указателю **Ppers** адрес объекта **Pers.** Тогда указанные выше присваивания элементам структуры можно выполнить так:

```
Ppers->Fam = "Иванов";
Ppers->Nam = "Иван";
Ppers->Par = "Иванович";
Ppers->Year = 1960;
Ppers->Sex = true;
Ppers->Dep = "Бухгалтерия";
```

2.12.2 Самоадресуемые структуры

Теперь рассмотрим еще один вид структур — самоадресуемые структуры. Нередко в памяти надо динамически размещать (см. гл. 1, разд. 1.11) последовательность структур, как бы формируя некий фрагмент базы данных, предназначенный для оперативного анализа и обработки. Поскольку динамическое размещение проводится в непредсказуемых местах памяти, то такие структуры надо снабдить элементами, содержащими указатели на следующую аналогичную структуру. Такие структуры со ссылками на аналогичные структуры и называются самоадресуемыми. Ниже приведена схема связи таких структур в последовательность. Полю указателя в последней структуре обычно присваивается значение **NULL,** что является признаком последней структуры при организации поиска в списке.

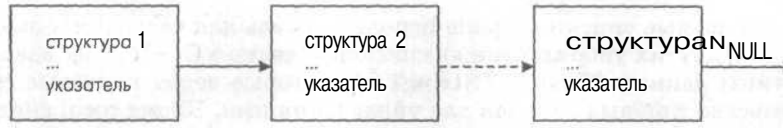


Рис. 2.1. Список структур

Если мы хотим структуру, рассмотренную в разд. 2.12.1, сделать самоадресуемой, следует изменить ее объявление следующим образом:

```
struct TPers {
  AnsiString  Fam,Nam,Par;
  unsigned    Year;
  bool        Sex;
  AnsiString  Dep;
  TPers * pr;
};
```

Приведем пример формирования в памяти списка таких структур. Для этого надо определить три переменные, являющиеся указателями на структуры:

```
TPers *PO = NULL, *Pnew, *Pold;
```

Первая из этих переменных будет всегда указывать на первую структуру в списке. Две остальные переменные — вспомогательные. Если в некоторый момент возникла необходимость динамически разместить в памяти очередную структуру и вставить ее в конец списка, это можно сделать следующим кодом:

```
// выделение памяти под новую структуру
Pnew = new TPers;

// заполнение элементов структуры
Pnew->Fam = "Иванов";
Pnew->Nam = "Иван";
Pnew->Par = "Иванович";
Pnew->Year = 1960;
Pnew->Sex = true;
Pnew ->Dep = "Бухгалтерия";
Pnew->pr = NULL;

if (PO == NULL) PO = Pnew; // PO — указатель на первую структуру
else Pold->pr = Pnew;      // указатель на очередную структуру
Pold = Pnew;
```

Если список еще не начат **(PO = NULL),** то указателю **PO** присваивается ссылка на вновь размещенную структуру (Pnew). В противном случае ссылка на новую структуру присваивается полю **pr** предыдущей структуры в списке **(Pold).** Таким образом новая структура включается в общий список. Полю **pr** этой структуры присваивается значение **NULL.** Это является признаком того, что данная структура является последней в списке.

Сформировав список в памяти, далее легко его просматривать, проходя в цикле по указателям. Например:

```
Pnew = PO;
while(Pnew != NULL)
{
 ShowMessage(Pnew->Fam + " " + Pnew->Nam + " " + Pnew->Par);
 Pnew = Pnew->pr;          // переход к новой структуре
}
```

Легко также делать в списке перестановки структур, их удаление и т.п. Для всех этих операций не надо ничего перемещать в памяти. Достаточно только изменять соответствующие ссылки в полях **pr.**

Раньше подобные списки широко использовались для создания в памяти стеков, очередей и других упорядоченных списков. Однако в C++Builder введены специальные типы данных **TList и TStringList,** которые ведут подобные списки и имеют множество удобных методов для управления *ими.* Кроме того, аналогичное объединение элементов в список используется для классов связных списков стандартной библиотеки, рассмотренных в разд. 5.4.4.

2.12.3 Структуры в стиле C++

Все, что рассмотрено в предыдущих разделах, относится как к языку C, так и к C++. Но в C++ понятие структуры существенно расширено и приближено к понятию класса (см. разд. 2.14).

В частности, в структурах кроме рассмотренных ранее данных-элементов разрешается описывать функции-элементы. Рассмотрим это на примере использованной в предыдущих разделах структуры **TPers.** Давайте введем в эту структуру функцию-элемент **Show,** отображающую информацию, хранящуюся в структуре:

```
struct TPers {
  AnsiString Fam,Nam,Par;
  unsigned   Year;
  bool       Sex;
  AnsiString Dep;
  TPers * pr;
  void Show()
  {
   ShowMessage("Сотрудник отдела \""+Dep+"\" "+Fam+" "+Nam+" "+
               Par+", "+IntToStr(Year)+" г.р., пол "+
               (Sex ? "мужской" : "женский"));
  }
};
```

Функция **Show** отображает информацию вида: "Сотрудник отдела "Бухгалтерия" Иванов Иван Иванович, 1960 г.р., пол мужской".

Обращение к этой функции-элементу производится через переменную структуры операцией точка или через указатель на переменную операцией стрелка. Например:

```
Pers.Show();
Pnew->Show();
```

С использованием введенной функции **Show** приведенный в разд. 2.12.2 пример просмотра списка можно упростить:

```
Pnew = P0;
while (Pnew != NULL)
 {
 Pnew->Show();
 Pnew = Pnew->pr;
}
```

В C++ можно вводить спецификаторы доступа к данным-элементам и функциям-элементам так же, как это делается в классе. Разрешаются спецификаторы **public** (открытый) и **private** (закрытый). Закрытые элементы структуры могут быть доступны только для функций-элементов этой структуры. Ни через объект, ни через указатель на объект доступ к ним невозможен. Закрытыми объявляются какие-то вспомогательные данные-элементы, не представляющие интереса для пользователя, а также вспомогательные функции (утилиты), требующиеся для работы основных функций-элементов структуры.

Открытые элементы структуры могут быть доступны для любых функций в программе. Основная задача открытых элементов состоит в том, чтобы дать клиентам структуры представление о возможностях, которые она имеет. Это открытый интерфейс структуры.

По умолчанию доступ к элементам структуры **public** — открытый. Если вам надо спрятать от пользователя какие-то элементы, укажите спецификатор **private,** завершающийся двоеточием, и помещайте после него объявления закрытых элементов. Все, что помещено после спецификатор **private** до конца структуры или до спецификатора **public,** будет скрыто от пользователя. Например, в следующем объявлении структуры

```
struct MyStr { .
  int x, y;
  int Get () ;
private:
  int a, b;
  void F();
};
```

данные x и y и функция Get — открытые и могут использоваться при работе со структурой, а данные a и b и функция F — закрытые и ими может пользоваться только функция **Get.**

Есть еще ряд особенностей, сближающих в С++ структуры и классы. Они будут рассмотрены в разд. 2.14, посвященном классам.

2.12.4 Битовые поля

Язык Си++ предоставляет возможность задавать количество битов, в которых хранятся элементы типов **unsigned** или **int** структуры (а также класса и объединения — см. разд. 2.14 и 2.13. Такие элементы называются битовыми полями. Битовые поля позволяют рационально использовать память с помощью хранения данных в минимально требуемом количестве битов.

В структуре **TPers,** использовавшейся в предыдущих разделах, можно, например, сократить затраты на хранение года рождения и пола сотрудника. Если ориентироваться на даты до 2047 года, то для хранения года рождения достаточно 11 битов. Если вы рассчитываете, что ваша программа просуществует дольше, то можете даже выделить под год 12 битов — этого хватит на ближайшие две тысячи лет. А под хранение сведений о поле вполне достаточна 1 бита. Таким образом, под эти два элемента вам достаточно 2 байтов, а в описанной ранее версии структуры под эти элементы отводилось 5 байтов: 4 под год плюс один под пол. Выигрыш 3 байта. Конечно, немного, но если при выполнении вашей программы в памяти формируются списки из тысяч структур, то такой выигрыш уже может быть заметен.

При объявлении битового поля вслед за указанием типа элемента ставится двоеточие ":" и пишется целочисленная константа, задающая ширину поля (т.е. число битов, в которых хранится этот элемент). Ширина поля должна быть целочисленной константой в диапазоне между 0 и заданным общим числом битов, используемых для хранения целого значения типа int в вашей системе. Например:

```
struct TPers {
  AnsiString Fam,Nam,Par;
  AnsiString Dep;
  TPers *    pr;
  unsigned   Year : 12;
  bool       Sex : 1;
};
```

Можно задавать неименованное битовое поле. Такое поле используется в структуре как заполнение. Дело в том, что при работе с битовыми полями надо учитывать длину машинного слова. Если следующий элемент структуры не является битовым полем, то место его хранения должно начинаться с нового машинного слова. Для округления объемов памяти до слова, т.е. для заполнения оставшихся неиспользованными битов и вводятся неименованные битовые поля. В приведенном ниже примере неименованное поле шириной в 3 бита используется как заполнение:

```
struct Example {
  unsigned a : 13;
  unsigned   :  3;
  unsigned b :  4;
};
```

Можно использовать неименованное битовое поле нулевой ширины, которое воспринимается как указание выровнять следующее битовое поле по границе нового элемента памяти.

Следует предостеречь от чрезмерного увлечения битовыми полями. Манипуляции с битовыми полями являются машинно-зависимыми. Например, в некоторых компьютерах битовые поля могут пересекать границы машинного слова, тогда как в других компьютерах это недопустимо.

2.13 Объединения

Объединение (union) — это область памяти, в которой *в* разные моменты времени могут находиться объекты разных типов. В любой момент времени объединение может содержать максимум один объект, потому что элементы объединения совместно используют одну и ту же область памяти. На программиста возлагается обязанность следить за тем, чтобы к данным в объединении обращались по имени элемента соответствующего типа данных. Если тип ссылки на элемент объединения не соответствует типу данных, хранящемуся в этот момент в объединении, то возникает ошибка, последствия которой зависят от реализации системы.

В разные отрезки времени выполнения программы некоторые объекты могут быть не нужны, т.е. программе требуется только часть ее объектов. Вместо того, чтобы впустую растрачивать память на объекты, которые используются не постоянно, можно поместить их в объединение, где они будут делить между собой одну и ту же область памяти. Число байтов памяти, выделяемых для объединения, должно быть не меньше, чем размер самого большого элемента объединения.

Не всегда объединение может быть легко перенесено на другие компьютерные платформы. Перенесется ли объединение, или нет, часто зависит от соглашений о выравнивании в памяти типов данных элементов объединения. Так что использование объединений снижает мобильность вашей программы.

Объединения объявляются при помощи ключевого слова **union** в таком же формате, как структуры и классы (см. разд. 2.12.1 и 2.14). Например:

```
union Tunion {
  int    i;
  double d;
  char * s;
};
```

Это объявление создает тип объединения с именем **Tunion,** которое хранит в одной и той же области памяти или целое значение i, или действительное значение d, или указатель на строку s.

Само по себе объявление объединения создает новый тип, но не объект. В дальнейшем для использования объединения надо объявить переменную этого типа, например:

```
Tunion N;
```

К элементам переменной типа объединения можно обращаться так же, как к элементам структуры или класса. Например, вы можете записать операторы:

```
N.i = 5;
...
N.d = 5.1;
...
char *S = "объединение";
N.s = S;
```

Но учтите, что при использовании объединения вам надо все время знать, какое значение вы занесли в эту переменную последней операцией присваивания. Если, например, вы выполнили первый из приведенных выше операторов, а затем обратились к элементу d, то вы получите бессмысленное значение. А если вы по ошибке обратились в этом случае к элементу s, то вас ждут крупные неприятности, поскольку неизвестно, на что будет указывать s.

Использование объединений позволяет экономить ресурсы, но существенно усложняет программирование и затрудняет отладку. Так что решайте, что вам важнее, и не увлекайтесь излишне объединениями.

2.14 Классы

Классы и шаблоны классов - это, пожалуй, самое главное в C++. Все, связанное с этими понятиями, в рамках данной книги рассмотреть невозможно. В реализации классов и шаблонов есть масса тонкостей, на которых я останавливаться не буду. Но все, необходимое для использования классов и их шаблонов, а также для разработки достаточно сложных собственных классов и шаблонов, будет рассмотрено.

2.14.1 Объявление класса

Класс — это тип данных, определяемый пользователем. То, что в стандартной библиотеке C++ и в C++Builder имеется множество предопределенных классов, не противоречит этому определению — ведь разработчики C++Builder тоже пользователи C++. Понятия класса, структуры (см. разд. 2.12) и объединения (см. разд. 2.13) в C++ довольно близки друг к другу. Поэтому почти все, что будет далее говориться о классах, применимо также к структурам и объединениям.

Класс должен быть объявлен до того, как будет объявлена хотя бы одна переменная этого класса. Т.е. класс не может объявляться внутри объявления переменной.

Синтаксис объявления класса следующий:

```
class <имя класса> : <список классов — родителей>
{
  public:              // доступно всем
    <данные, методы, свойства, события>
  __published          // видны в Инспекторе Объекта и изменяемы
    <данные, свойства>
  protected:           // доступно только потомкам
    <данные, методы, свойства, события>
  private:             // доступно только в классе
    <данные, методы, свойства, события>
) <список переменных>;
```

Например:

```
class MyClass : public Class1, Class2
{
public:
  MyClass(int = 0);
  void SetA(int);
  int GetA(void);
private:
  int FA;
  double B, C;
protected:
  int F(int);
};
```

Имя класса может быть любым допустимым идентификатором. Идентификаторы классов, наследующих классам библиотеки компонентов C++Builder, принято начинать с символа "T".

Класс может наследовать поля (они называются данные-элементы), методы (они называются функции-элементы), свойства, события от других классов — своих предков, может отменять какие-то из этих элементов класса или вводить новые. Если предусматриваются такие классы-предки, то в объявлении класса после его имени ставится двоеточие и затем дается список родителей. В приведенном выше примере предусмотрено множественное наследование классам **Classl и CIass2.** Если среди классов-предков встречаются классы библиотеки компонентов C++Builder или классы, наследующие им, то множественное наследование запрещено.

Если объявляемый класс не имеет предшественников, то список классов-родителей вместе с предшествующим двоеточием опускается. Например:

```
class MyClass1
{
 ...
};
```

Доступ к объявляемым элементам класса определяется тем, в каком разделе они объявлены. Раздел **public** (открытый) предназначен для объявлений, которые доступны для внешнего использования. Это открытый интерфейс класса. Раздел **__published** (публикуемый) содержит открытые свойства, которые появляются в процессе проектирования на странице свойств Инспектора Объектов и которые, следовательно, пользователь может устанавливать в процессе проектирования. Раздел **private** (закрытый) содержит объявления полей и функций, используемых только внутри данного класса. Раздел **protected** (защищенный) содержит объявления, доступные только для потомков объявляемого класса. Как и в случае закрытых элементов, можно скрыть детали реализации защищенных элементов от конечного пользователя. Однако в отличие от закрытых, защищенные элементы остаются доступны для программистов, которые захотят производить от этого класса производные классы, причем не требуется, чтобы производные классы объявлялись в этом же модуле.

В приведенном выше примере через объект данного класса можно получить доступ только к функциям **MyClass, SetA и GetA.** Поля FA, B, C и функция F – закрытые элементы. Это вспомогательные данные и функция, которые используют в своей работе открытые функции. Открытая функция **MyClass** с именем, совпадающим с именем класса, это так называемый *конструктор класса,* который должен инициализировать данные в момент создания объекта *класса.* Присутствие конструктора в объявлении класса не обязательно. При отсутствии конструктора пользователь должен сам позаботиться о *задании* начальных значений данным — элементам класса.

Перед именами классов-родителей в объявлении класса также может указываться спецификатор доступа (в примере **public).** Смысл этого спецификатора тот же, что и для элементов класса: при наследовании **public** (открытом наследовании) можно обращаться через объект данного класса к методам и свойствам классов-предков, при наследовании **private** подобное обращение невозможно. Подробнее этот вопрос рассмотрен в разд. 2.14.5.

По умолчанию в классах (в отличие от структур) предполагается спецификатор **private.** Поэтому можно включать в объявление класса данные и функции, не указывая спецификатора доступа. Все, что включено в описание до первого спецификатора доступа, считается защищенным. Аналогично, если не указан спецификатор перед списком классов-родителей, предполагается защищенное наследование.

Объявления данных-элементов (полей) выглядят так же, как объявления переменных или объявления полей в структурах:

```
<тип> <имена полей>;
```

В объявлении класса поля запрещается инициализировать. Для инициализации данных служат конструкторы, о которых упоминалось выше и которые рассматриваются подробно в разд. 2.14.4.

Объявления функций-элементов в простейшем случае не отличаются от обычных объявлений функций (см. гл. 1 разд. 1.7.1).

После того, как объявлен класс, можно создавать объекты этого класса. Если ваш класс не наследует классам библиотеки компонентов C++Builder, то объект класса создается как любая переменная другого типа простым объявлением. Например, оператор

```
MyClass MC, MC10[10], *Pmc;
```

создает объект MC объявленного выше класса MyClass, массив MC10 из десяти объектов данного класса и указатель **Pmc** на объект этого класса.

В момент создания объекта класса, имеющего конструктор, можно инициализировать его данные, перечисляя в скобках после имени объекта значения данных. Например, оператор

```
MyClass MC(3);
```

не только создает объект MC, но и задает его полю **FA** значение 3. Если этого не сделать, то в момент создания объекта поле получит значение по умолчанию, указанное в содержащемся в объявлении класса прототипе конструктора.

Создание переменных, использующих класс, можно совместить с объявлением самого класса, размещая их список между закрывающей класс фигурной скобкой и завершающей точкой с запятой. Например:

```
class MyClass : public Classl, Class2
{
...
} MC, MC10[10], *Pmc;
```

Если создается динамически размещаемый объект класса (см. гл. 1, разд. 1.11), то это делается операцией new. Например:

```
MyClass *PMC = new MyClass;
```

ИЛИ

```
MyClass *PMC1 = new MyClass(3);
```

Эти операторы создают где-то в динамически распределяемой области памяти сами объекты и создают указатели на них — переменные PMC и PMC1.

Создание объектов класса простым объявлением переменных возможно только в случае, если среди предков вашего класса нет классов библиотеки компонентов C++Builder. Если же такие предки есть, то создание указателя на объект этого класса возможно только операцией **new.** Например, если класс объявлен так:

```
class MyClass2 : public TObject
{
 ...
};
```

то создание указателя на объект этого класса может осуществляться оператором

```
MyClass2 *P2 = new MyClass2;
```

2.14.2 Функции-элементы, дружественные функции, константные функции

Поля данных, исходя из принципа скрытия данных, всегда должны быть защищены от несанкционированного доступа. Доступ к ним, как правило, должен осуществляться только через функции, включающие методы чтения и записи полей. В этих функциях должна осуществляться проверка данных, чтобы не записать случайно в поля неверные данные или чтобы не допустить их неверной трактовки. Кроме того, функции чтения позволят вам не переписывать использующую их программу, даже если вы решили изменить что-то в типе, способах хранения и размещения данных в классе.

Поэтому данные всегда целесообразно объявлять в разделе **private** — закрытом разделе класса. В редких случаях их можно помещать в **protected** — защищенном разделе класса, чтобы возможные потомки данного класса имели к ним доступ.

Приведем пример. Пусть класс имеет следующее объявление:

```
class MyClass
{
public:
  void SetA(int);    // функция записи
  int GetA(void);    // функция чтения
private:
  int FA;
  double B, C;
};
```

Реализация функций записи и чтения может иметь вид:

```
void MyClass::SetA(int Value)
{
 if (...)          // проверка корректности данных
   FA = Value;
}

int MyClass::GetA(void) {return FA;}
```

В данном случае функция чтения просто возвращает значение поля, но в более сложных классах может потребоваться какая-то предварительная обработка данных. Обратите внимание, что все описания функций-элементов содержат ссылку на класс с помощью операции разрешения области действия "::".

В приведенном примере объявление класса содержит только прототипы функций, а их реализация вынесена из описания класса. Для простых функций реализация может быть размещена непосредственно в объявлении класса. Например:

```
class MyClass
{
public:
```

```
  MyClass(int = 0);
  void SetA(int Value) {FA= Value;};    // функция записи
  int GetA(void) {return FA;};          // функция чтения
private:
  int FA;
  double B, C;
};
```

Функции, описание которых содержится непосредственно в объявлении класса, в действительности являются встраиваемыми функциями **inline** (см. разд. 1.7.6, в котором обсуждаются достоинства и недостатки таких функций).

Введение описания функций в объявление класса — это плохой стиль программирования: следует избегать смешения открытого интерфейса класса, содержащегося в его объявлении, и реализации класса. Если уж вы хотите реализовать встраиваемые функции, то лучше поместить в объявлении класса их прототип со спецификатором **inline:**

```
inline void SetA(int);    // функция записи
```

и отдельно дать реализацию функции. При этом в реализации спецификатор **inline** не указывается.

Объявления классов следует размещать в заголовочном файле модуля, а реализацию функций — элементов в отдельном файле реализации. При этом в объявлении класса должны содержаться только прототипы функций. Это следует из принципа скрытия информации — одного из основных в объектно-ориентированном программировании. Такая организация программы обеспечивает независимость всех модулей, использующих заголовочный файл с объявлением класса, от каких-то изменений в реализации функций-элементов класса.

Функции-элементы класса имеют доступ к любым другим функциям-элементам и к любым данным-элементам, как открытым, так и закрытым. Клиенты класса (какие-то внешние функции, работающие с объектами данного класса) имеют доступ только к открытым функциям-элементам и данным-элементам. Но в некоторых случаях желательно обеспечить доступ к закрытым элементам для функций, не являющихся элементами данного класса. Это можно сделать, объявив соответствующую функцию как *друга класса с* помощью спецификации **friend.** Например, если в объявление класса включить оператор

```
friend void IncFA(MyClass *);
```

то функция **IncFA,** не являясь элементом данного класса, получает доступ к его закрытым элементам. Например, функция **IncFA** может быть описана где-то в программе следующим образом:

```
void IncFA(MyClass *P) {P->FA++;}
```

Дружественными могут быть не только функции, но и целые классы. Например, вы можете поместить в объявление своего класса оператор

```
friend Classl;
```

и все функции-элементы класса **Classl** получат доступ к закрытым элементам вашего класса.

Иногда программист может захотеть создать объект вашего класса как константный с помощью спецификатора **const.** Например:

```
const Classl MC1(3);
```

Если при этом ваш класс содержит не только функции чтения, но и записи данных, то реакция на такой оператор, введенный пользователем, зависит от версии и настройки компилятора. Компилятор может выдать сообщение об ошибке и отказаться от компиляции, а может просто выдать предупреждение и проигнорировать спецификатор пользователя **const.** Если же ваш класс содержит только функции чтения, то все должно бы быть нормально. Но компилятор подойдет

к этому чисто формально и все равно выдаст предупреждение, а может и отказаться компилировать программу.

Чтобы избежать этого, можно объявить функции чтения как константные. Для этого и в прототипе, и в реализации после закрывающей список параметров круглой скобки надо написать спецификатор **const.** Например, вы можете включить в объявление класса оператор

```
int GetA(void) const;
```

а реализацию этой функции оформить как:

```
int MyClass::GetA(void) const {return FA;}
```

Тогда неприятные замечания компилятора о константных объектах исчезнут.

Таким образом, если предполагается, что объект вашего класса может быть объявлен константным, снабжайте все функции-элементы класса, предназначенные для чтения данных, спецификаторами **const.**

2.14.3 Данные-элементы, статические данные, константные данные

Теперь рассмотрим несколько подробнее данные-элементы. Обычно каждый объект класса имеет свою собственную копию всех данных-элементов класса. Но в определенных случаях во всех объектах класса должна фигурировать только одна копия некоторых данных. Например, это может быть счетчик числа созданных объектов класса.

Единственную копию данных полезно иметь и во многих иных случаях. Например, если в классе имеются некоторые константы, одинаковые для всех объектов класса, то нерационально хранить в каждом объекте собственные копии этих констант. Рациональнее иметь единственные экземпляры этих констант для всех объектов.

Для введения в класс подобных данных используются статические данные, которые содержат информацию «для всего класса». Объявление статических элементов в классе начинается с ключевого слова **static.** Например:

```
static int D;
```

Статические элементы могут быть открытыми, закрытыми или защищенными **(protected).** Доступ к открытым статическим элементам класса возможен посредством любого объекта класса или посредством имени класса с помощью бинарной операции разрешения области действия. Например:

```
MyClass::D = 10;
```

Закрытые и защищенные статические элементы класса должны быть доступны открытым функциям-элементам этого класса или друзей класса.

Статические элементы класса существуют даже тогда, когда не существует никаких объектов этого класса. В этом случае доступ к открытому статическому элементу обеспечивается так же, как указано выше: с помощью имени класса и бинарной операции разрешения области действия. Для обеспечения доступа в отсутствие объектов к закрытому или защищенному элементу класса должна быть предусмотрена открытая статическая функция-элемент, которая должна вызываться с добавлением перед ее именем имени класса и бинарной операции разрешения области действия.

Начальные значения статических элементов (как открытых, так и закрытых) должны задаваться вне объявления класса. Для этого достаточно разместить где-то в файле, например, после объявления класса или среди реализаций функций-элементов (но не внутри их) оператор вида

```
int MyClass::D = 0;
```

Статическим данным-элементам можно задать начальные значения один и только один раз в области действия файл. Если вы нигде не инициализировали статический элемент данных, будет выдано сообщение компилятора о неразрешенной внешней ссылке и программа не будет скомпилирована. Если вы дважды инициализируете статический элемент, будет выдано сообщение о дублировании инициализации и программа также не будет скомпилирована.

Приведем пример, демонстрирующий все сказанное относительно статических данных-элементов:

```
class MyClass
{
public:
  static int D;
  static int GetD1(void);
  ...
private:
  static int D1;
...
};
...
int  MyClass::GetD1(void)  {return D1;}

int MyClass::D = 0;
int MyClass::D1 = 1;
```

В этом примере имеются два статических элемента данных: открытый D и закрытый **D1.** Если пользователь должен иметь возможность получать значение закрытой статической переменной **D1,** то должна быть предусмотрена функция ее чтения, названная в примере **GetDl.** Она должна быть объявлена открытой **(public)** и статической (со спецификатором **static).** Статической может быть объявлена любая функция, работающая только со статическими данными.

После объявления класса в примере расположена реализация функции **GetD1.** В реализации не требуется указывать спецификатор **static.** Далее приведены предложения, инициирующие открытые и закрытые статические данные. На этом все, связанное с объявление и инициализацией статических данных завершается. В дальнейшем вы можете из любой внешней функции обращаться к ним с помощью операции разрешения области действия. Например:

```
i = MyClass::D;
j = MyClass::GetD1();
```

Среди данных — элементов могут быть объявлены именованные константы. Например:

```
static const int MaxA = 10;
const int MinA;
```

Значения статических именованных констант могут задаваться в момент их объявления в классе, как показано в предыдущем примере. Инициализация нестатических констант — вопрос более сложный, связанный с построением конструкторов. Он рассматривается в разд. 2.14.4.

2.14.4 Конструкторы и деструкторы

Остановимся теперь на конструкторах класса. Прежде всего отметим, что наличие конструктора в классе не обязательно. Но если конструктор отсутствует, то клиенты класса (внешние функции, использующие класс) должны сами заботиться об инициализации данных, т.е. о задании им некоторых начальных значений. Это не всегда возможно. Например, если класс имеет закрытые данные, предназначенные только для чтения, то для этих данных не предусматриваются открытые

функции записи. И клиент не в состоянии присвоить данным какие-то начальные значения.

Конструктором класса называется открытая функция-элемент, которая вызывается в момент создания объекта класса и должна инициализировать данные указанными в вызове значениями или значениями по умолчанию. Конструктор имеет то же имя, что и сам класс.

Пример объявления и реализации конструктора:

```
class MyClass
{
public:
  MyClass(void); // конструктор класса
  ...
private:
  int A;
...
 };
...

MyClass::MyClass(void) {A = 0;}
```

В этом примере объявлен конструктор **MyClass** без параметров, который при создании объекта задает начальное значение поля A равным 0. Обратите внимание на то, что в отличие от других функций в объявлении конструктора не указывается тип возвращаемого значения.

Простое задание в конструкторе значений данных в общем случае не гарантирует их целостность. Обычно нужна еще проверка допустимости данных. Например, если в классе есть функция записи SetA, осуществляющая такие проверки, то лучше обратиться к ней и при задании начального значения. В этом случае реализация конструктора может иметь вид:

```
MyClass::MyClass(void) { SetA(0); }
```

Создание объекта описанного класса MyClass в программе должно осуществляться или объявлением соответствующей переменной:

```
MyClass MC;
```

или динамическим размещением переменной в памяти:

```
MyClass *PMC = new MyClass;
```

В момент выполнения каждого из этих операторов неявным образом вызывается конструктор, устанавливающий начальные значения данных.

Недостатком конструкторов показанного типа является то, что все начальные значения данных задаются в них конструктором. Вызывающая функция никак не может вмешаться в этот процесс и задать какое-то другое значение.

Другой крайностью являются конструкторы, в которых все начальные значения задаются как параметры. Например, прототип конструктора может иметь вид

```
MyClass(int);
```

а его реализация:

```
MyClass::MyClass(int a) ( SetA(a); }
```

В этом случае поле FA инициализируется параметром, передаваемым в конструктор. Создание объекта подобного класса должно выполняться операторами

```
MyClass MC(1);
```

или

```
MyClass *PMC = new MyClass(1);
```

в которых подразумевается, что начальное значение поля FA должно быть равно 1.

Такой конструктор обычно тоже неудобен, поскольку в классе может быть много параметров и задавать значения их всех при создании объекта очень громоздко и чревато ошибками.

Чаще всего используются конструкторы с параметрами по умолчанию (см. разд. 1.7.4). В этом случае объявление конструктора может иметь вид:

```
MyClass (int = 0);
```

а его реализация:

```
MyClass::MyClass(int a) { SetA(a); }
```

Объект такого класса можно создавать любым из приведенных ранее операторов создания объекта. Если при создании указывается аргумент, то его значение присваивается полю. Если аргумент не указывается, то присваивается значение по умолчанию (в нашем примере 0). Этот вариант конструктора наиболее гибкий. Поэтому он чаще всего используется при построении классов.

В объявлении класса могут быть определены не только поля переменных, но и некоторые именованные константы. Например:

```
const int MaxA;
```

Подобная константа может служить, в частности, предельно допустимым значением поля FA.

Если такая константа объявлена как статическая (см. разд. 2.14.3), то в ее объявление в классе можно непосредственно включить инициализацию:

```
static const int MaxA = 10;
```

Но тогда это значение клиент при желании не сможет изменить. А задать значение такой константы в конструкторе невозможно, поскольку компилятор не разрешает присваивать значения константам. Выходом из положения является специальный синтаксис конструктора с *инициализатором элементов*. Инициализатор элементов записывается после заголовка конструктора в его реализации, предваряется двоеточием и содержит имена константных данных, после которых в скобках указываются их значения. Например, если в объявлении вашего класса MyClass имеются строки

```
const int MaxA;
const int MinA;
```

вводящие две константы — максимальное и минимальное значения переменной A, то реализацию конструктора такого класса с описанным ранее прототипом

```
MyClass(int = 0);
```

надо дополнить инициализатором элементов:

```
MyClass::MyClass(int a) : MaxA(10), MinA(1) {SetA(a);};
```

В данном случае инициализатор задает константе MaxA начальное значение 10, а константе MinA — значение 1.

Можно предоставить пользователю возможность изменять значения констант в момент создания объекта. В этом случае в конструкторе с умолчанием надо предусмотреть для констант соответствующие значения по умолчанию:

```
MyClass(int A = 0, int MaxA = 10, int MinA = 1);
```

или

```
MyClass(int = 0, int = 10, int = 1);
```

(второй вариант менее удобен, так как не позволяет по прототипу функции понять, в какой последовательности должны задаваться параметры).

Тогда реализацию конструктора можно оформить так:

```
MyClass::MyClass(int a, int i, int j) : MaxA(i), MinA(j) { SetA(a); }
```

Создание объектов такого типа может осуществляться, например, такими операторами:

```
MyClass MC;          // умолчание: A = 0,  MaxA = 10, MinA = 1
MyClass MC(20);      // задано:    A = 20, MaxA = 10, MinA = 1
MyClass MC(20,15);   // задано:    A = 20, MaxA = 15, MinA = 1
MyClass MC(20,15,2); // задано:    A = 20, MaxA = 15, MinA = 2
```

Конструкторы глобальных переменных вызываются в самом начале выполнения приложения, до того, как будет вызвана функция **WinMain** или **main.** Если директивами **#pragma startup** (см. разд. 1.4.4) указаны функции, которые должны вызываться до **WinMain** или **main,** то конструкторы глобальных переменных вызываются ранее этих функций.

Теперь остановимся на деструкторах. Это специальные функции-элементы, срабатывающие при уничтожении динамически размещенного объекта класса и освобождающие занимаемую им память. Имя деструктора совпадает с именем класса, но перед ним записывается символ тильда "~". Как и для конструктора, в деструкторе не указывается возвращаемый тип. Например:

```
class MyClass
{
public:
  ~MyClass(); // деструктор класса
  ...
};
```

Деструкторы необходимы, если конструктор или какие-то функции-элементы класса динамически распределяют память, создавая в ней какие-то объекты. Тогда деструктор должен эти объекты удалять. В остальных случаях можно обычно обойтись без деструктора.

Если деструктор явным образом в классе не объявлен, компилятор сам генерирует необходимые коды освобождения памяти.

2.14.5 Копирование объектов классов

Объекты классов можно копировать. Это осуществляется с помощью конструктора копии. Например, если y - объект класса X, то выражение

```
X x(y);
```

объявляет переменную x того же класса X и вызывает конструктор копии, который заносит в x копию объекта y. То же самое можно сделать оператором, использующим операцию присваивания,

```
X x = y;
```

или

```
X x = X(y);
```

Во всех случаях значение x становится равным значению y, но в то же время x и y - это разные объекты. В дальнейшем их значения могут изменяться независимо друг от друга.

Конструктор копии работает также при указании объекта в качестве аргумента функции, если используется передача его по значению.

Если соответствующий конструктор не объявлен, то C++Builder, когда это требуется, автоматически генерирует конструктор копии. По умолчанию копирование сводится к побитовому копированию данных одного объекта в другой. В результате будут скопированы все поля исходного объекта. Для простых объектов этого достаточно. Но если, например, данные содержат какие-то указатели, то побитовое копирование, естественно даст неверный результат, так как указатели ко-

пии будут указывать на данные источника, а не копии. В подобных случаях надо определять в классе конструктор копии. Необходимо это делать и в случаях, когда копирование сопровождается приведением типов, т.е. когда типы источника и приемника не тождественны друг другу. Создавать собственный конструктор копии надо также, если вы перегружаете операцию присваивания.

Построим простой пример, чтобы разобраться в работе конструкторов. Опишите простой класс:

```
class X
{
 public:
 int a;
 X(int = 0);
};
```

Класс имеет одно поле a и конструктор с умолчанием, реализацию которого можно записать так:

```
X::X(int A)
{
 a = A;
 ShowMessage("конструктор с умолчанием, a = " + IntToStr(a));
)
```

Мы ввели в конструктор вызов диалогового окна с сообщением, чтобы знать, когда и сколько раз он вызывается.

Теперь в обработчике щелчка на какой-то кнопке вы можете, например, написать операторы:

```
X y1;
X y2 = X(2) ;
X y3 = 3.5;
X x1(y1);
X x2 = y2;
X x3 = X (y3) ;
ShowMessage(IntToStr(x1.a) + ' ' +
            IntToStr(x2.a) + ' ' + IntToStr(x3.a));
```

Если вы выполните ваше приложение и щелкните на кнопке, то увидите, что конструктор вызывается три раза со значениями a, равными 0, 2 и 3. Это происходит при создании объектов **yl,** y2 и y3. Для yl срабатывает значение параметра по умолчанию - 0. Для y2 значение 2 задается при явном вызове конструктора. А вот для y3 вступает в строй «интеллект» компилятора. Компилятор видит, что конструктор может принять целый параметр. Он приводит заданное число 3.5 к целому (естественно, округляя) и передает его в конструктор.

При создании объектов **xl,** x2 и x3 конструктор не вызывается. Компилятор автоматически генерирует для всех этих объектов конструктор копии, неявно вызывает его и копирует поля переменных yl, y2, y3 в поля переменных **xl, x2, x3.**

В данном случае все прекрасно. Но в более сложных случаях подобное творчество компилятора может приводить к ошибкам. Так что иногда его требуется запретить. Это можно сделать, добавив перед объявлением конструктора в классе ключевое слово **explicit** (явный):

```
explicit X(int = 0);
```

Тем самым вы запрещаете неявный вызов конструктора. Если теперь вы попробуете откомпилировать приложение, компилятор выдаст сообщение об ошибке в операторе, создающем объект y3. Смысл его сводится к тому, что он не может привести действительное число к типу X. Точнее, мы запретили ему делать это.

Впрочем, если вы закомментируете операторы, связанные с переменными y3 и x3, то увидите, что все остальное работает нормально и компилятор по-прежнему неявно создает и вызывает конструкторы копии.

Выше говорилось, что неявное побитовое копирование в ряде случаев может приводить к ошибкам. В этих случаях надо явно определять в классе конструктор копии, передавая в него ссылку на данный класс. В нашем случае это может иметь вид:

```
class X
{
 public:
 int a, b;
 X(int = 0);
 X(X&);
};
```

Реализацию конструктора копии сделайте такой:

```
X::X(X& x)
{
 a = x.a;
 ShowMessage("конструктор копии, a = " + IntToStr(a));
}
```

В этой реализации опять предусмотрено сообщение, чтобы можно было убедиться, что работает именно этот конструктор.

Выполнив теперь приложение, вы сможете убедиться, что после трех вызовов конструктора с умолчанием следуют три вызова вашего конструктора копии. Он вызывается при создании объектов **x1**, **x2,** x3.

2.14.6 Наследование и полиморфизм, виртуальные функции, абстрактные классы

При описании нового класса, производного от какого-то одного или нескольких базовых классов, можно добавлять новые функции-элементы и данные-элементы, сохраняя при этом все элементы родителей, а можно родительские элементы переопределить или перегрузить. В производном классе доступны открытые и защищенные элементы базового класса (прямого или косвенного предшественника). Закрытые элементы базового класса в производном классе недоступны.

Производный класс может наследоваться от базового класса как **public, protected** или **private** (см. синтаксис такого наследования в разд. 2.14.1). Защищенное и закрытое наследования встречаются редко и каждое из них нужно использовать с большой осторожностью.

При порождении класса как **public** открытые элементы базового класса становятся открытыми элементами производного класса, а защищенные элементы базового класса становятся защищенными элементами производного класса. Закрытые элементы базового класса никогда не бывают доступны для производного класса.

При защищенном наследовании открытые и защищенные элементы базового класса становятся защищенными элементами производного класса. При закрытом наследовании открытые и защищенные элементы базового класса становятся закрытыми элементами производного класса. При закрытом и защищенном наследованиях не справедливо отношение, что объект производного класса является объектом базового класса.

В целом доступ к элементам базового класса из производного класса можно представить следующей таблицей.

	Тип наследования		
Спецификатор доступа к элементам в базовом классе	public открытое наследование	protected защищенное наследование	private закрытое наследование
public	public в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам, дружественным функциям и функциям, не являющимся элементами.	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	private в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.
protected	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	private в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.
private	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.

Если в классе-наследнике переопределить функцию-элемент (ввести новую функцию с тем же именем), то для объектов этого класса новая функция отменит родительскую. Если обращаться к объекту этого класса, то вызываться будет новая функция. Если все-таки нужно вызвать именно функцию базового класса, надо использовать операцию разрешения области действия.

Создайте в качестве примера приложение с классом форм **Shape:**

```
class Shape
{
public:
 void Draw(void);
};
```

и наследующим ему классом кругов **Circl:**

```
class Circl : public Shape
{
```

```
public:
 void Draw(void);
};
```

В каждом из классов вы объявили метод рисования **Draw.** Чтобы не усложнять задачу и не отвлекаться на методы рисования, ограничимся в реализации этих методов сообщением о том, какая фигура рисуется. Тогда реализация методов **Draw** может иметь вид:

```
void Shape::Draw(void)
{
 ShowMessage("Абстрактная фигура");
}
void Circl::Draw(void)
{
 ShowMessage("Круг");
}
```

А теперь введите в приложение кнопку и в обработчик щелчка на ней занесите код:

```
Shape *PQ1 = new Shape;
PQ1->Draw();              // вызов Draw класса Shape
Circl *PQ2 = new Circl;
PQ2->Draw();              // вызов Draw класса Circl
PQ2->Shape::Draw();       // вызов Draw класса Shape
((Shape *)PQ2)->Draw();   // вызов Draw класса Shape
```

В комментариях к коду указано, функции **Draw** каких классов вызывают эти операторы. Впрочем, выполнив приложение, вы сами увидите подтверждение этого. Если мы обращаемся через указатель к самому объекту типа **Circl,** то вызывается переопределенная в нем функция. Но если мы с помощью приведения типа обращаемся к нему как к объекту базового класса (последний оператор), или соответствующим образом используем операцию разрешения области, то вызывается функция базового класса.

Если бы в классе-наследнике **Circl** отсутствовала функция **Draw,** то все приведенные операторы вызывали бы функцию базового класса.

Таким образом, механизм наследования позволяет использовать функции базового класса или переопределять их.

Теперь рассмотрим другую задачу. Создайте несколько классов, наследующих **Shape.** К уже имеющемуся у вас классу **Circl** добавьте аналогичные классы **Rectang** (прямоугольник - не назовите его случайно **Rectangle,** так как возникнет путаница с одноименным методом канвы) и **Square** (квадрат). Каждый из этих классов имеет свою функцию **Draw,** которая умеет рисовать соответствующую форму (для упрощения замените в новых классах рисование выдачей соответствующего сообщения, как сделали это раньше). Мы хотим работать с объектами этих фигур, как с объектами базового класса **Shape,** не разбираясь в истинной природе каждого объекта. И при этом хотим, чтобы программа сама понимала, что это за объект и как его рисовать. Например, введите в модуль глобальную переменную, являющуюся массивом указателей на объекты различных форм:

```
Shape *ShapeArray[3];
```

Загрузите его в обработчике события **OnCreate** формы указателями на объекты разных фигур:

```
ShapeArray[0] = new Circl;
ShapeArray[1] = new Rectang;
ShapeArray[2] = new Square;
```

А теперь добавьте в приложение кнопку с обработчиком вида:

```
for(int i = 0; i < 3; i++)
   ShapeArray[i]->Draw();
```

Мы хотим, чтобы в цикле рисовались фигуры объектов, хранящихся в массиве.

Рассмотренный ранее механизм наследования такую задачу решить не может. Поскольку ко всем объектам мы обращаемся через тип их базового класса **Shape,** то только функция этого класса и будет вызываться.

Поставленную задачу *полиморфизма* позволяют решить виртуальные функции. Они не связаны с другими функциями с тем же именем в классах — наследниках. Если в классах — наследниках эти функция переопределены, то при обращении к такой функции во время выполнения будет вызываться та из виртуальных функций с одинаковыми именами, которая соответствует классу объекта, указанного при вызове. Поэтому, если в базовом классе **Shape** объявить функцию **Draw** как виртуальную, то задача будет решена и каждая фигура будет рисоваться своей функцией.

Синтаксически это оформляется следующим образом. В базовом классе **Shape** функция объявляется следующим образом:

```
virtual void Draw(void);
```

И это все! Если функция была однажды объявлена виртуальной, она остается виртуальной и во всех классах наследниках. Таким образом для решения задачи полиморфизма хватило одного спецификатора **virtual.** Правда, обычно предпочитают для большей ясности программы в классах-наследниках тоже вводить спецификатор **virtual,** чтобы была ясна суть этих функций для тех, кто будет строить наследников данного класса. Но с точки зрения языка C++ это не обязательно.

Выполните теперь ваше тестовое приложение. Вы убедитесь, что каждый объект использует свою собственную функцию рисования.

Подобное полиморфное поведение можно оформить совершенно иначе, воспользовавшись описанной в разд. 2.2 операцией явного динамического приведения типов **dynamic_cast.** В этом случае можно воспользоваться одной универсальной функцией в базовом классе и в ней определять, объект какого реального типа к ней обратился. Тогда в классах-наследниках эту функцию можно не переопределять. Но функция в базовом классе по-прежнему должна быть объявлена виртуальной.

Измените ваше тестовое приложение, чтобы опробовать этот механизм. Удалите (закомментируйте) из классов **Circl, Rectang и Square** объявления функции **Draw** и, соответственно, удалите реализации этих функций. А реализацию **Draw** в классе **Shape** оформите следующим образом:

```
if (dynamic_cast<Circl *>(this) != 0)
  ShowMessage("Круг");
else if (dynamic_cast<Rectang *>(this) != 0)
  ShowMessage("Прямоугольник");
else if (dynamic_cast<Square *>(this) != 0)
  ShowMessage("Квадрат");
else ShowMessage("Абстрактная фигура");
```

В этом коде с помощью **dynamic_cast** поочередно осуществляется приведение класса указателя **this** к типу указателей различных классов-наследников. Ключевое слово **this** определяет указатель на тот объект, из которого была вызвана функция. Если приведение типа завершилось успешно, операция **dynamic_cast** возвращает ненулевое значение. Тогда рисуется соответствующая фигура (в нашем примере просто выдается соответствующее сообщение). Если же приведение типа закончилось неудачей, операция **dynamic_cast** возвращает 0. В этом случае функция переходит к проверке указателя на принадлежность его следующему типу.

Выполнив приложение, вы можете убедиться, что полиморфное поведение сохраняется.

Рассмотренный вариант компактнее предыдущего, так как требует описания всего одной функции в базовом классе. Но он менее универсален. При создании новых классов-наследников надо будет добавлять соответствующие коды в функцию

базового класса. А в первом варианте базовый класс не затрагивается. Просто, надо написать соответствующую реализацию функции в новом классе-наследнике.

Иногда в базовом классе определяют *чистую виртуальную функцию* (абстрактную функцию). Это функция, для которой не указана реализация. Для того чтобы определить такую функцию, достаточно указать, что ее тело равно нулю:

```
virtual void Draw(void)=0;
```

В первом варианте нашего примера именно так было бы целесообразно объявить функцию **Draw** в базовом классе **Shape,** поскольку непонятно, как можно нарисовать просто абстрактную фигуру. Реализация для чистой полиморфной функции не пишется.

Класс, в котором имеется хоть одна чистая виртуальная функция, называется *абстрактным.* Для абстрактного класса невозможно создать объект. Такие классы предназначены только для построения на их основе конкретных классов-наследников.

2.14.7 Особенности классов, наследующих классам библиотеки компонентов C++Builder

2.14.7.1 Свойства

О некоторых особенностях построения классов, наследующих классам библиотеки компонентов C++Builder, уже говорилось ранее. К этим особенностям относится невозможность для таких классов множественного наследования и необходимость создавать объекты только с помощью операции **new.** Теперь остановимся на других особенностях, связанных с понятиями *свойства* и *события.*

Понятие свойства **(property),** объединяет поле данных и функции (методы) его записи и чтения. В рассматриваемых классах сами поля объявляются как обычно, но, как правило, в разделе **private.** Традиционно идентификаторы полей совпадают с именами соответствующих свойств, но с добавлением в качестве префикса символа 'F'.

Свойство объявляется оператором вида:

```
__property <тип> <имя> = {read=<имя поля или метода чтения>
                          write=<имя поля или метода записи>
                          <директивы запоминания
                           и значения по умолчанию>;
```

Если в разделах **read** или **write** этого объявления записано имя поля, значит предполагается прямое чтение или запись данных.

Если в разделе **read** записано имя метода чтения, то чтение будет осуществляться только функцией с этим именем. Функция чтения — это функция без параметра, возвращающее значение того типа, который объявлен для свойства. Имя функции чтения принято начинать с префикса Get, после которого следует имя свойства.

Если в разделе **write** записано имя метода записи, то запись будет осуществляться только процедурой с этим именем. Процедура записи — это процедура с одним параметром того типа, который объявлен для свойства. Имя процедуры записи принято начинать с префикса Set, после которого следует имя свойства.

Если раздел **write** отсутствует в объявлении свойства, значит это свойство только для чтения и пользователь не может задавать его значение.

Директивы запоминания определяют, как надо сохранять значения свойств при сохранении пользователем файла формы **.dfm.** Чаще всего используется директива

```
default <значение по умолчанию>
```

Она не задает начальное значение. Это дело конструктора. Директива просто говорит, что если пользователь в процессе проектирования не изменил значение свойства по умолчанию, то сохранять значение свойства не надо.

Приведем пример. Пусть требуется объявить класс с именем **MyClass,** наследующий непосредственно **TObject** и имеющий свойство целого типа с именем A. Тогда объявление этого класса может иметь вид:

```
class MyClass1 : public TObject
{
 private:
  int FA;
 protected:
  void__fastcall SetA(int); // функция записи
 published:
  __property int A = {read = FA, write = SetA, default = true};
};
```

Здесь вводится закрытое поле **FA,** объявляется защищенная функция SetA, используемая для записи значения в это поле, и вводится опубликованное свойство A, оперирующее этим полем. В объявлении свойства после ключевого слова **read** записано просто имя поля. Это означает, что функция чтения отсутствует и пользователь может читать непосредственно значение поля. После ключевого слова **write** следует ссылка на функцию записи SetA, с помощью которой будут записываться в поле A новые значения. В этой функции можно предусмотреть какие-то проверки допустимости вводимого значения A.

Описание этой функции может иметь вид:

```
void__fastcall MyClass1::SetA(int Value)
{
 if(...) FA = Value;
)
```

В приведенном примере описание свойства A помещено в раздел **published.** Следовательно, если этот класс описывает создаваемый вами новый компонент, то после его установки в систему свойство A будет появляться в окне Инспектора Объектов при использовании этого компонента. Если перенести объявление свойства в раздел **public,** то свойством по-прежнему можно будет пользоваться, но только во время выполнения приложения, поскольку в окне Инспектора Объектов оно появляться не будет. Если удалить из определения свойства слово **write** с последующей ссылкой на функцию записи, то свойство станет свойством только для чтения, т.к. изменить его непосредственно будет невозможно.

Для свойств типа массивов приведенный ранее оператор **__property** изменяется следующим образом:

```
__property <тип> <имя> <список размерностей> =
                       {read=<имя поля или метода чтения>
                        write=<имя поля или метода записи>
                        <директивы запоминания
                        и значения по умолчанию>;
```

Список размерностей представляет собой последовательность квадратных скобок, в которых записывается тип размерности и может записываться идентификатор. Приведем в качестве примера возможный вариант описания класса матриц действительных чисел размером N x M:

```
// Класс матриц действительных чисел
class Matrix
{
   float *data;
   int N;    // число строк
   int M;    // число столбцов
```

```
public:
   Matrix(int,int);
   ~Matrix( ) { delete[ ] data; }
   __property float Items [int i] [int j] =
                          { read=GetItems, write=SetItems };
private:
   float___fastcall GetItems(int i, int j);
   void___fastcall SetItems (int i, int j, float value);
};

Matrix::Matrix(int n, int m)    // конструктор
{
  data = new float [n*m];       // создание экземпляра класса
  for(int i = 0; i < n*m; i++) // инициализация
    data[i] = 0.;
  N = n;
  M = m;
}

void___fastcall Matrix::SetItems(int i, int j, float value)
{
// запись значения value в элемент (i,j)
  if ( (i<l) I I (i>N) I I (j<l) I I (j>M))
    ShowMessage("Недопустимые индексы (" + IntToStr(i) +
                ", " + IntToStr (j) + ")");
  else data[(i-l) * M + j - 1] = value;
}

float___fastcall Matrix: :GetItems (int i, int j)
{
// чтение значения элемента (i,j)
  if((i<1) I I (i>N) | | (j<l) | | (j>M))
    ShowMessage("Недопустимые индексы (" + IntToStr(i) +
                ","+ IntToStr(j) + ")");
  else return data[(i-1) * M + j - 1];
}
```

В приведенном коде создается класс матриц Matrix. Класс имеет открытое свойство **Items,** к которому можно обращаться как к двумерному массиву. Об этом говорит его определение в операторе **property: float Items [int i] [int j].** Указание в списке размерностей идентификаторов i и j не является обязательным. Список мог бы иметь вид: **[int] [int].** •

Задание размерностей изменяет вид функций чтения и записи. В функцию чтения **GetItems** передаются два целых параметра, определяющих индексы читаемого элемента матрицы. В приведенном примере индексы матриц отсчитываются от 1, а не от нуля, что, вероятно, более удобно пользователю. В функцию записи **SetItems** помимо записываемого значения **value** также передаются индексы того элемента, в который должно быть записано это значение.

Создание экземпляра матрицы в программе может, осуществляться, например, оператором:

```
Matrix x(4,5);
```

Этот оператор создает матрицу x размерностью 4 x 5. Запись и чтение элементов матрицы в программе осуществляется через свойство **Items.** Например:

```
x.Items[2][3] = 1.5;
float y = x.Items[2][3];
```

Первый из этих операторов заносит значение 1,5 в 3-ий элемент 2-ой строки, а второй оператор читает это значение.

2.14.7.2 События

Событие - - это специальное свойство, являющееся указателем функции. В C++Builder тип обобщенного указателя на функцию, которой передается один параметр типа **TObject** (обычно this), — **TNotifyEvent.** Подобный тип используется в C++Builder для событий типа **OnClick** и многих других, которые передают в обработчик только один параметр - - **TObject *Sender.** Если требуется ввести в класс подобное событие, достаточно определить в объявлении класса соответствующее поле и метод работы с ним. Например:

```
private:
  ...
  TNotifyEvent  FMyEvent;
  ...
_ published:
  ...
  _ property TNotifyEvent MyEvent = {read= FMyEvent,
                                     write= FMyEvent};
```

Остается только вызвать в нужный момент обработчик событий пользователя, если пользователь его предусмотрел. Проверка, имеется ли обработчик пользователя, осуществляется проверкой соответствующего события как булевой величины, возвращающей **true,** если пользователь предусмотрел свой обработчик. Значит, при возникновении события надо проверять, имеется ли обработчик пользователя, и, если имеется, то вызывать его. Для этого можно использовать оператор вида:

```
if(FMyEvent) OnMyEvent(this);
```

Функция **OnMyEvent,** которая вызывается этим оператором, это и есть обработчик пользователя. Его имя совпадает с именем свойства, перед которым добавляется префикс "On".

Место, куда надо включать подобный оператор, зависит от вида события. Если событие вызывается каким-то из ваших методов, то вызов обработчика пользователя надо осуществлять из этого метода. Если событие связано с какими-то сообщениями, поступающими от других приложений или от Windows, то надо предусмотреть обработчик соответствующего сообщения и из него вызывать обработчик пользователя.

Если в обработчик события надо передать какие-то параметры помимо **this,** то тип функции **TNotifyEvent** уже не подходит и надо объявить свой собственный тип. Это объявление делается с помощью ключевого слова **_ closure.** Например:

```
typedef void _fastcall (_closure *TMyEvent)
                  (System::TObject *Sender, bool& MyParam);

class T : public TObject
{
 private
   TMyEvent FMyEvent;
 published
   _ property TMyEvent FMyEvent = {read= FMyEvent,
                                   write= FMyEvent};
   ...
}
```

Выше было рассмотрено введение в класс какого-то нового события. Если же вам надо переопределить одно из традиционных событий, связанных с клавиатурой, мышью и т.п., то это можно сделать, переопределив соответствующий стандартный обработчик родительского класса.

2.14.8 Шаблоны классов

C++ позволяет определять шаблоны классов, называемые также родовыми (generic) классами или генераторами классов. Иногда их называют параметризованными типами, так как они имеют один или большее количество параметров типа, определяющих настройку шаблона класса на специфический тип данных при создании объекта класса.

Для того чтобы использовать шаблонные классы, программисту достаточно один раз описать шаблон класса. Каждый раз, когда требуется реализация класса для нового типа данных, программист, используя простую краткую запись, сообщает об этом компилятору, который и создает исходный код для требуемого класса.

Шаблоны классов задаются аналогично шаблонам функций (см. разд. 1.7.8). Описание шаблона отличается от описания класса первой строкой

```
template <class идентификатор> class имя класса
```

В этой строке **идентификатор** является произвольным именем формального типа, который используется далее в описании шаблона. Но негласно принято, если нет каких-то иных соображений, задавать в качестве имени типа "T". Например:

```
template <class T> class Matrix
{
...
};
```

Этот заголовок объявляет о создании шаблона класса **Matrix** и задает идентификатор T для формального типа данных. Этот идентификатор следует использовать в описании класса вместо указания типа соответствующих данных.

Определяемый в заголовке идентификатор типа (в приведенном примере — T) совершенно не обязательно должен быть классом. Пользователь может указать при вызове шаблона любой тип. Например, **float.** Так что спецификатор **class,** предшествующий имени типа в заголовке шаблона, является некоторым анахронизмом, искажающим истинную сущность идентификатора. В современном варианте C++ вместо этого спецификатора можно задавать ключевое слово **typename.** Например:

```
template <typename T> class Matrix
{
...
};
```

Не все компиляторы пока поддерживают ключевое слово **typename** (компилятор C++Builder 6 поддерживает), так что для общности стандартные библиотеки пока обычо используют в заголовках спецификатор **class.**

Приведем в качестве примера шаблон класса матриц, аналогичных классу, описанному в разд. 2.14.6.1.

```
// Шаблон класса матриц
template <class T> class Matrix
{
  T *data;
  int N;    // число строк
  int M;    // число столбцов
public:
  Matrix(int,int);
  ~Matrix( ) { delete[ ] data; }
  __property T Items [int i] [int j] =
                          { read=GetItems, write=SetItems };
private:
  T___fastcall GetItems(int i, int j);
  void___fastcall SetItems(int i, int j, T value);
};
```

```
// конструктор
template <class T> Matrix<T>::Matrix(int n, int m)
(
 data = new T[n*m];
 for(int i = 0; i < n*m; i++)
   data[i] = 0;
 N = n;
 M = m;
}


 template <class T> void _fastcall
                    Matrix<T>::SetItems(int i, int j , T value)
 (
// запись значения value в элемент (i,j)
  if ( (i<1) I I (i>N) ] | (j<l) | | (j>M))
    ShowMessage ("Недопустимые индексы (" + IntToStr(i) +
                 "," + IntToStr (j) + ")") ;
  else data[(i — 1) * M + j - 1] = value;
}


template <class T> T _fastcall
                              Matrix<T>::GetItems(int i, int j)
{
// чтение значения элемента (i,j)
  if((i<1)| I (i>N) I | (j<l) I |(j>M))
  {
    ShowMessage("Недопустимые индексы (" + IntToStr(i) +
                "," + IntToStr(j) + ")");
    return 0;
  }
  else return data[(i - 1) * M + j — 1];
}
```

Если вы сравните этот код с приведенным ранее в разд. 2.14.6.1, то увидите, что основное отличие заключается в замене типа **float,** который использовался в разд. 2.14.6.1, на формальный тип T. Благодаря этому в самом шаблоне не указывается действительный тип хранимых данных. И при создании конкретного экземпляра класса можно будет задавать любой тип: целый, действительный, комплексный и т.п. Другое отличие приведенного кода от рассмотренного в разд. 2.14.6.1 заключается в форме ссылок заголовков элементов-функций на шаблон класса.

Создание экземпляра матрицы конкретного типа в программе может, осуществляться, например, оператором:

```
Matrix<float> x(4,5);
```

Этот оператор создает матрицу x действительных чисел размерностью 4 x 5. Отличие от приведенного в разд. 2.14.6.1 аналогичного оператора заключается в том, что после имени класса в угловых скобках указывается тип, для которого создается экземпляр класса. Компилятор заменит на этот тип (в данном случае **float)** формальный тип T, использованный в описании шаблона.

Запись и чтение элементов матрицы в программе осуществляется точно так же, как в разд. 2.14.6.1, через свойство **Items.** Например:

```
x.Items[2][3] = 1.5;
float y = x.Items[2][3];
```

Первый из этих операторов заносит значение 1,5 в 3-ий элемент 2-ой строки, а второй оператор читает это значение.

В объявлении шаблона может использоваться уже описанное выше ключевое слово typename для обозначения того, что следующий за ним идентификатор является именем типа. Это облегчает компилятору распознавание того, что скрывается за тем или иным идентификатором. Например, если в описании шаблона встретится оператор

```
T:x(y);
```

то компилятор будет в недоумении: то ли это вызов функции-элемента класса T, то ли это объявление переменной y типа T:x, в котором вы почему-то заключили имя переменной в скобки (это не запрещается синтаксисом C++). В подобных случаях принято следующее правило. Если перед идентификатором записано ключевое слово typename, то это имя типа. Например:

```
typename T:x(y);
```

В остальных случаях идентификатор — это что угодно, но только не обозначение типа.

Глава 3

Функции C, C++, библиотек C++Builder, API Windows

В настоящей главе описано около 650 функций C, C++, библиотек C++Builder, API Windows. Это еще далеко не все функции, которые можно использовать. Но ограничения на объем книги потребовали отобрать из всего трудно обозримого множества функций те, которые используются чаще всего. Более подробные списки функций вы найдете в [2].

Многие из перечисленных в данной главе функций подробно рассмотрены в гл. 4. В данной главе имена таких функций выделяются подчеркиванием. Например, Ceil. Это значит, что подробную информацию об этой функции вы найдете в гл. 4. Там же даются примеры применения этих функций. Но дать развернутые описания всех функций не представлялось возможным. Поэтому ряд разделов данной главы снабжен комментариями, в которых даются разъяснения по тем функциям, которые не описаны в гл. 4, но требуют некоторых пояснений.

3.1 Справочные сведения общего характера

3.1.1 Коды клавиш

Ниже приведены виртуальные коды клавиш, которыми можно пользоваться при обработке символов, строк, при проверке параметра Key в обработчиках событий OnKeyDown и OnKeyUp. Символы кириллицы соответствуют тем клавишам с латинскими символами, на которых они размещены.

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по символу
F1	112	0x70	VK_F1	
F2	113	0x71	VK_F2	
F3	114	0x72	VK_F3	
F4	115	0x73	VK_F4	
F5	116	0x74	VK_F5	
F6	117	0x75	VK_F6	
F7	118	0x76	VK_F7	
F8	119	0x77	VK_F8	
F9	120	0x78	VK_F9	
F10	121	0x79	VK_F10	
пробел	32	0x20	VK_SPACE	
Backspace	8	0x8	VK_BACK	
Tab	9	0x9	VK_TAB	

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по символу
Enter	13	OxOD	VK_RETURN	
Shift	16	0x10	VK_SHIFT	
Ctrl	17	0x11	VK_CONTROL	
Alt	18	0x12	VK_MENU	
CapsLock	20	0x14	VK_CAPITAL	
Esc	27	0x1B	VK_ESCAPE	
Insert	45	Ox2D	VK_INSERT	
PageUp	33	0x21	VK_PRIOR	
PageDown	34	0x22	VK_NEXT	
End	35	0x23	VK_END	
Home	36	0x24	VK_HOME	
←	37	0x25	VK_LEFT	
t	38	0x26	VK_UP	
→	39	0x27	VK_RIGHT	
↓	40	0x28	VK_DOWN	
Delete	46	Ox2E	VK_DELETE	
PrintScreen	44	Ox2C	VK_SNAPSHOT	
ScrollLock	145	0x91	VK_SCROLL	
Pause	19	0x13	VK_PAUSE	
NumLock	144	0x90	VK_NUMLOCK	
0,)	48	0x30		'0'
1, !	49	0x31		'1'
2, @	50	0x32		'2'
3, #	51	0x33		'3'
4, $	52	0x34		'4'
5, %	53	0x35		'5'
6, ^	54	0x36		'6'
7, &	55	0x37		'7'
8, *	56	0x38		'8'
9, (	57	0x39		'9'
` ~	192	0xC0		
− _	189	OxBD		
= +	187	OxBB		
[{	219	OxDB		
] }	221	OxDD		

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по символу
; :	186	OxBA		
' "	222	OxDE		
\ |	220	OxDC		
, <	188	OxBC		
. >	190	OxBE		
/ ?	191	OxBF		
a, A	65	0x41		'A'
b, B	66	0x42		'B'
c, C	67	0x43		'C'
d, D	68	0x44		'D'
e, E	69	0x45		'E'
f, F	70	0x46		'F'
g, G	71	0x47		'G'
h, H	72	0x48		'H'
U	73	0x49		'I'
У	74	Ox4A		'J'
k, K	75	Ox4B		'K'
l, L	76	Ox4C		'L'
m, M	77	Ox4D		'M'
n, N	78	Ox4E		'N'
o, O	79	Ox4F		'O'
P. P	80	0x50		'P'
q, Q	81	0x51		'Q'
r, R	82	0x52		'R'
s, S	83	0x53		'S'
t, T	84	0x54		'T'
u, U	85	0x55		'U'
v, V	86	0x56		'V'
w, W	87	0x57		'W'
x, X	88	0x58		'X'
y, Y	89	0x59		'Y'
z, Z	90	Ox5A		'Z'
На правой клавиатуре при **выключенной** клавише **NumLock**				
0	96	0x60	VK_NUMPAD0	
1	97	0x61	VK_NUMPAD1	

Клавиша	Десятичное число	Шестнадцатеричное число	Символическое имя	Сравнение по символу
2	98	0x62	VK_NUMPAD2	
3	99	0x63	VK_NUMPAD3	
4	100	0x64	VK_NUMPAD4	
5	101	0x65	VK_NUMPAD5	
6	102	0x66	VK_NUMPAD6	
7	103	0x67	VK_NUMPAD7	
8	104	0x68	VK_NUMPAD8	
9	105	0x69	VK_NUMPAD9	
*	106	0x6A	VK_MULTIPLY	
+	107	0x6B	VK_ADD	
-	109	Ox6D	VK_SUBTRACT	
.	110	0x6E	VK_DECIMAL	
/	111	Ox6F	VK_DIVIDE	

Комментарии

Приведенные коды клавиш можно, например, использовать в обработчиках событий **OnKeyDown** таких компонентов, как окна редактирования. В обработчики этих событий передается параметр Key, значение которого равно виртуальному коду нажатой пользователем клавиши. Этот параметр является целым числом, определяющим клавишу, а не символ. Например, один и тот же код соответствует прописному и строчному символам "Y" и "y". Если, как это обычно бывает, в русской клавиатуре этой клавише соответствуют символы кириллицы "Н" и "н", то их код будет тем же самым. Различить прописные и строчные символы или символы латинские и кириллицы невозможно.

Проверять нажатую клавишу можно, сравнивая Key с целым десятичным кодом клавиши, приведенном во втором столбце таблицы. Например, реакцию на нажатие пользователем клавиши Enter можно оформить оператором:

```
if(Key == 13) ... ;
```

Можно сравнивать Key и с шестнадцатеричным эквивалентом кода, приведенным в третьем столбце таблицы. Например, приведенный выше оператор можно записать в виде:

```
if(Key== 0xOD) ... ;
```

Для клавиш, которым не соответствуют символы, введены также именованные константы, которые облегчают написание программы, поскольку не требуют помнить численные коды клавиш. Например, приведенный выше оператор можно записать в виде:

```
if(Key == VK_RETURN) ... ;
```

Для клавиш символов и цифр можно производить проверку сравнением с десятичным или шестнадцатеричным кодом, но это не очень удобно, так как трудно помнить коды различных символов. Другой путь — воспользоваться тем, что коды латинских символов в верхнем регистре совпадают с виртуальными кодами, используемыми в параметре Key. Поэтому, например, если вы хотите распознать клавишу, соответствующую символу "Y", вы можете написать:

```
if(Key == 'Y') ... ;
```

В этом операторе можно использовать только латинские символы в верхнем регистре. Если вы напишете "у" или захотите написать русские символы, соответствующие этой клавише — "Н" или "н", то оператор не сработает.

3.1.2 Коды основных символов

Ниже приведена таблица основных символов и соответствующих им чисел (см. разд. 2.4) при работе с русифицированными версиями Windows.

число	символ	число	символ	число	символ	число	символ	число	символ
33	!	34	"	35	#	36	$	37	%
38	&	39	'	40	(	41	)	42	*
43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4
53	5	54	6	55	7	56	8	57	9
58	:	59	;	60	<	61	=	62	>
63	?	64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G	72	H
73	I	74	J	75	K	76	L	77	M
78	N	79	O	80	P	81	Q	82	R
83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\
93	]	94	^	95	_	96	`	97	a
98	b	99	c	100	d	101	e	102	f
103	g	104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o	112	p
113	q	114	r	115	s	116	t	117	и
118	v	119	w	120	x	121	y	122	z
123	{	124	|	125	}	126	~	132	„
133	…	145	‘	146	’	147	“	148	”
149	•	150	–	151	—	153	™	166	
167	§	168	Ё	169	©	171	«	172	¬
174	®	176	°	177	±	178	І	179	і
182	¶	183	·	184	ё	185	№	187	»
192	А	193	Б	194	В	195	Г	196	Д
197	Е	198	Ж	199	З	200	И	201	Й
202	К	203	Л	204	М	205	Н	206	О
207	П	208	Р	209	С	210	Т	211	У
212	Ф	213	Х	214	Ц	215	Ч	216	Ш

число	символ	число	символ	число	символ	число	символ	число	символ
217	Щ	218	Ъ	219	Ы	220	Ь	221	Э
222	Ю	223	Я	224	а	225	б	226	в
227	г	228	д	229	е	230	ж	231	з
232	и	233	й	234	к	235	л	236	м
237	н	238	о	239	п	240	р	241	с
242	т	243	у	244	ф	245	х	246	ц
247	ч	248	ш	249	щ	250	ъ	251	ы
252	ь	253	э	254	ю	255	я	–	–

3.1.3 Форматы и типы, используемые при форматировании данных

3.1.3.1 Строка форматирования функций вывода

Строка формата, используемая во многих функциях вывода данных (printf, cprintf, sprintf и др.), состоит из обычных символов, управляющих последовательностей символов и спецификаций полей формата вывода аргументов. Обычные символы и управляющие последовательности просто копируются в выходную строку.

Спецификации полей формата начинаются с символа % и имеют вид:

```
%[flags][width][.precision][F|N|h|l|L]type
```

Все символы спецификации записываются без пробелов между ними.

Единственно обязательным элементом спецификации является type — символ, указывающий на тип данных вводимого поля. Остальные необязательные элементы задают параметры форматирования:

[flags]	Флаги выравнивания, управления печатью знака числа, управления пробелами, десятичной точкой, основанием печати (восьмеричная, шестнадцатеричная)
[width]	Ширина поля — минимальное число выводимых символов
[.precision]	Спецификатор точности — максимальное количество печатаемых символов или минимальное количество разрядов печатаемого целого
[F|N|h|l|L]	Модификаторы, изменяющие размер аргумента по умолчанию:
	N ближний указатель (near) F дальний указатель (far) h short int I long L long double

Ниже приведены возможные значения type.

Символ	Тип аргумента	Формат вывода
Числа		
d	целый	десятичное signed integer
i	целый	десятичное signed integer
o	целый	восьмеричное unsigned integer
u	целый	десятичное unsigned integer
x	целый	шестнадцатеричное unsigned int (с цифрами a, b, c, d, e, f)
X	целый	то же, что x, но с цифрами A, B, C, D, E, F
f	действительный	формат с фиксированной точкой: [-]dddd.dddd
e	действительный	экспоненциальный (научный) формат: [-]d.dddd e[+/-]ddd
E	действительный	то же, что e, но с символом E
g	действительный	наиболее компактный из форматов e и f для данного числа и данной точности; незначащие нули не выводятся
(i	действительный	то же, что g, но с символом E в экспоненциальном формате
Символы		
c	символ	один символ
s	указатель на строку	строка символов до нулевого символа в конце или с числом символов, заданных точностью
%	нет	печать символа %
Указатели		
n	указатель на int	в ячейку памяти, на которую указывает аргумент, заносится количество выведенных к данному моменту символов
P	указатель	печать аргумента как указателя; в зависимости от используемой модели памяти печатается или XXXX:YYYY, или YYYY (только смещение)

Перечисленные ниже флаги **flags** могут записываться в любой последовательности и в любой комбинации.

Флаг	Пояснение
–	Выравнивание влево, оставшееся поле справа заполняется пробелами. Если этот флаг не задан, то производится выравнивание вправо, а оставшееся поле слева заполняется нулями или пробелами.
+	Обязательно перед числом указывается знак плюс (+) или минус (–).
пробел	Если значение не отрицательное, то печать начинается с пробела вместо знака плюс (+). Для отрицательного значения знак минус (-) печатается. Если наряду с этим флагом задан флаг +, то он должен быть указан до флага пробела.

Флаг	Пояснение
#	В форматах o, x, X добавляется префикс 0, Ox, OX соответственно. В форматах *e,* E, f, g, G во всех случаях выводится десятичная точка. Кроме того в форматах g, G не подавляется вывод незначащих нулей.

Спецификатор **width** задает минимальную ширину поля. Спецификатор может быть задан или явным образом — десятичным числом, или косвенно — символом звездочки (*). В последнем случае предполагается, что ширину поля задает очередной аргумент из списка.

Спецификатор **width** указывает только минимальную ширину. Если вывод данного аргумента требует большей ширины поля, то поле расширяется и значение никогда не усекается.

Спецификатор может принимать следующие значения:

n	Выводится по крайней мере n символов. Если для вывода требуется меньше символов, то лишние позиции (слева или справа, в зависимости от флагов) заполняются пробелами.
On	Выводится по крайней мере n символов. Если для вывода требуется меньше символов, то лишние позиции слева заполняются нулями.
*	Ширину поля задает очередной аргумент из списка.

Спецификатор точности **precision** определяет максимальное число выводимых символов или место десятичной точки. Он записывается после символа точки (.), чтобы отделить его от предшествующего спецификатора **width.** Данный спецификатор, как и **width,** может быть задан или явным образом — десятичным числом, или косвенно — символом звездочки (*). В последнем случае предполагается, что точность задается очередным аргументом из списка.

Отсутствие спецификатора **precision** означает точность по умолчанию и эквивалентно:

1	для форматов d, i, o, u, x, X
6	для форматов e, E, f
числу значащих цифр	для форматов g, G
выводу до нулевого символа	для формата s
не влияет	на формат c

Возможные значения **precision:**

.0	Для фоматов d, i, o, u, x эквивалентно точности по умолчанию. Для форматов e, E, f означает вывод без десятичной точки.
.n	Задает вывод n символов или позицию n десятичной точки. Если выводимая величина содержит более n символов, то строка символов усекается, а число может округляться (в зависимости от формата).
*	Точность задает очередной аргумент из списка.

Ниже приведены сведения о влиянии значения **precision** на различные форматы.

d, i, o, u, x, X	Указывает, что должно выводиться по крайней мере п цифр. Если число имеет менее п цифр, позиции слева заполняются нулями. Если число имеет более п цифр, число не усекается.
e, E, f	Указывает, что после десятичной точки должно выводиться п цифр. Последняя цифра округляется.
g. G	Указывает, что должно выводиться до п цифр.
c	Спецификатор не влияет.
s	Указывает, что должно выводиться не более п символов.

Примеры

Примеры влияния формата:

%f	%e	%g	%#G
110000.000000	1.100000e+05	110000	110000.
-110000000.000000	-1.100000e+08	-1.1e+08	-1.10000E+08
0.000110	1.100000e-04	0.00011	0.000110000
0.000000	1.100000e-07	1.1e-07	1.10000E-07
12.000000	1.200000e+01	12	12.0000
0.000000	0.000000e+00	0	0.00000

Примеры влияние флагов:

спецификация	результат
%6i	12 -12
%-6i	12 -12
%+6i	+12 -12
%06i	000012 -00012

Примеры влияния точности:

спецификация		
%f	123456789.000000	0.123457
%.5f	123456789.00000	0.12346
%.4f	123456789.0000	0.1235
%.3f	123456789.000	0.123
%e	1.234568e+08	1.234568e-01
%.5e	1.23457e+08	1.23457e-01

спецификация		
%.4e	1.2346e+08	1.2346e-01
%.3e	1.235e+08	1.235e-01
%g	1.23457e+08	0.123457
%.5g	1.2346e+08	0.12346
%.4g	1.235e+08	0.1235
%.3g	1.23e+08	0.123
%.2g	1.2e+08	0.12

Примеры использования строка форматирования вы можете также найти в гл. 4 в описаниях функций вывода.

3.1.3.2 Строка форматирования функций ввода

Описанная ниже строка формата, используется во многих функциях ввода данных **(scanf. fscanf. sscanf** и др.). Строка может включать три вида элементов:

- пробельные символы (пробел " ", табуляцию "\t", символ новой строки "\n"
- не пробельные печатные символы (кроме %)
- спецификации формата

Если в строке встретился пробельный символ, то с этого момента пробельные символы до первого не пробельного символа считываются из входного потока, но не участвуют в присваивании значений переменным (игнорируются).

Если в строке встретился печатный не пробельный символ, то с этого момента из входного потока считывается и игнорируется последовательность символов, встретившаяся в строке формата. Если последовательность символов во входном потоке не соответствует записанной в строке формата, то форматирование прерывается.

Спецификации формата начинаются с символа % и имеют вид:

```
% [*] [width] [F|N] [h|l|L] type
```

Все символы спецификации записываются без пробелов между ними.

Единственно обязательным элементом спецификации является **type** — символ, указывающий на то, как будет трактоваться вводимый аргумент. Остальные необязательные элементы задают параметры форматирования:

[*]	Запрет занесения в память читаемого поля. Поле сканируется, но его значение не присваивается аргументу из списка.	
[width]	Ширина поля — максимальное число читаемых символов. Реально может быть прочитано меньше символов, если во входном потоке раньше встретится пробельный символ или символ, который не может быть преобразован согласно заданному формату.	
[F|N]	Модификаторы, изменяющие размер аргумента по умолчанию:	
	N	ближний указатель (near)
	F	дальний указатель (far)
[h|l|L]	Модификаторы, изменяющие размер аргумента по умолчанию:	
	h	short int
	1	long int, если type соответствует целому числу, или double, если type соответствует действительному числу
	L	long double

Ниже приведены возможные значения **type.**

Символ	Ожидаемый тип данных	Тип аргумента
Числа		
d	целый	указатель на int (int *arg)
D	целый	указатель на long (long *arg)
e,E	действительный	указатель на float (float *arg)
f	действительный	указатель на float (float *arg)
g,G	действительный	указатель на float (float *arg)
o	восьмеричный целый	указатель на int (int *arg)
O	восьмеричный целый	указатель на long (long *arg)
i	десятичный, восьмеричный или шестнадцатеричный целый	указатель на int (int *arg)
I	десятичный, восьмеричный или шестнадцатеричный целый	указатель на long (long *arg)
и	десятичный целый без знака	указатель на unsigned int (unsigned int *arg)
U	десятичный целый без знака	указатель на unsigned long (unsigned long *arg)
x	шестнадцатеричный целый	указатель на int (int *arg)
X	шестнадцатеричный целый	указатель на int (int *arg)
Символы		
s	строка символов	указатель на массив символов (char arg[])
c	символ	указатель на char (char *arg) или, если задана ширина поля (например, %5c), то на массив символов размером W (char arg[W])
%	символ %	не преобразуется
Указатели		
п	указатель на int (int *arg)	в ячейку памяти, на которую указывает аргумент, заносится количество успешно прочитанных к данному моменту символов
P	шестнадцатеричный формат: YYYY:ZZZZ или ZZZZ	указатель на объект (far* или near*)

Примеры использования строки форматирования вы можете найти в гл. 4 в описании функций ввода.

3.1.3.3 Строка форматирования функций типа Format

Описанная ниже строка форматирования используется в функциях **Format. FormatBuf, FmtStr, StrFmt, StrLFmt и др.**

Строка содержит обычные символы и спецификаторы формата полей. Обычные символы просто копируются в выходную строку, а спецификаторы определяют форматирование аргументов из заданного списка.

Спецификации формата начинаются с символа % и имеют вид:

```
"%" [index ":"] ["-"] [width] ["." prec] type
```

Единственно обязательным элементом спецификации является **type** — символ, указывающий на то, как будет трактоваться аргумент. Остальные необязательные элементы задают параметры форматирования:

[index ":"]	Устанавливает текущий индекс массива аргументов в заданное значение index. Индексы начинаются с 0. Например, спецификатор %0: переводит индекс на начало массива и обеспечивает повторное форматирование первого аргумента.
["-"]	Обеспечивает выравнивание результата влево с заполнением оставшихся правых позиций поля пробелами. В отсутствие спецификатора ["-"] выравнивание производится вправо.
[width]	Устанавливает минимальную ширину поля в результирующей строке. Если результат преобразования короче ширины поля, происходит выравнивание вправо (или влево, если был записан спецификатор ["-"]) с заполнением лишних позиций пробелами.
["." prec]	Спецификатор точности, определяющий число выводимых символов (в зависимости от принятого формата). Спецификатор записывается после символа точки (.), чтобы отделить его от предшествующего спецификатора width.

Значения спецификаторов **index, width** и **prec** могут задаваться в виде целых значений или в виде символа звездочки (*).В последнем случае предполагается, что значение спецификатора задается очередным аргументом из списка.

Ниже приведены возможные значения **type.**

Символ	Тип аргумента	Формат вывода
d	целый	Десятичный формат — строка десятичных цифр. Если используется спецификатор ["." **prec],** то он указывает минимальное количество выводимых цифр. Если действительное количество цифр результата меньше указанного спецификатором точности, то происходит выравнивание вправо с заполнением лишних позиций нулями.
e	действительный	Научный формат — строка вида "-d.ddd...E+ddd". Перед десятичной точкой всегда помещается одна цифра и для отрицательных величин — знак минус. Если используется спецификатор ["." **prec],** то он указывает общее количество выводимых цифр, включая цифру перед десятичной точкой (по умолчанию 15 цифр). После символа порядка "E" всегда указывается знак — плюс или минус.
f	действительный	Формат с фиксированной точкой — строка вида "-ddd.ddd...". Если используется спецификатор ["." **prec],** то он указывает количество выводимых цифр после десятичной точки (по умолчанию 2 цифры).

Символ	Тип аргумента	Формат вывода
e	действительный	Универсальный формат — преобразование в научный формат или формат с фиксированной точкой, в зависимости от того, какой из них дает более компактный результат. Если используется спецификатор ["." prec], то он указывает количество выводимых значащих разрядов (по умолчанию — 15). Начальные нули не печатаются, десятичная точка печатается, если необходимо. Формат с фиксированной точкой используется, если в преобразуемом значении число цифр до десятичной точки меньше заданной точности и если значение не меньше 0.00001. В остальных случаях используется научный формат.
n	действительный	Числовой формат — то же, что формат с фиксированной точкой, но с добавлением разделителей тысяч: "–d,ddd,ddd.ddd...".
m	действительный	Монетарный формат — число преобразуется в строку, отображающую денежную сумму. Формат контролируется глобальными переменными CurrencyString, Currency Format, NegCurrFormat, ThousandSeparator, DecimalSeparator, CurrencyDecimals, задаваемыми для монетарного формата разделом Currency Format элемента International Контрольной панели Windows. Если используется спецификатор ["." prec], то он заменяет собой значение глобальной переменной CurrencyDecimals.
P	указатель	Указатель — значение преобразуется в строку вида "XXXX:YYYY", где XXXX и YYYY — сегмент и смещение, выраженные четырьмя шестнадцатеричными цифрами.
s	символ, строка или тип PChar	Строка символов. Если используется спецификатор ["." prec], то он задает максимальное число символов. Если строка длиннее указанного числа, она усекается.
x	целый	Шестнадцатеричный формат — строка шестнадцатеричных цифр. Если используется спецификатор ["." prec], то он указывает минимальное количество цифр результата. Если результат короче, лишние позиции слева заполняются нулями.

Все указанные в приведенной таблице обозначения форматов могут записываться в нижнем или верхнем регистре, что никак не влияет на результат.

Для форматов действительных чисел реально используемые символы десятичной точки и разделителей тысяч определяются глобальными переменными **DecimalSeparator и ThousandSeparator.**

Примеры

Примеры влияния формата:

число	%f	%e	%g
110000.	110000,00	1,10000000000000E+005	110000
-1.1e+08	-110000000,00	-1,10000000000000E+008	-110000000

число	%f	%e	%g
0.00011	0,00	1,10000000000000E-004	0,00011
1.1e-07	0,00	1,10000000000000E-007	1,1E-7
12.	12,00	1,20000000000000E+001	12
0,	0,00	0,00000000000000E+000	0

Примеры влияния точности:

спецификация / число	1,1E-4	12.	0,00
%.2f	0,00	12,00	0,000
%.3f	0,000	12,000	0,0000
%.4f	0,0001	12,0000	0,0E+000
%.2e	1,1E-007	1,2E+001	0,OE+000
%.3e	1,10E-007	1,20E+001	0,OOE+000
%.4e	1,100E-007	1,200E+001	0,000E+000
%.2g	1,1E-7	12	0
%.3g	1,1E-7	12	0
%.4g	1,1E-7	12	0

3.1.3.4 TFloatFormat и TFloatValue — типы форматирования действительных чисел

Типы **TFloatFormat и TFIoatValue** определяют форматирование действительных чисел в таких функциях, как **FloatToText, FloatToStrF. FloatToDecimal, TextToFloat.**

Синтаксис

```
#include <SysUtils.hpp>
enum TFloatFormat { ffGeneral, ffExponent, ffFixed, ffNumber,
                    ffCurrency };
enum TFIoatValue { fvExtended, fvCurrency };
```

Описание

TFIoatValue указывает тип преобразуемого числа. Значение **fvExtended** соответствует обычному числу с плавающей запятой, а значение **fvCurrency** — числу типа **Currency.**

Тип TFloatFormat определяет коды форматирования чисел с плавающей запятой в функциях **FloatToText, FloatToStrF. FloatToDecimal, TextToFloat.** Возможные значения формата определяют следующие правила форматирования:

ffGeneral	Основной числовой формат. Число преобразуется по формату с фиксированной точкой или научному в зависимости от того, какой из них оказывается короче. Начальные нули удаляются, десятичная точка ставится только при необходимости. Фиксированный формат используется, если число разрядов слева от точки не больше указанной точности **Precision** и если значение не меньше 0.00001. В противном случае используется научный формат, в котором параметр **Digits** определяет число разрядов степени — от 0 до 4.

ffExponent	Научный формат. Число преобразуется в строку вида "-d.ddd...E+dddd". Общее число цифр, включая одну перед десятичной точкой, задается параметром **Precision.** После символа "**E**" всегда следует знак "+" или "-" и до четырех цифр. Параметр Digits определяет минимальное число разрядов степени — от 0 до 4.
ffFixed	Формат с фиксированной точкой. Число преобразуется в строку вида "-ddd.ddd...". По крайней мере одна цифра всегда предшествует десятичной точке. Число цифр после десятичной точки задается параметром **Digits,** который может лежать в пределах от О до 18. Если число разрядов слева от десятичной точки больше указанного параметром **Precision,** то используется научный формат.
ffNumber	Числовой формат. Число преобразуется в строку вида "-d,ddd,ddd.ddd...". Данный формат совпадает с **ffFixed** за исключением наличия в нем разделителей тысяч.
ffCurrency	Монетарный формат. Число преобразуется в строку, отображающую денежную сумму. Формат контролируется глобальными переменными **CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator,** задаваемыми для монетарного формата разделом Currency Format элемента International Контрольной панели Windows. Число цифр после десятичной точки задается параметром **Digits,** который может лежать в пределах от 0 до 18.

Для всех форматов действительные символы, используемые в качестве десятичной точки и разделителя тысяч, определяются глобальными переменными **DecimalSeparator и ThousandSeparator.**

3.1.3.5 Строка форматирования функций типа FormatFloat

Строка форматирования, описанная ниже, применяется в функциях **FormatFloat, FloatToTextFmt,** в методе **FormatFloat** класса AnsiString и в некоторых других.

В строке используются следующие символы:

0	Сохранение позиции для цифры. Если форматируемое число содержит цифру в позиции, в которой в строке форматирования имеется символ "О", то эта цифра копируется в выходную строку. В противном случае в этой позиции в выходной строке содержится "О".
#	Сохранение позиции для цифры. Если форматируемое число содержит цифру в позиции, в которой в строке форматирования имеется символ "#", то эта цифра копируется в выходную строку. В противном случае в эту позицию в выходной строке ничего не заносится.
.	Десятичная точка. Первый символ точки "." в строке форматирования определяет позицию десятичной точки в отформатированном числе. Любые последующие символы "." в строке игнорируются. Действительный символ, используемый в качестве десятичной точки, определяется глобальной переменной **DecimalSeparator,** установленной в разделе Number Format элемента International программы «Панель управления» Windows.

,	Разделитель тысяч. Если строка форматирования содержит один или более символов ",", то в выходной строке будут использованы разделители тысяч. Местоположение символов "," в строке форматирования безразлично — это просто указание, что надо использовать разделители тысяч. Действительный символ, используемый в качестве разделителя, определяется глобальной переменной ThousandSeparator, установленной в разделе Number Format элемента International программы «Панель управления» Windows.
E+, E–, e+, e–	Научный формат. Если в строке форматирования встречаются символы "E+", "E–", "e+" или "e–", то при форматировании используется научный формат. Сразу после этих символов может быть расположена группа символов "0" (до четырех символов), которая определяет минимальное число цифр в показателе степени. Если в строке использованы символы "E+" или "e+", то как перед положительной, так и перед отрицательной степенью будет всегда помещаться знак "+" или "–". Если в строке использованы символы "E–" или "e–", то знак будет помещаться только перед отрицательной степенью.
'xx'/"xx"	Символы, заключенные в одинарные или двойные кавычки, выводятся в выходную строку, никак не влияя на форматирование.
;	Символ разделяет разделы строки, связанные с форматированием положительных, отрицательных и нулевых значений.

Расположение крайнего левого символа "0" перед десятичной точкой и крайнего правого символа "0" после десятичной точки определяет число цифр, всегда присутствующих в выходной строке.

Форматируемое число всегда округляется до стольких десятичных разрядов, сколько символов "0" и "#" находится справа от десятичной точки. Если строка форматирования не содержит десятичной точки, значение форматируемого числа округляется до ближайшего целого.

Если форматируемое число имеет больше цифр слева от десятичной точки, чем количество расположенных в строке форматирования символов "0" и "#", то лишние цифры все равно выводятся в начале числа.

Строка форматирования может содержать от одной до трех секций, разделяемых точкой с запятой. Если задана только одна секция, то она применяется для форматирования любых чисел. Если задано две секции, то первая используется при форматировании положительных чисел и нуля, а вторая — при форматировании отрицательных чисел. Если заданы три секции, то первая относится к положительным числам, вторая — к отрицательным, третья — к нулю.

Если секции отрицательных чисел или нуля пустые (т.е. ничего не написано после соответствующей точки с запятой), то вместо них используется секция положительных чисел.

Если секция положительных чисел пустая или вообще строка форматирования пустая, то используется основной формат чисел с плавающей запятой с 15 значащими разрядами. Этот формат соответствует формату **ffGeneral** типа TFloatFormat (см. разд. 3.1.3.4). Этот же формат используется, если число имеет более 18 разрядов до десятичной точки и строка форматирования не содержит указания на применение научного формата.

Примеры

Ниже приведены строки форматирования и соответствующие им выходные строки.

Строка форматирования				
пустая	1234	-1234	0.5	0
0	1234	-1234	1	0
0.00	1234,00	-1234,00	0,50	0,00
#.##	1234	-1234	,5	
#,##0.00	1 234,00	-1 234,00	0,50	0,00
#,##0.00;(#,##0.00)	1 234,00	(1 234,00)	0,50	0,00
#,##0.00;;Нуль	1 234,00	-1 234,00	0,50	Нуль
0.000E+00	1,234E+03	-1,234E+03	5,000E-01	0,000E+00
#.###E-0	1,234E3	-1,234E3	5E-1	0E0

3.1.4 Обработка ошибок времени выполнения, диагностика

Чтобы работать с сообщениями об ошибках времени выполнения, в приложение должна быть включена директива

```
#include <errno.h>
```

3.1.4.1 _doserrno, errno и _sys_nerr — переменные, содержащие коды ошибок

Переменные **_doserrno** и errno типа **int** получают положительные значения при возникновении различных ошибок времени выполнения. Значения **_doserrno** и errno задаются одновременно, но иногда они могут различаться, поскольку errno — переменная, совместимая с UNIX.

Коды ошибок, присваиваемые errno, являются одновременно индексами массива _sys_errlist, содержащего сообщения об ошибках. Кроме того имеется переменная **_sys_nerr**, которая содержит номер ошибки и используется функцией perror (файл stdio.h) для вывода в стандартный поток сообщений об ошибках stder. Поэтому доступ к соответствующему сообщению можно получить или как _sys_errlist[errno], или как **_sys_errlist[_sys_nerr]**.

Нормальное значение рассматриваемых переменных — 0. При выполнении различных математических функций, при манипуляциях с файлами и т.п. это значение при возникновении ошибки изменяется и сохраняется таким вплоть до следующего обращения к соответствующей функции. Если это нежелательно, надо программно сбрасывать значения в 0.

3.1.4.2 Коды ошибок

Ниже приводится таблица, содержащая мнемонические константы ошибок, их коды и соответствующие сообщения из массива _sys_errlist.

Константа	Код	Сообщение в **_sys_errlist**
E2BIG	20	Arg list too long
EACCES	5	Permission denied
EAGAIN	42	Resource temporarily unavailable
EBADF	6	Bad file number
EBUSY	44	Resource busy

Константа	Код	Сообщение в _sys_errlist
ECHILD	24	No child process
ECONTR	7	Memory blocks destroyed
ECURDIR	16	Attempt to remove CurDir
EDEADLOCK	36	Locking violation
EDOM	33	Math argument
EEXIST	35	File already exists
EFAULT	14	Unknown error
EFBIG	27	Для UNIX — в MSDOS отсутствует
EINTR	39	Interrupted function call
EINVACC	12	Invalid access code
EINVAL	19	Invalid argument
EINVDAT	13	Invalid data
EINVDRV	15	Invalid drive specified
EINVENV	10	Invalid environment
EINVFMT	11	Invalid format
EINVFNC	1	Invalid function number
EINVMEM	9	nvalid memory block address
EЮ	40	Input/output error
EISDIR	46	Для UNIX — в MSDOS отсутствует
EMFILE	4	Too many open files
EMLINK	31	Для UNIX — в MSDOS отсутствует
ENFILE	23	Too many open files
ENMFILE	18	No more files
ENODEV	15	No such device
ENOENT	2	No such file or directory
ENOEXEC	21	Exec format error
ENOFILE	2	File not found
ENOMEM	8	Not enough core
ENOPATH	3	Path not found
ENOSPC	28	No space left on device
ENOTBLK	43	Для UNIX — в MSDOS отсутствует
ENOTDIR	45	Для UNIX — в MSDOS отсутствует
ENOTSAM	17	Not same device
ENOTTY	25	Для UNIX — в MSDOS отсутствует
ENXIO	41	No such device or address
EPERM	37	Operation not permitted

Константа	Код	Сообщение в _sys_errlist
EPIPE	32	Broken pipe
ERANGE	34	Result too large
EROFS	30	Read-only file system
ESPIPE	29	Illegal seek
ESRCH	38	Для UNIX — в MSDOS отсутствует
ETXTBSY	26	Для UNIX — в MSDOS отсутствует
EUCLEAN	47	Для UNIX — в MSDOS отсутствует
EXDEV	22	Cross-device link
EZERO	0	Error 0

Стандартные сообщения можно изменять. Например, оператор

```
strcpy(_sys_errlist[ENOENT],"Нет такого файла или каталога");
```

русифицирует стандартное сообщение "No such file or directory".

Ниже приведена таблица других кодов ошибок -- ошибок файлового ввода-вывода, которые возникают, если в проекте C++Builder включена опция I/O checking на странице Pascal окна опций проекта. Эти коды генерируются в C++Builder при создании исключения **EInOutError**.

Код	Ошибка
2	Файл не найден
3	Неправильное имя файла
4	Слишком много открытых файлов
5	Файл не доступен
100	Достигнут конец файла (EOF)
101	Диск переполнен
106	Ошибка ввода

3.1.4.3 EDOM, ERANGE — константы сообщений об ошибках

Символические целочисленные константы **EDOM и ERANGE** используются в математических и других функциях для сообщений об ошибках. Узнать, возникла ли соответствующая ошибка при выполнении некоторой функции, можно проверкой переменной **errno,** например:

```
if(errno == EDOM) ...;
```

3.1.4.4 _matherr и _matherrl — обработчики ошибок

Синтаксис

```
#include <math.h>
int _matherr(struct _exception *e);
int _matherrl(struct _exceptionl *e);
```

Описание

Функция **_matherr** или **_matherrl** (для типов **long double)** вызываются библиотечными математическими функциями при возникновении в них ошибок, свя-

занных с недопустимыми значениями параметров (корень или логарифм отрицательного числа и т.п.). Функции перехватывают только ошибки, выхода за пределы области определения и выхода за диапазон допустимых значений, но не реагируют на исключения при выполнении математических операций, например, при делении на 0. Для перехвата таких событий служит функция **signal.**

Стандартные варианты **_matherr и _matherrl** могут быть переопределены пользователем, если он объявит в своем приложении аналогичные функции. Эти функции пользователя должны возвращать ненулевое значение, если они обработали ошибку. В этом случае не возникает стандартного сообщения об ошибке и не изменяется значение переменной **errno.** Если переписанные пользователем варианты **_matherr и _matherrl** не обработали данную ошибку, они должны вернуть 0. Тогда будет проведена стандартная обработка ошибки.

В качестве параметра e в функции передаются структуры: *

```
struct _exception {
   int    type;
   char   *name;
   double arg1, arg2, retval;
};

struct _exception1 {
   int    type;
   char   *name;
   long double arg1, arg2, retval;
};
```

Элемент структуры **type** определяет тип ошибки. Элемент **name** указывает на строку, содержащую имя функции, в которой произошла ошибка. Элементы **arg1** и *arg2* — это значения аргументов, приведшие к ошибкам (если функция имеет один аргумент, то его значение помещается в arg1). Элемент **retval** — возвращаемое по умолчанию значение функции. Пользователь может изменить это значение.

Тип ошибки, хранящийся в элементе **type,** может принимать одно из следующих значений:

DOMAIN	аргумент выходит за пределы области определения; например, **log(−1)**
SING	аргумент соответствует особой точке функции; например, pow(0, -2)
OVERFLOW	аргумент приводит к значению функции, превышающему DBL_MAX (или LDBL_MAX); например, **exp(1000)**
UNDERFLOW	аргумент приводит к значению функции, меньшему чем DBL_MIN (или LDBL_MIN); например, **exp(−1000)**
TLOSS	аргумент приводит к значению функции с полной потерей значащих разрядов; например, sin(10e70)

Фигурирующие в приведенном описании макросы **DBL_MAX, DBL_MIN, LDBL_MAX и LDBL_MIN** определены в файле **float.h.**

Пример

Приведенный ниже пример показывает функцию, обрабатывающую ошибку типа **DOMAIN** функции **sqrt** (корень из отрицательного числа), заменяя результат на корень из положительного числа:

```
int _matherr (struct _exception *a)
{
  if (a->type == DOMAIN)
    if (!strcmp(a->name,"sqrt")) {
```

```
        a->retval = sqrt (-(a->arg1));
      return 1;
      }
    return 0;
}
```

3.1.5 Некоторые сообщения Windows

В разд. 1.14, в данной главе и в гл. 4 описан ряд функций работы с сообщениями Windows. Ниже приводятся сведения о некоторых наиболее часто используемых сообщениях.

WM_ACTIVATE

Сообщение посылается, когда окно переводится в активное или неактивное состояние. Сначала посылается окну, переходящему в неактивное состояние, а потом — активируемому.

Определение

```
WM_ACTIVATE
  fActive = LOWORD(wParam);
  fMinimized = (BOOL) HIWORD(wParam);
  hwndPrevious = (HWND) IParam;
```

Параметры

fActive — показывает, как активируется или деактивируется окно. Возможные значения:

WA_ACTIVE	активируется не щелчком мыши (например, функцией SetActiveWindow или клавиатурой)
WA_CLICKACTIVE	активируется щелчком мыши
WA_INACTIVE	деактивируется

fMinimized — ненулевое значение, показывающее, что окно минимизировано.

hwndPrevious — дескриптор, который указывает на окно, из которого фокус переключился на данное окно, если оно активируется, или на окно, в которое передается управление, если данное окно деактивируется.

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно возвращать нуль.

Действие по умолчанию

Если активируемое окно не свернуто, то оно получает фокус.

Примечания

Если окно активируется щелчком мыши, оно получает также сообщение **WM_MOUSEACTIVATE.**

WM_ACTIVATEAPP

Сообщение посылается при переходе активности от окна одного приложения к окну другого приложения. Сообщения посылаются обоим окнам.

Определение

```
WM_AC TIVATEAPP
 fActive = (BOOL) wParam;
 dwThreadID = (DWORD) IParam;
```

Параметры

fActive — значение **true** означает, что окно становится активным, а **false** — что окно теряет активность.

dwThreadID — указывает сторонний процесс, который теряет или приобретает активность.

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно возвращать нуль.

WM_CANCELMODE

Сообщение посылается окну, имеющему фокус при отображении модальных форм — диалогов и сообщений об ошибках. Дает возможность окну закрыться и освобождает мышь.

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно возвращать нуль.

Действие по умолчанию

Внутренний процесс завершается и мышь освобождается.

WM_CLOSE

Сигнализирует, что окно или приложение закрывается.

Определение

```
WM_CLOSE
```

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно возвращать нуль.

Действие по умолчанию

Вызывается функция **DestroyWindow,** уничтожающая окно.

Примечания

Приложение при обработке этого сообщения может запросить пользователя о необходимости закрывать окно и вызвать функцию **DestroyWindow** только при положительном ответе.

WM_GETMINMAXINFO

Посылается перед изменением размеров или положения окна. Обработчик сообщения может использоваться для ограничения допустимых размеров и координат положения на экране.

Определение

```
WM_GETMINMAXINFO
lpmmi = (LPMINMAXINFO) lParam;
```

Параметры

Параметр **lpmmi** указывает на структуру типа MINMAXINFO, содержащую принятые по умолчанию пределы изменения размеров и координат положения окна. Описание этой структуры:

```
typedef struct tagMINMAXINFO {
    POINT ptReserved;
    POINT ptMaxSize;
    POINT ptMaxPosition;
    POINT ptMinTrackSize;
    POINT ptMaxTrackSize;
} MINMAXINFO;
```

Поля структуры означают следующее:

ptReserved	Зарезервировано и пока не используется
ptMaxSize	Поле типа Point определяет ширину (Point.x) и высоту (Point.y) развернутого окна
ptMaxPosition	Поле типа Point определяет положения левого (Point.x) и верхнего (Point.y) краев развернутого окна
ptMinTrackSize	Поле типа Point определяет минимальную ширину (Point.x) и минимальную высоту (Point.y) окна при изменении пользователем размеров его рамки
ptMaxTrackSize	Поле типа Point определяет максимальную ширину (Point.x) и максимальную высоту (Point.y) окна при изменении пользователем размеров его рамки

Возвращаемое значение

Если приложение обрабатывает это сообщение, оно должно вернуть 0.

Пример

Приведенный ниже обработчик события **WM_GETMINMAXINFO** ограничивает высоту окна в пределах 100 х 200 пикселов, ширину — в пределах 150 х 300 пикселов и задает координаты левого верхнего угла распахнутого окна равными текущим координатам левого верхнего угла окна.

```
type
 TForm1 = class(TForm)
  ...
  private
   procedure WMGetMinMaxInfo(var Info: TWMGetMinMaxInfo);
                              message WM_GETMINMAXINFO;
  ...
 end;
...
implementation

procedure TForm1.WMGetMinMaxInfo(var Info: TWMGetMinMaxInfo);
begin
 with Info.MinMaxInfo^ do
  begin
   ptMinTrackSize.x := 150;
   ptMaxTrackSize.x := 300;
   ptMinTrackSize.y := 100;
   ptMaxTrackSize.y := 200;
   ptMaxPosition.x:=BoundsRect.Left;
   ptMaxPosition.y:=BoundsRect.Top;
  end;
 inherited;
end;
```

WM_GETTEXT

Посылается, чтобы скопировать текст, связанный с окном, в указанный буфер.

Определение

```
WM_GETTEXT
 wParam = (WPARAM) cchTextMax;
 lParam = (LPARAM) lpszText;
```

Параметры

cchTextMax указывает минимальное число символов, которые должны быть скопированы, включая нулевой конечный символ.

lpszText указывает на буфер, принимающий текст.

Возвращаемое значение

Возвращает число скопированных символов.

Действие по умолчанию

Копируется текст, связанный с окном, в указанный буфер и возвращается число скопированных символов.

Примечания

Для всех окон редактирования текст — это содержимое окна. Для выпадающих списков текст — это выделенный текст. Для кнопок текст — это имя кнопки. Для остальных оконных компонентов текст — это заголовок окна.

Для копирования обогащенного текста, превышающего 64K, надо использовать сообщения EM_STREAMOUT или EM_GETSELTEXT.

WM_SETFONT

Посылается, чтобы задать шрифт, который будет использоваться для текста указанного окна.

Определение

```
WM_SETFONT
wParam = (WPARAM) hfont;              // дескриптор шрифта
lParam = MAKELPARAM(fRedraw, 0);      // флаг перерисовки
```

Параметры

hfont — указатель на шрифт. Если hfont задан равным **NULL,** будет использоваться системный шрифт по умолчанию.

fRedraw — при значении **true** компонент будет немедленно перерисован после изменения шрифта.

Возвращаемое значение

Сообщение не возвращает никакого значения.

Примечания

Сообщение можно посылать диалоговым окнам и любым оконным компонентам. При изменении шрифта размер окна не изменяется. Так что, если необходимо изменять размер окна, чтобы в нем поместился текст с новым шрифтом, это надо делать отдельно.

Пример

Приведенный ниже оператор изменяет шрифт окна редактирования **Edit1** формы **Form2,** заменяя его выбранным пользователем в диалоге выбора шрифта, инициированном компонентом **FontDialogl:**

```
if (FontDialog1.Execute) then
 SendMessage(Form2.Edit1.Handle, WM_SETFONT,
                                  FontDialog1.Font.Handle, 0);
```

WM_SETTEXT

Посылается, чтобы задать текст указанного окна.

Определение

```
WM_SETTEXT
 wParam = 0;
 lParam = (LPARAM) (LPCTSTR)lpsz;
```

Параметры

lpsz — указатель на строку текста окна с нулевым конечным символом.

Возвращаемое значение

Возвращает true, если текст установлен. В противном случае возвращает **false** (для окна редактирования), **LB_ERRSPACE** (для списка) или CB_ERRSPACE (для выпадающего списка) если не хватает места для размещения текста. Возвращает **CB_ERR,** если сообщение посылается выпадающему списку без окна редактирования.

Действие по умолчанию

Устанавливает и отображает текст окна.

Примечания

Для всех окон редактирования текст — это содержимое окна. Для выпадающих списков текст — это выделенный текст. Для кнопок текст — это имя кнопки. Для остальных оконных компонентов текст — это заголовок окна.

Сообщение не изменяет текущее выделение в списках. Чтобы выделялся элемент списка, соответствующий тексту, надо использовать сообщение **CB_SELECTSTRING.**

3.1.6 AnsiString — тип строк

В C++Builder тип строк **AnsiString** реализован как класс, объявленный в файле vcl/dstring.h и аналогичный типу длинных строк в Delphi. Это строки с нулевым символом в конце. При объявлении переменные типа **AnsiString** инициализируются пустыми строками.

Для **AnsiString** определены операции отношения ==, !=, >, <, >=, <=. Сравнение производится с учетом регистра. Сравниваются коды символов, начиная с первого, и если очередные символы не одинаковы, строка, содержащая символ с меньшим кодом считается меньше. Если все символы совпали, но одна строка длиннее и в ней имеются еще символы, то она считается больше, чем более короткая.

Для **AnsiString** определены операции присваивания =, += и операция склеивания строк (конкатенации) +. Определена также операция индексации [ ]. Индексы начинаются с 1. Например, если S1 = "Привет", то **Sl[l]** вернет 'П', **Sl[2]** вернет 'р' и т.д.

Работа с классом **AnsiString** рассмотрена в гл. 2 в разд. 2.5.2. Основные методы класса **AnsiString** (в описаниях методов через **S1** обозначена строка, метод которой используется):

Метод	Синтаксис / Описание
AnsiCompare	**int AnsiCompare(const** AnsiString& **rhs)** const Сравнивает данную строку **S1** с rhs с учетом регистра. Сравнение зависит от текущих установок Windows и может отличаться от сравнения, осуществляемого операциями сравнения. Возвращает значение > 0 при **S1** > rhs, значение < 0 при S1 < rhs и значение 0 при Sl = **rhs**
AnsiCompareIC	**int AnsiCompareIC(const** AnsiString& rhs) const Осуществляет сравнение, аналогичное AnsiCompare, но без учета регистра
AnsiLastChar	**char* AnsiLastChar()** const Возвращает указатель на последний значащий символ. Поддерживает многобайтные символы

Метод	Синтаксис / Описание	
AnsiPos	int **AnsiPos(const AnsiString& subStr)** const Возвращает индекс первого символа первого вхождения subStr в S1. Индексы начинаются с 1. Если subStr не содержится в S1, возвращается 0. В отличие от Pos поддерживает многобайтные символы	
AnsiString	**AnsiString(аргумент)** Конструктор класса. В зависимости от типа аргумента создает:	
	Аргумент	Создает
	Отсутствует	Пустую строку
	const char* **src**	Строку с нулевым символом в конце из массива символов
	const AnsiString& **src**	Копию AnsiString **src**
	const char* **src**, unsigned char **len**	Строку с нулевым символом в конце, являющуюся копией первых len символов из src
	const wchar_t* **src**	Строку с нулевым символом в конце из массива src символов типа wchar_t
	int src	Строку с нулевым символом в конце из массива src целых значений символов
	double src	Строку с нулевым символом в конце из массива **src** значений символов с плавающей запятой; преобразуются первые 15 значащих разрядов
c_str	char* c_str()const Возвращает указатель на строку с нулевым символом в конце, содержащую те же символы, что в AnsiString	
CurrToStr	static AnsiString **CurrToStr(Currency** value) Преобразует значение value типа Currency в строку	
CurrToStrF	static AnsiString CurrToStrF(Currency value, TStringFloatFormat format, int digits) Преобразует значение value типа Currency в строку, используя указанный формат преобразования чисел с плавающей запятой (см. разд. 3.1.8). Параметр определяет задаваемое число разрядов. Функция соответствует функции CurrToStrF с заданной точностью 19 разрядов	
Delete	void Delete(int index, int count) Удаляет из строки, начиная с позиции index число символов, равное count	

Метод	Синтаксис / Описание
FloatToStrF	static AnsiString FloatToStrF(long double value, TStringFloatFormat format, int precision, int digits) Преобразует значение value с плавающей запятой в строку, используя указанный формат (см. разд. 3.1.8). Параметры precision и digits задают точность и число разрядов. Точность должна задаваться не более 7 для типа float, не более 15 для double и не более 18 для Extended. Число разрядов зависит от выбранного формата
Format	static AnsiString **Format(const** AnsiString& format, const TVarRec *args, int size) Формирует строку, используя строку формата format и массиваргументов**args**
FormatFloat	static AnsiString FormatFloat(const AnsiString& format, const long double& value) Преобразует значение value с плавающей запятой в строку, используя указанный формат format
Insert	void Insert(const AnsiString& str, int index) Вставляет в строку подстроку str, начиная с индекса index
IntToHex	static AnsiString IntToHex(int value, int digits) Преобразует значение value в строку, содержащую минимум digits шестнадцатеричных цифр
IsDelimiter	**bool IsDelimiter(const** AnsiString& delimiters, int index) const Возвращает true, если символ с индексом index является одним из разделителей, указанных в строке delimiters. Работает и для многобайтных символов
IsEmpty	bool IsEmpty() const Возвращает true, если строка пустая

LastDelimiter	int **LastDelimiter(const** AnsiString& delimiters) const
	Возвращает последний из символов строки, входящих в строку разделителей delimiters. Например, если
	`AnsiString s = "c:\\filename.ext";`
	то
	`s.LastDelimiter("\\.:" );`
	вернет 12 (индекс символа точки)

Length	int **Length()** const Возвращает число символов в строке
LowerCase	AnsiString **LowerCase()** const Возвращает строку, в которой все символы приведены к нижнему регистру. Не влияет на исходную строку
Pos	int Pos(const AnsiString& subStr) const Возвращает индекс первого символа первого вхождения subStr в S1. Индексы начинаются с 1. Если subStr не содержится в S1, возвращается 0. В отличие от AnsiPos не поддерживает многобайтные символы

Метод	Синтаксис / Описание
SetLength	void SetLength(int **newLength**) Усекает строку до newLength символов. Если исходная строка короче, то она не увеличивается
StringOfChar	static AnsiString **StringOfChar(char** ch, int count) Возвращает строку, в которой символ ch повторен count раз. Например, `AnsiString s = AnsiString::StringOfChar('A', 10);` задаст строке s значение "AAAAAAAAAA"
Substring	**AnsiString SubString(int** index, int count) const Возвращает подстроку, начинающуюся с символа в позиции index и содержащую count символов
ToDouble	double ToDouble() const Преобразует строку в число с плавающей запятой. Если строка не соответствует формату числа с плавающей запятой, генерируется исключение EConvertError
ToInt	int **ToInt()** const Преобразует строку в целое число. Если строка не соответствует формату целого числа, генерируется исключение EConvertError
ToIntDef	int ToIntDef(int defaultValue) const Преобразует строку в целое число. Если строка не соответствует формату целого числа, возвращается значение по умолчанию defaultValue
Trim	AnsiString Trim() const Возвращает строку, соответствующую исходной, но без пробельных символов до и после значащих символов
TrimLeft	AnsiString TrimLeft() const Возвращает строку, соответствующую исходной, но без начальных пробельных символов
TrimRight	AnsiString TrimRight() const Возвращает строку, соответствующую исходной, но без заключительных пробельных символов
Unique	void Unique() Делает строку уникальной, т.е. устанавливает число ссылок на нее (refcnt) в 1. Таким образом, на нее ссылается только один объект
UpperCase	AnsiString **UpperCase()** const Возвращает строку, в которой все символы приведены к верхнему регистру. Не влияет на исходную строку
WideChar	wchar_t* WideChar(wchar_t* dest, int destSize) const Преобразует строку в массив символов dest типа **wchar_t** и возвращает указатель на этот массив

Метод	Синтаксис / Описание
WideCharBuf-Size	**int WideCharBufSize() const** **Возвращает размер буфера, требуемого для функции WideChar**

3.1.7 Тип данных TDateTime

Тип, используемый функциями и процедурами, работающими с датами и временем

Заголовочный файл *systdate.h*.

Описание

Тип TDateTime был введен в Object Pascal как число с плавающей запятой, целая часть которого содержит число дней, отсчитанное от некоторого начала календаря, а дробная часть равна части 24-часового дня, т.е. характеризует время и не относится к дате. Для 32-разрядных версий за начало календаря принята дата 00 часов 30 декабря 1899 года. Прибавление к значению типа **TDateTime** целого числа D равносильно увеличению даты на D дней. Разность двух значений типа **TDateTime** дает разность двух дат с точностью до долей дня.

В C++Builder тип **TDateTime** реализован в виде класса. Впрочем, его можно использовать точно так же, как в Object Pascal. Но в действительности, возможности класса **TDateTime** шире. В частности, можно использовать конструктор, инициирующий переменную заданным значением **TDateTime.** Например, оператор

```
TDateTime T(Now());
```

объявляет переменную T и передает в нее текущую дату и время с помощью функции Now.

В классе определен ряд операций: "+" и " - сложение и вычитание числа дней, включая дробную часть дня, "++" и " — прибавление и вычитание одного дня, **double** — перевод в форму действительного числа, типичную для Delphi, операции отношения и ряд других.

Можно также использовать ряд полезных функций-элементов данного класса:

Функция-элемент	Объявление / Описание
CurrentDate	TDateTime **CurrentDate()** Возвращает текущую дату с нулевым временем
CurrentDateTime	TDateTime CurrentDateTime() Возвращает текущую дату и время
CurrentTime	TDateTime **CurrentTime()** Возвращает текущее время с нулевой датой
DateString	**AnsiString DateString()** const Возвращает дату объекта TDateTime в виде строки, отформатированной в соответствии с глобальной переменной **ShortDateFormat**
DateTimeString	AnsiString **DateTimeString()** const Возвращает дату и время объекта TDateTime в виде строки. Дата форматируется в соответствии с глобальной переменной ShortDateFormat. Время форматируется в соответствии с глобальной переменной LongTimeFormat

Функция-элемент	Объявление / Описание
DayOfWeek	int DayOfWeek() const Возвращает день недели объекта TDateTime (1 — воскресенье, 7 — суббота)
DecodeDate	void DecodeDate(unsigned short* year, unsigned short* month, unsigned short* day) const Выделяет год year, месяц month и день day из объекта TDateTime
DecodeTime	void DecodeTime(unsigned short* hour, unsigned short* min, unsigned short* sec, unsigned short* msec) const Выделяет час hour, минуту min, секунду sec и миллисекунды msec из объекта TDateTime
FileDate	int FileDate() const Возвращает объект TDateTime, переведенный в формат дат и времени DOS
FileDateToDateTime	TDateTime FileDateToDateTime(int fileDate) Переводит в объект TDateTime дату и время fileDate, заданные в формате DOS
FormatString	AnsiString FormatString(const AnsiString& format) Возвращает строку объекта TDateTime, сформированную по строке форматирования format
TimeString	AnsiString TimeString() const Возвращает строку, содержащую время, записанное в объекте, отформатированное с помощью глобальной переменной LongTimeFormat

Примеры

```
Label1->Caption = T;                    // 26.05.2002 18:44:07
Label2->Caption = T.DateTimeString();
Label3->Caption = T.DateString();       // 26.05.2002
Label4->Caption = T.TimeString();       //18:44:07
```

Два первых оператора отобразят одно и то же: дату и время, в виде, показанном в комментарии к первому из них. Третий и четвертый операторы отобразят соответственно дату и время, вид которых указан в комментариях.

3.1.8 TStringFloatFormat - тип

В ряде методов и функций тип TStringFloatFormat определяет формат представления чисел строкой.

Определение

```
enum TStringFloatFormat {sffGeneral, sffExponent, sffFixed,
                         sffNumber, sffCurrency};
```

Различные значения формата означают следующее:

Значение	Описание
sffGeneral	Значение преобразуется в наиболее компактное из двух форматов: с фиксированной точкой или научного формата. Младшие нулевые разряды усекаются. Десятичная точка появляется только при необходимости. Формат с фиксированной точкой используется только при числе цифр целой части большем не больше указанной точности и при значениях не меньше 0.00001. В остальных случаях используется научный формат с минимальным числом цифр в степени порядка (от 0 до 4).
sffExponent	Научный формат. Значение преобразуется в строку вида "−d.ddd...E+dddd". Символ '−' записывается только для отрицательных чисел. Перед десятичной точкой записывается всегда одна цифра. Общее число цифр (включая цифру перед точкой) определяется заданной точностью. После символа 'E' всегда ставится знак + или −. Число цифр в степени (порядок числа) лежит в пределах от 0 до 4.
sffFixed	Формат с фиксированной точкой. Значение преобразуется в строку вида "−ddd.ddd...". Символ '−' записывается только для отрицательных чисел. Перед десятичной точкой записывается пот крайней мере одна цифра. Число цифр поле точки определяется заданным числом разрядов (от 0 до 18). Если число цифр слева от точки должно быть больше заданной точности, используется научный формат.
sffNumber	Числовой формат. Значение преобразуется в строку вида "−d,ddd,ddd.ddd...". Совпадает с форматом **sffFixed** за исключением наличия разделителей после каждых трех разрядов в целой части.
sffCurrency	Монетарный формат для представления чисел, отображающих денежные суммы. Определяется установками Windows (глобальными переменными **CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator**). Число цифр после десятичной точки определяется заданным числом разрядов (от 0 до 18).

3.2 Математические функции

3.2.1 Константы, используемые в математических выражениях

Константа	Описание	Значение
M_1_PI	1 / π	0.318309886183790671538
M_1_SQRTPI	корень из 1 / я	0.564189583547756286948
M_2_PI	2 / л	0.636619772367581343076
M_2_SQRTPI	2 / корень из я	1.12837916709551257390
M_E	число *e*	2.71828182845904523536
M_LN10	ln(10) — логарифм натуральный от 10	2.30258509299404568402
M_LN2	ln(2) — логарифм натуральный от 2	0.693147180559945309417
M_LOG10E	$\log_{10}$(e) — логарифм десятичный от e	0.434294481903251827651

Константа	Описание	Значение
M_LOG2E	$\log_2(e)$ — логарифм по основанию 2 от e	1.44269504088896340736
M_PI	число я	3.14159265358979323846
M_PI_2	$\pi / 2$	1.57079632679489661923
M_PI_4	$\kappa / 4$	0.785398163397448309616
M_SQRT_2	корень из 2, деленный на 2	0.707106781186547524401
M_SQRT2	корень из 2	1.41421356237309504880

3.2.2 Арифметические и алгебраические функции

Функция	Синтаксис	Описание	Файл
abs	int abs(int x)	абсолютное значение	*stdlib.h*

cabs	double cabs(struct complex z)
	struct complex {
	double x, y;
	};
		модуль комплексного числа z	*math.h*

cabsl	long double cabsl(
	struct _complexl z)
	struct _complex {
	long double x, y;
	};
		модуль комплексного числа z	*math.h*

ceil	double ceil(double x)	округление вверх: наименьшее целое, не меньшее X	*math.h*
Ceil	int Ceil(Extended X);	округление вверх: наименьшее целое, не меньшее X	*Math.hpp*
ceill	long double ceill(long double x)	округление вверх: наименьшее целое, не меньшее X	*math.h*
_crotl	unsigned char _crotl(unsigned char val, int count)	циклический сдвиг value влево на count битов	*stdlib.h*
_crotr	unsigned char _crotr(unsigned char val, int count)	циклический сдвиг value вправо на count битов	*stdlib.h*

div	div_t div(int numer, int denom)
	typedef struct {
	int quot; // *частное*
	int rem; // *остаток*
	} div_t;
		целочисленное деление numer / denom	*math.h*

exp	double exp(double x)	экспонента	*math.h*
expl	long double expl(long double x)	экспонента	*math.h*
fabs	double fabs(double x)	абсолютное значение	*math.h*

Функция	Синтаксис	Описание	Файл
fabsl	long double fabsl(long double x)	абсолютное значение	*math.h*
floor	double floor(double x)	округление вниз: наибольшее целое, не большее X	*math.h*
Floor	int Floor(Extended X);	округление вниз: наибольшее целое, не большее X	*Math.hpp*
floorl	long double floorl(long double x)	округление вниз: наибольшее целое, не большее X	*math.h*
fmod	double fmod(double x, double y)	остаток от деления x / y	*math.h*
fmodl	long double fmodl(long double x, long double y)	остаток от деления x / y	*math.h*
frexp	double frexp(double x, int *exponent)	разделяет x на мантиссу (возвращает) и степень exponent	*math.h*
Frexp	void Frexp(Extended X, Extended &Mantissa, int &Exponent)	разделяет X на мантиссу Mantissa и степень Exponent	*Math.hpp*
frexpl	long double frexpl(long double x, int *exponent)	разделяет x на мантиссу (возвращает) и степень exponent	*math.h*
IntPower	Extended IntPower(Extended Base, int Exponent)	возводит Base в целую степень Exponent	*Math.hpp*
labs	long labs(long int x)	абсолютное значение	*stdlib.h*
ldexp	double ldexp(double x, int exp)	$x \cdot 2^{exp}$	*math.h*
Ldexp	Extended Ldexp(Extended X, int P)	$x \cdot 2^{P}$	*Math.hpp*
ldexpl	long double ldexpl(long double x, int exp)	$x \cdot 2^{exp}$	*math.h*
ldiv	typedef struct { long int quot; // целое long int rem; // остаток } ldiv_t; ldiv_t ldiv(long int numer, long int denom)	целочисленное деление: numer / denom; quot — результат rem — остаток	*math.h, stdlib.h*
LnXP1	Extended LnXPl(Extended X)	натуральный логарифм (X + 1)	*Math.hpp*
log	double log(double x)	натуральный логарифм	*math.h*
log10	double loglO(double x)	десятичный логарифм	*math.h*
Log10	Extended LoglO(Extended X)	десятичный логарифм	*Math.hpp*
loglOl	long double log10l(long double x)	десятичный логарифм	*math.h*

Функция	Синтаксис	Описание	Файл
Log2	Extended Log2(Extended X)	логарифм по основанию 2	*Math.hpp*
logl	long double logl(long double x)	натуральный логарифм	*math.h*
LogN	Extended LogN(Extended Base, Extended X)	логарифм X по основанию Base	***Math.hpp***
_lrotl	unsigned long lrotl(unsigned long val, int count)	циклический сдвиг val влево на count битов	*stdlib.h*
_lrotr	unsigned long _lrotr(unsigned long val, int count)	циклический сдвиг val вправо на count битов	*stdlib.h*
max	max(a, b)	макрос возвращает максимальное значение из a и b любых типов	***stdlib.h***
min	min(a, b)	макрос возвращает минимальное значение из a и b любых типов	***stdlib.h***
modf	double modf(double x, double *ipart)	разделяет x на целую часть ipart и возвращаемую дробную часть	***math.h***
modfl	long double modfl(long double x, long double *ipart)	разделяет x на целую часть ipart и возвращаемую дробную часть	***math.h***
poly	double poly(double x, int degree, double coeffs[])	полином от x степени degree с коэффициентами coeffs	***math.h***
Poly	Extended Poly(Extended X, const double * Coefficients, const int Coefficients_Size)	полином от X степени Coefficients_Size с коэффициентами Coefficients	***Math.hpp***
polyl	long double polyl(long double x, int degree, long double coeffs[])	полином от x степени degree с коэффициентами coeffs	***math.h***
pow	double pow(double x, double y)	x^y	***math.h***
Power	Extended Power(Extended Base, Extended Exponent)	возводит Base в степень Exponent	***Math.hpp***
powl	long double powl(long double x, long double y)	x^y	***math.h***
_rotl	unsigned short _rotl(unsigned short value, int count)	циклический сдвиг value влево на count битов	***stdlib.h***
_rotr	unsigned short _rotr(unsigned short value, int count)	циклический сдвиг value вправо на count битов	***stdlib.h***
sqrt	double sqrt(double x)	корень квадратный	***math.h***
sqrtl	long double sqrtl(long double x)	корень квадратный	***math.h***

Комментарии

При работе с математическими функциями надо иметь в виду, что файлы *math.h* и *Math.hpp* в C++Builder автоматически не подключаются к модулю вашего приложения. Поэтому для использования описанных в этих файлах функций необходимо вручную вводить директивы

```
#include <math.h>
#include <Math.hpp>
```

Функции **exp, expl, ldexp, ldexpl в** случае выхода аргумента за диапазон допустимых значений генерируют ошибку **ERANGE.**

Функции **log, log10, loglOl, logl** в случае отрицательного аргумента генерируют ошибку **ERANGE,** а при нулевом аргументе — **EDOM.**

Функции **pow** и **powl** генерируют ошибку **EDOM,** если x < 0 и y не является целым числом, а также если x = 0 и y <= 0. Возможно также появление ошибки **ERANGE.**

Функции **sqrt и sqrtl** генерируют ошибку **EDOM,** если x < 0.

Функции файла *Math.hpp* в основном повторяют возможности функций файла *math.h,* но для типа **Extended.**

3.2.3 Тригонометрические функции

Функция	Синтаксис	Описание	Файл
acos	double acos(double x)	арккосинус	*math.h*
acosl	long double acosl(long double x)	арккосинус	*math.h*
ArcCos	Extended ArcCos(Extended X)	арккосинус	*Math.hpp*
ArcCosh	Extended ArcCosh(Extended X)	арккосинус гиперболический	*Math.hpp*
ArcSin	Extended ArcSin(Extended X)	арксинус	*Math.hpp*
ArcSinh	Extended ArcSinh(Extended X)	арксинус гиперболический	*Math.h*
ArcTan2	Extended ArcTan2(Extended Y, Extended X)	арктангенс (Y / X)	*Math.hpp*
ArcTanh	Extended ArcTanh(Extended X)	арктангенс гиперболический	*Math.hpp*
asin	double asin(double x)	арксинус	*math.h*
asinl	long double asinl(long double x)	арксинус	*math.h*
atan	double atan(double x)	арктангенс	*math.h*
atan2	double atan2(double y, double x	арктангенс y / x	*math.h*
atan2l	long double atan21(long double y, long double x)	арктангенс y / x	*math.h*
atanl	long double atanl(long double x)	арктангенс	*math.h*
cos	double cos(double x)	косинус	*math.h*
cosh	double cosh(double x)	косинус гиперболический	*math.h*
Cosh	Extended Cosh(Extended X)	косинус гиперболический	*Math.hpp*

Функция	Синтаксис	Описание	Файл
coshl	long double coshl(long double x)	косинус гиперболический	*math.h*
cosl	long double cosl(long double x)	косинус	*math.h*
Cotan	Extended Cotan(Extended X)	котангенс	*Math.hpp*
CycleToRad	Extended CycleToRad(Extended Cycles)	вычисляет угол в радианах по его значению в периодах Cycles: 2π • Cycles.	*Math.hpp*
DegToRad	Extended DegToRad(Extended Degrees)	вычисляет угол в радианах по его значению в градусах Degrees: Degrees • π / 180.	*Math.hpp*
hypot	double hypot(double x, double y)	гипотенуза треугольника с катетами x и y	*math.h*
Hypot	Extended Hypot(Extended X, Extended Y)	расчет гипотенузы по катетам X и Y	*Math.hpp*
hypotl	long double hypotl(long double x, long double y)	гипотенуза треугольника с катетами x и y	*math.h*
RadToCycle	Extended RadToCycle(Extended Radians)	вычисляет угол в периодах по его значению в радианах Radians: Radians / (2л).	*Math.hpp*
RadToDeg	Extended RadToDeg(Extended Radians)	вычисляет угол в градусах по его значению в радианах Radians: Radians • 180 / я.	*Math.hpp*
sin	double sin(double x)	синус	*math.h*
SinCos	void SinCos(Extended Theta, Extended &Sin, Extended &Cos)	расчет синуса Sin и косинуса Cos угла Theta	*Math.hpp*
Sinh	Extended Sinh(Extended X)	синус гиперболический	*Math.hpp*
sinh	double sinh(double x)	синус гиперболический	*math.h*
sinhl	long double sinhl(long double x)	синус гиперболический	*math.h*
sinl	long double sinl(long double x)	синус	*math.h*
Tan	Extended Tan(Extended X)	тангенс	*Math.hpp*
tan	double tan(double x)	тангенс	*math.h*

Функция	Синтаксис	Описание	Файл
Tanh	Extended Tanh(Extended X)	тангенс гиперболический	*Math.hpp*
tanh	double tanh(double x)	тангенс гиперболический	*math.h*
tanhl	long double tanhl(long double x)	тангенс гиперболический	*math.h*
tanl	long double tanl(long double x)	тангенс	*math.h*

Комментарии

При работе с тригонометрическими функциями надо иметь в виду, что файлы *math.h* и *Math.hpp* в C++Builder автоматически не подключаются к модулю вашего приложения. Поэтому для использования описанных в этих файлах функций необходимо вручную вводить директивы

```
#include <math.h>
#include <Math.hpp>
```

Во всех тригонометрических функциях угол задается в радианах. Пересчет угла в радианы из значения, заданного в градусах или периодах, позволяют осуществить функции **DegToRad и CycleToRad.** Например, оператор

```
double Rad = DegToRad(90);
```

заносит в переменную Rad значение угла 90 градусов в радианах. То же самое значение заносит в переменную **Rad** оператор

```
double Rad = CycleToRad(0.25);
```

в котором значение угла задано в периодах (четверть периода). Следующее выражение вычисляет синус 90 градусов:

```
double S = sin(DegToRad(90));
```

Все обратные тригонометрические функции вычисляют главные значения: **acos и acosl** — в диапазоне **[0,** я], **asin, asinl, atan, atan'2, atan21, atanl** — в диапазоне [−π/2, я/2]. Результат возвращается в радианах. Пересчет угла в радианах в значения градусов или долей периода позволяют осуществить функции **RadToDeg и RadToCycle.** Например, операторы

```
double A = atan(T);
double A1 = RadToDeg(atan(T));
double A2 = RadToCycle(atan(T));
```

вычисляют арктангенс Т в радианах (А), в градусах (А1) и в долях периода (А2).

В функциях **acos, acosl, asin, asinl,** если заданный аргумент не попадает в диапазон значений [-1, + 1], происходит ошибка выхода за пределы области определения **(EDOM).**

В гиперболических функциях **cosh, coshl, sinh, sinhl,** если заданный аргумент слишком велик, происходит ошибка выхода за диапазон допустимых значений **(ERANGE).**

3.2.4 Генерация псевдослучайных чисел

Функция	Синтаксис / Описание	Файл
_lrand	**long lrand(void)** **Псевдослучайное целое, диапазон от 0 до 2^{31} - 1**	*stdlib.h*

Функция	Синтаксис / Описание	Файл
rand	int rand(void) Псевдослучайное целое, диапазон от 0 до RAND_MAX	*stdlib.h*
RandG	Extended RandG(Extended Mean, Extended StdDev) Псевдослучайные числа, распределенные по нормальному закону; Mean — математическое ожидание, StdDev — среднее квадратичное отклонение	*Math.hpp*
random	int **random(int num)** Псевдослучайное целое, диапазон от 0 до num - 1	*stdlib.h*
randomize	void **randomize(void)** Рандомизация генераторов (кроме RandG) случайной величиной	*stdlib.h*
Randomize	void **Randomize(void)** Рандомизация RandG случайной величиной	*Math.hpp*
srand	void **srand(unsigned** seed) Рандомизация генераторов (кроме RandG) числом seed	*stdlib.h*

3.2.5 Функции обработки статистических данных

Приведенные ниже функции обрабатывают данные, хранящиеся в массиве **Data,** в котором максимальное значение индекса равно **Data_Size.**

Функция	**Синтаксис / Описание**	**Файл**
MaxIntValue	**int MaxIntValue(const int * Data, const int Data_Size)** **Максимальное значение**	***Math.hpp***
MaxValue	**double MaxValue(const double * Data, const int Data_Size)** **Максимальное значение**	***Math.hpp***
Mean	**Extended Mean(const double * Data, const int Data_Size)** **Среднее значение (математическое ожидание)**	***Math.hpp***
MeanAndStd-Dev	**void MeanAndStdDev(const double * Data, const int Data_Size, Extended &Mean, Extended &StdDev)** **Среднее значение Mean и среднее квадратичное отклонение StdDev**	***Math.hpp***
MinIntValue	**int MinIntValue(const int * Data, const int Data_Size)** **Минимальное значение**	***Math.hpp***
MinValue	**double MinValue(const double * Data, const int Data_Size)** **Минимальное значение**	***Math.hpp***

Функция	Синтаксис / Описание	Файл
MomentSkewKurtosis	void MomentSkewKurtosis(const double * Data, const int Data_Size, Extended &M1, Extended &M2, Extended &M3, Extended &M4, Extended &Skew, Extended &Kurtosis) Первые четыре момента M1, M2, M3, M4, коэффициент асимметрии Skew, эксцесс Kurtosis	*Math.hpp*
Norm	Extended Norm(const double * Data, const int Data_Size) Эвклидова норма: корень из суммы квадратов	*Math.hpp*
PopnStdDev	Extended PopnStdDev(const double * Data, const int Data_Size) Смещенная оценка среднего квадратичного отклонения	*Math.hpp*
PopnVariance	Extended PopnVariance(const double * Data, const int Data_Size) Смещенная оценка дисперсии (см. Variance)	*Math.hpp*
StdDev	Extended StdDev(const double * Data, const int Data_Size) Несмещенная оценка среднего квадратичного отклонения	*Math.hpp*
Sum	Extended Sum(const double * Data, const int Data_Size) Сумма значений	*Math.hpp*
SumInt	int SumInt(const int * Data, const int Data_Size) Сумма значений	*Math.hpp*
SumOfSquares	Extended SumOfSquares(const double * Data, const int Data_Size) Сумма квадратов значений	*Math.hpp*
SumsAndSquares	void SumsAndSquares(const double * Data, const int Data_Size, Extended &Sum, Extended &SumOfSquares) Сумма Sum и сумма квадратов значений SumOfSquares	*Math.hpp*
TotalVariance	Extended TotalVariance(const double * Data, const int Data_Size) Сумма квадратов отклонений от среднего значения	*Math.hpp*
Variance	Extended Variance(const double * Data, const int Data_Size) Несмещенная оценка дисперсии (см. PopnVariance)	*Math.hpp*

3.2.6 Функции управления FPU

Функция	Синтаксис / Описание
_clear87	unsigned int _clear87 (void) Очищает слово состояния FPU
_clearfp	unsigned int _clearfp (void) Очищает слово состояния FPU
_control87	unsigned int _control87(unsigned int newcw, unsigned int mask) Обеспечивает доступ к управляющему слову FPU
_controlfp	unsigned int _controlfp(unsigned int newcw, unsigned int mask) Обеспечивает доступ к управляющему слову FPU
_fpreset	void _fpreset(void) Повторно инициализирует пакет математики с плавающей запятой
Get8087CW	Word Get8087CW(void) Возвращает управляющее слово FPU
GetExceptionMask	TFPUExceptionMask GetExceptionMask(void) Возвращает маску исключений
GetPrecisionMode	TFPUPrecisionMode GetPrecisionMode(void) Возвращает значение, соответствующее текущим значениям битов управления точностью управляющего слова FPU, в виде перечислимого тип TFPUPrecisionMode
GetRoundMode	TFPURoundingMode GetRoundMode(void) Возвращает значение, соответствующее текущим значениям битов управления округлением управляющего слова FPU, в виде перечислимого тип TFPURoundingMode
Set8087CW	void Set8087CW(Word NewCW) устанавливает управляющее слово FPU
SetExceptionMask	TFPUExceptionMask SetExceptionMask(void TFPUExceptionMask Mask) устанавливает маску исключений
SetPrecisionMode	TFPUPrecisionMode SetPrecisionMode(const TFPUPrecisionMode Precision) Возвращает и устанавливает значение, соответствующее значениям битов управления точностью управляющего слова FPU, в виде перечислимого тип TFPUPrecisionMode
SetRoundMode	TFPURoundingMode SetRoundMode(const TFPURoundingMode RoundMode) Возвращает и устанавливает значение, соответствующее текущим значениям битов управления округлением управляющего слова FPU, в виде перечислимого тип TFPURoundingMode

Функция	Синтаксис / Описание
_status87	unsigned int _status87(void) Возвращает текущее значение слова состояния FPU
_statusfp	unsigned int statusfp (void) Возвращает текущее значение слова состояния FPU

Комментарий

См. подробнее об FPU (floating-point unit) в разд. 1.9.3 и в описаниях функций в гл. 4.

3.3 Преобразование типов данных

3.3.1 Функции взаимного преобразования чисел и строк

3.3.1.1 Функции взаимного преобразования чисел и строк типа char *

Функция	Синтаксис / Преобразует	Файл
_atoi64	__int64 _atoi64(const char *s) Строку s в целое	*stdlib.h*
_atold	long double _atold(const char *s) Строку s в число с плавающей запятой	*math.h*
_i64toa	char *_i64toa(__int64 value, char *strP, int radix) Целое value в строку; radix — основание (от 2 до 36)	***stdlib.h***
_itow	wchar_t *_itow(int value, wchar_t *string, int radix) Целое value в строку string по основанию radix	***stdlib.h***
_ltoa	char *_ltoa(long value, char *string, int radix) Целое value в строку; radix — основание (от 2 до 36)	*stdlib.h*
_strtold	long double _strtold(const char *s, char **endptr) Строки s в действительное число	***stdlib.h***
_ui64toa	char *_ui64toa(unsigned__int64 value, char *strP, int radix) Целое value в строку; radix — основание (от 2 до 36)	*stdlib.h*
_ultow	wchar_t *_ultow(unsigned long Value, wchar_t *string, int radix) Целое value в строку string по основанию radix	***stdlib.h***
_wcstold	long double _wcstold(const wchar_t *s, wchar_t **endptr) Строку s в действительное число	*stdlib.h*
_wtof	double _wtof(const wchar_t *s) Строку s в число с плавающей запятой	***math.h***
_wtoi	int _wtoi(const wchar_t *s) Строку s в целое	***stdlib.h***

Функция	Синтаксис / Преобразует	Файл
_wtoi64	__int64 _wtoi64(const wchar_t *s) Строку s в целое	stdlib.h
_wtol	long _wtol(const wchar_t *s) Строку s в целое	stdlib.h
_wtold	long double _wtold(const wchar_t *s) Строку s в число с плавающей запятой	math.h
atof	double atof(const char *s) Строку s в число с плавающей запятой	stdlib.h, math.h
atoi	int atoi(const char *s) Строку s в целое	stdlib.h
atol	long atol(const char *s) Строку s в целое	stdlib.h
ecvt	char *ecvt(double value, int ndig, int *dec, int *sign) Число с плавающей запятой value в строку с числом цифр ndig; dec сохраняет позицию десятичной точки, sign — знак	stdlib.h
fcvt	char *fcvt(double value, int ndig, int *dec, int *sign) Число с плавающей запятой value в строку с числом цифр ndig; dec сохраняет позицию десятичной точки, sign — знак	stdlib.h
gcvt	char *gcvt(double value, int ndec, char *buf) value в строку buf с числом цифр ndec	stdlib.h
itoa	char *itoa(int value, char *string, int radix) Целое value в строку string по основанию radix	stdlib.h
strtod	double strtod(const char *s, char **endptr) Строку s в действительное число	stdlib.h
strtol	long strtol(const char *s, char **endptr, int radix) Строку s в длинное целое	stdlib.h
strtoul	unsigned long strtoul(const char *s, char **endptr, int radix) Строку s в unsigned long по основанию radix	stdlib.h
ultoa	char *ultoa(unsigned long value, char *string, int radix) Целое value в строку string по основанию radix	stdlib.h
wcstod	double wcstod(const wchar_t *s, wchar_t **endptr) Строку s в действительное число	stdlib.h
wcstol	long wcstol(const wchar_t *s, wchar_t **endptr, int radix) Строку s в длинное целое	stdlib.h
wcstoul	unsigned long wcstoul(const wchar_t *s, wchar_t **endptr, int radix) Строку s в unsigned long по основанию radix	stdlib.h

Комментарии

Функции преобразования строки в число требуют, чтобы строка была записана в формате чисел соответствующего типа. Преобразование прерывается, когда функция встречает первый символ, не соответствующий требуемому формату. Если формат вообще не соответствует ожидаемому, функции возвращают 0.

Функции **atof и strtod** распознают кроме соответствующих цифровых последовательностей тексты "+INF" и "-INF", которыми обозначаются плюс и минус бесконечности, а также тексты "+NAN" и "-NAN", обозначающие «не цифровая величина».

В функциях **strtod,** strtol, _strtold, **strtoul,** wcstod, wcstol, **wcstoul** параметр **endptr** может задаваться равным **NULL.** Например, оператор:

```
double y = strtod(Edit1->Text.c_str(),NULL);
```

преобразует текст, введенный пользователем в окне редактирования **Edit1,** в значение y. Если же задать параметр **endptr:**

```
char *endptr;
double y = strtod(Edit1->Text.c_str(),&endptr);
```

то величина ***endptr** будет равна тому символу, на котором остановилось преобразование строки. Этот параметр можно использовать для проверки правильности преобразуемой строки.

Если при преобразовании наступает переполнение, то функции возвращают положительные или отрицательные значения **HUGE_VAL** (для типа **double)** или **LHUGE_VAL** (для типа **long double).**

Рассмотренные функции преобразования можно использовать и для типа строк AnsiString (см. разд. 2.5.2). Но при этом эти строки надо переводить в тип char * с помощью метода c_str(), как показано в двух предыдущих примерах.

3.3.1.2 Функции взаимного преобразования чисел и строк, описанные в файле SysUtils.hpp

Функция	Синтаксис / Преобразует
CurrToStr	**System::AnsiString** CurrToStr(System::Currency Value) число Value типа Currency в строку
CurrToStrF	**System::AnsiString CurrToStrF(System::Currency** Value, **TFloatFormat** Format, int Digits) число типа Currency в строку с помощью формата типа TFloatFormat (см. разд. 3.1.2.4)
FloatToDecimal	void **FloatToDecimal(TFloatRec** &Result, const void *Value, **TFloatValue** ValueType, int Precision, int Decimals) число Value типа ValueType (СМ. TFloatValue в разд. 3.1.2.4) в структуру TFloatRec
FloatToStr	System::AnsiString FloatToStr(Extended Value) число Value в строку
FloatToStrF	System::AnsiString FloatToStrF(Extended Value, TFloatFormat Format, int Precision, int Digits) число Value в строку с помощью формата типа TFloatFormat (см. разд. 3.1.2.4)

Функция	Синтаксис / Преобразует
FloatToText	int FloatToText(char * Buffer, const void *VaIue, TFloatValue ValueType, TFloatFormat Format, int Precision, int Digits) число Value типа ValueType в строку Buffer с помощью формата типа TFloatFormat (см. разд. 3.1.2.4)
FloatTo-TextFmt	int FloatToTextFmt(char * Buffer, const void *Value, TFloatValue ValueType, char * Format) число Value типа ValueType (СМ. TFloatValue в разд. 3.1.2.4) в строку Buffer с помощью формата FormatFloat (см. разд. 3.1.2.5)
FmtStr	void FmtStr(System::AnsiString &Result, const System::AnsiString Format, const System::TVarRec * Args, const int Args_Size) аргументы из открытого массива Args размера Args_Size — 1 в строку Result по формату Format (см. разд. 3.1.2.3)
Format	System::AnsiString Format(const System::AnsiString Format, const System::TVarRec* Args, const int Args_Size) аргументы из открытого массива Args размера Args_Size - 1 в строку по формату Format (см. разд. 3.1.2.3)
FormatBuf	Cardinal FormatBuf(void *Buffer, Cardinal BufLen, const void *Format, Cardinal FmtLen, const System::TVarRec * Args, const int Args_Size) аргументы из открытого массива Args размера Args_Size - 1 в строку Buffer длины BufLen по формату Format (см. разд. 3.1.2.3) длины FmtLen
FormatCurr	System::AnsiString FormatCurr(const System::AnsiString Format, System::Currency Value) число типа Currency в строку с помощью формата функции FormatFloat (см. разд. 3.1.2.5)
FormatFloat	System::AnsiString FormatFloat(const System::AnsiString Format, Extended Value) число Value в возвращаемую строку с помощью формата типа FormatFloat (см. разд. 3.1.2.5)
GetFormat-Settings	void GetFormatSettings(void) устанавливает значения по умолчанию всех глобальных переменных, определяющих форматы дат и чисел
IntToHex	System::AnsiString IntToHex(int Value, int Digits) целое Value в строку с минимум Digits шестнадцатеричных цифр
IntToStr	System::AnsiString IntToStr(int Value) целое Value в строку

Функция	Синтаксис / Преобразует
StrFmt	char * StrFmt(char * Buffer, char * Format, const System::TVarRec * Args, const int Args_Size) аргументы из открытого массива Args размера Args_Size - 1 в строку Buffer по формату Format (см. разд. 3.1.2.3)
StrLFmt	char * StrLFmt(char * Buffer, Cardinal MaxLen, char * Format, const System::TVarRec* Args, const int Args_Size) аргументы из открытого массива Args размера Args_Size - 1 в строку Buffer размера MaxLen по формату Format (см. разд. 3.1.2.3)
StrToCurr	System::Currency StrToCurr(const System::AnsiString S) строку S в число типа Currency
StrToCurrDef	System::Currency StrToCurrDef(const AnsiString S, const System::Currency Default) строку S в число типа Currency со значением по умолчанию Default
StrToFloat	Extended StrToFloat(const System::AnsiString S) строку S в число
StrToFloatDef	Extended StrToFloatDef(const AnsiString S; const Extended Default) строку S в число со значением по умолчанию Default
StrToInt	int StrToInt(const System::AnsiString S) строку S в целое
StrToIntDef	int StrToIntDef(const System::AnsiString S, int Default) строку S в целое, при ошибке — значение Default по умолчанию
TextToFloat	bool TextToFloat(char * Buffer, void *Value, TFloatValue ValueType) строку Buffer в число Value типа ValueType (см. TFloatValue в разд. 3.1.2.4)
TryStrToInt	bool __fastcall TryStrToInt(const AnsiString S, int &Value) строку S в целое Value, возвращая false в случае неудачи

Комментарии

Многие функции взаимного преобразования чисел и строк, объявленные в файле *SysUtils.hpp*, используют для указания типа числа переменную ValueType, которая может принимать значение **fvExtended** — число с плавающей запятой типа **Extended,** или значение **fvCurrency** -- число типа **Currency.** Многие функции используют для форматирования строку типа **TFloatFormat,** подробно описанную в разд. 3.1.2.4, или формат функции **FormatFloat,** описанный в разд. 3.1.2.5, или строку форматирования функции, **Format,** описанную в разд. 3.1.2.3.

Ряд функций получает список форматируемых значений из открытого массива аргументов **Args** размера **Args_Size** - 1. В качестве Args_Size в них задается последний индекс массива Args типа **TVarRec.** В этих функциях используется строка форматирования, описанная в разд. 3.1.2.3.

При ошибках преобразования большинство рассматриваемых функций генерирует исключение **EConvertError**. Это правило не затрагивает функции с суффиксом "Def", которые в случае ошибки заносят в результат указанное в них значение по умолчанию, и функцию TryStrToInt, которая в случае ошибки возвращает **false.**

Функция **FloatToDecimal** преобразует число с плавающей запятой типа **Extended** или **Currency** в десятичное представление, которое может в дальнейшем подвергаться дополнительному форматированию. Для значения типа **Extended** параметр **Precision** указывает число значащих цифр от 1 до 18. Для значения типа **Currency** параметр **Precision** игнорируется, а точность предполагается равной 19 разрядам.

Параметр **Decimals** указывает максимально требуемое число цифр слева от десятичной точки. Таким образом, параметры **Precision** и **Decimals** совместно определяют способ округления результата. Чтобы результат всегда имел заданное количество значащих цифр независимо от значения числа, можно указать **Decimals** равным 9999.

Результат преобразования заносится в структуру типа **TFloatRec,** имеющую поля:

Exponent	Хранит количество значащих цифр до десятичной точки. Если число меньше 1, то поле Exponent содержит отрицательное число, модуль которого равен номеру первого значащего разряда после десятичной точки. Если значение равно NAN (не число), Exponent равняется -32768. Если значение INF или **–INF** (плюс или минус бесконечность), то Exponent = 32767.
Negative	При отрицательном числе — **true**, при положительном или нуле — false.
Digits	Строка с нулевым символом в конце, содержащая до 18 (для Extended) или 19 (для Currency) значащих цифр. Десятичная точка не хранится. Завершающие нули удаляются. Если число рано нулю, NAN или INF, Digits содержит только нулевой символ.

Ниже приведен пример использования функции **FloatToDecimal:**

```
struct TFloatRec Result;
Extended Value = ...;
FloatToDecimal(Result, &Value,fvExtended,18,9999);
```

При различных значениях **Value** получаются результаты:

Value	Exponent	Negative	Digits
123.4567890123456789	3	false	123456789012345681
1234567890123456789	19	false	123456789012345679
-0.001234567890123456789	-2	true	123456789012345671

3.3.2 Функции преобразования дат и времени

Функция	Синтаксис / Описание	Файл
asctime	**char *asctime(const struct tm *tblock)** **Переводит структуру типа tm в строку**	*time.h*

Функция	Синтаксис / Описание	Файл
CompareDate	Types::TValueRelationship CompareDate(const System::TDateTime A, const System::TDateTime B) Сравнивает два значения дат A и B. Возвращает -1 при A < B, 0 при A = B, +1 при A > B	*DateUtils.hpp*
CompareDateTime	Types::TValueRelationship CompareDateTime(const System::TDateTime A, const System::TDateTime B) Сравнивает два значения дат и времени A и B. Возвращает -1 при A < B, 0 при A = B, +1 при A > B	*DateUtils.hpp*
CompareTime	Types::TValueRelationship CompareTime(const System::TDateTime A, const System::TDateTime B) Сравнивает два значения времени A и B. Возвращает -1 при A < B, 0 при A = B, +1 при A > B	*DateUtils.hpp*
ctime	char *ctime(const time_t *time) Переводит время time, полученное функцией time, в строку	*time.h*
Date	System::TDateTime Date(void) Возвращает текущую дату	*SysUtils.hpp*
DateTimeToFileDate	int DateTimeToFileDate(System::TDateTime DateTime) Переводит DateTime в формат даты и времени DOS	*SysUtils.hpp*
DateTimeToStr	System::AnsiString DateTimeToStr(System::TDateTime DateTime) Преобразует DateTime в строку	*SysUtils.hpp*
DateTimeToString	void DateTimeToString(System::AnsiString &Result, const System::AnsiString Format, System::TDateTime DateTime) Преобразует DateTime в строку Result по формату Format	*SysUtils.hpp*
DateTimeToSystemTime	void DateTimeToSystemTime(System::TDateTime DateTime, _SYSTEMTIME &SystemTime) Преобразует DateTime в формат TSystemTime, используемый в API Windows	*SysUtils.hpp*
DateTimeToTimeStamp	TTimeStamp DateTimeToTimeStamp(System::TDateTime DateTime) Преобразует DateTime в TTimeStamp	*SysUtils.hpp*
DateToStr	System::AnsiString DateToStr(System::TDateTime Date) Преобразует дату Date в строку	*SysUtils.hpp*

Функция	Синтаксис / Описание	Файл
DayOf	**Word DayOf (const System::TDateTime AValue)** Извлекает из значения **AValue** день месяца	*DateUtils.hpp*
DayOfTheMonth	**Word DayOfTheMonth(const System::TDateTime AValue)** Извлекает из значения **AValue** день месяца	*DateUtils.hpp*
DayOfTheWeek	**Word DayOfTheWeek(const System::TDateTime AValue)** Извлекает из даты **AValue** день недели (от 1 до 7, 1 — понедельник)	*DateUtils.hpp*
DayOfWeek	**Word DayOfWeek(const System::TDateTime DateTime)** Извлекает из даты **Date** день недели (от 1 до 7, 1 — воскресенье)	*DateUtils.hpp*
DaysBetween	**int DaysBetween(const System::TDateTime ANow, const System::TDateTime AThen)** Возвращает число полных суток между двумя значениями даты и времени	*DateUtils.hpp*
DaySpan	**double DaySpan(const System::TDateTime ANow, const System::TDateTime AThen)** Возвращает число суток между двумя значениями даты и времени	*DateUtils.hpp*
DecodeDate	**void DecodeDate(const System::TDateTime DateTime, Word &Year, Word &Month, Word &Day)** Разбивает **DateTime** на год **Year,** месяц **Month,** день Day	*SysUtils.hpp*
DecodeDateTime	**void DecodeDateTime(const System::TDateTime DateTime, Word &Year, Word &Month, Word &Day, Word &Hour, Word &Min, Word &Sec, Word &MSec)** Разбивает **DateTime** на год **Year,** месяц **Month,** день **Day,** часы **Hour,** минуты Min, секунды Sec, миллисекунды **MSec**	*DateUtils.hpp*
DecodeTime	**void DecodeTime(System::TDateTime Time, Word &Hour, Word &Min, Word &Sec, Word &MSec)** **Разбивает Time на часы Hour,** минуты **Min,** секунды Sec, миллисекунды **MSec**	*SysUtils.hpp*

Функция	Синтаксис / Описание	Файл
EncodeDate	TDateTime EncodeDate(Word Year, Word Month, Word Day) Преобразует год Year, месяц Month и день Day в TDateTime	*SysUtils.hpp*
EncodeDateTime	TDateTime EncodeDateTime(const Word Year, const Word Month, const Word Day, const Word Hour, const Word Min, const Word Sec, const Word MSec) Преобразует год Year, месяц Month, день Day в TDateTime, часы Hour, минуты Min, секунды Sec и миллисекунды MSec в TDateTime	*DateUtils.hpp*
EncodeTime	TDateTime EncodeTime(Word Hour, Word Min, Word Sec, Word MSec) Преобразует часы Hour, минуты Min, секунды Sec и миллисекунды MSec в TDateTime	*SysUtils.hpp*
FormatDateTime	System::AnsiString FormatDateTime(const System::AnsiString Format, System::TDateTime DateTime) Преобразует DateTime в строку по формату Format	*SysUtils.hpp*
getdate	void getdate(struct date *datep) Заносит в datep текущую дату	*dos.h*
gettime	void gettime(struct time *timep) Заносит в timep текущее время	*dos.h*
gmtime	struct tm *gmtime(const time_t *timer) Переводит время timer, полученное функцией time, в структуру типа tm	*time.h*
HourOf	Word HourOf(const System::TDateTime AValue) Извлекает из значения AValue час	*DateUtils.hpp*
HourOfTheDay	Word HourOfTheDay(const System::TDateTime AValue) Извлекает из значения AValue день месяца	*DateUtils.hpp*
HoursBetween	int HoursBetween(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число полных часов между двумя значениями даты и времени	*DateUtils.hpp*
HourSpan	double HourSpan(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число суток между двумя значениями даты и времени	*DateUtils.hpp*

Функция	Синтаксис / Описание	Файл
IncMonth	System::TDateTime IncMonth(const System:: TDateTime Date, int NumberOfMonths) Возвращает дату Date, измененную на NumberOfMonths месяцев	*SysUtils.hpp*
IsLeapYear	bool IsLeapYear(Word Year) Возвращает true, если год Year високосный	*SysUtils.hpp*
IsToday	bool IsToday(const System::TDateTime AValue) Возвращает true, если AValue соответствует сегодняшней дате	*DateUtlls.hpp*
local time	struct tm *localtime(const time_t *timer) Переводит время timer, полученное функцией time, в структуру типа tm с поправкой на локальное время	*time.h*
MilliSecondOf	Word MilliSecondOf(const System::TDateTime AValue) Извлекает из значения AValue миллисекунды	*DateUtils.hpp*
MilliSecond OfTheSecond	Word MilliSecondOfTheSecond(const System::TDateTime AValue) Извлекает из значения AValue миллисекунды	*DateUtils.hpp*
MilliSecondsBetween	int MilliSecondsBetween(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число миллисекунд между двумя значениями даты и времени	*DateUtils.hpp*
MilliSecondSpan	double MilliSecondSpan(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число миллисекунд между двумя значениями даты и времени	*DateUtils.hpp*
MinuteOf	Word MinuteOf(const System::TDateTime AValue) Извлекает из значения AValue минуты	*DateUtils.hpp*
MinuteOfTheHour	Word MinuteOfTheHour(const System::TDateTime AValue) Извлекает из значения AValue минуты	*DateUtils.hpp*
MinutesBetween	int MinutesBetween(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число полных минут между двумя значениями даты и времени	*DateUtils.hpp*

Функция	Синтаксис / Описание	Файл
MinuteSpan	double MinuteSpan(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число миллисекунд между двумя значениями даты и времени	DateUtils.hpp
mktime	time_t mktime(struct tm *t) Переводит время из структуры типа tm в формат time_t	time.h
MonthOf	Word MonthOf(const System::TDateTime AValue) Извлекает из значения AValue месяц	DateUtils.hpp
MonthOfTheYear	Word MonthOfTheYear(const System::TDateTime AValue) Извлекает из значения AValue месяц	DateUtils.hpp
MonthsBetween	int MonthsBetween(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число полных месяцев между двумя значениями даты и времени	DateUtils.hpp
MonthSpan	double MonthSpan(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число месяцев между двумя значениями даты и времени	DateUtils.hpp
MSecsToTimeStamp	TTimeStamp MSecsToTimeStamp(System::Comp MSecs) Преобразует миллисекунды MSecs в TTimeStamp	SysUtils.hpp
Now	System::TDateTime Now(void) Возвращает текущую дату и время	SysUtils.hpp
SecondOf	Word SecondOf (const System::TDateTime AValue) Извлекает из значения AValue секунды	DateUtils.hpp
SecondOfTheMinute	Word SecondOfTheMinute(const System::TDateTime AValue) Извлекает из значения AValue месяц	DateUtils.hpp
SecondsBetween	int SecondsBetween(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число полных секунд между двумя значениями даты и времени	DateUtils.hpp

Функция	Синтаксис / Описание	Файл
SecondSpan	double SecondSpan(const System::TDateTime ANow, const System::TDateTime AThen) Возвращает число секунд между двумя значениями даты и времени	*DateUtils.hpp*
setdate	void setdate(struct date *datep) Задает дату datep как системную (если пользователю разрешен доступ)	*dos.h*
settime	void settime(struct time *timep) Задает время timep как системное	*dos.h*
stime	int stime(time_t *tp) Задает системную дату и время из tp	*time.h*
StrToDate	System::TDateTime StrToDate(const System::AnsiString S) Преобразует строку S в дату TDateTime	*SysUtils.hpp*
StrToDateDef	System::TDateTime StrToDateDef(const AnsiString S, const System::TDateTime Default) Преобразует строку S в дату TDateTime, задавая значение по умолчанию Default в случае ошибки	*SysUtils.hpp*
StrToDateTime	System::TDateTime StrToDateTime(const System::AnsiString S) Преобразует строку S в дату и время TDateTime	*SysUtils.hpp*
StrToDateTimeDef	System::TDateTime StrToDateTimeDef(const AnsiString S, const System::TDateTime Default) Преобразует строку S в дату и время TDateTime, задавая значение по умолчанию Default в случае ошибки	*SysUtils.hpp*
StrToTime	System::TDateTime StrToTime(const System::AnsiString S) Преобразует строку S во время TDateTime	*SysUtils.hpp*
StrToTimeDef	System::TDateTime StrToTimeDef(const AnsiString S, const System::TDateTime Default) Преобразует строку S во время TDateTime, задавая значение по умолчанию Default в случае ошибки	*SysUtils.hpp*
SystemTime ToDateTime	System::TDateTime SystemTimeToDateTime(const _SYSTEMTIME &SystemTime) Преобразует формат TSystemTime, используемый в API Windows, в TDateTime	*SysUtils.hpp*

Функция	Синтаксис / Описание	Файл
time	time_t time(time_t *timer) Возвращает текущее время и заносит его в timer (если timer не NULL)	*time.h*
Time	System::TDateTime Time(void) Возвращает текущее время	*SysUtils.hpp*
TimeStamp ToDateTime	System::TDateTime TimeStampToDateTime(const TTimeStamp &TimeStamp) Преобразует структуру типа TTimeStamp в TDateTime	*SysUtils.hpp*
TimeStampTo MSecs	System::Comp TimeStampToMSecs(const TTimeStamp &TimeStamp) Возвращает 64-разрядное значение числа миллисекунд	*SysUtils.hpp*
TimeToStr	System::AnsiString TimeToStr(System::TDateTime Time) Преобразует время в строку	*SysUtils.hpp*
Today	TDateTime__fastcall Today(void) Возвращает текущую дату	*DateUtils.hpp*
Tomorrow	TDateTime__fastcall Tomorrow(void) Возвращает дату завтрашнего дня	*DateUtils.hpp*
TryEncodeDate	bool TryEncodeDate(Word Year, Word Month, Word Day, System::TDateTime &Date) Преобразует год Year, месяц Month и день Day в Date, возвращая false в случае ошибки	*SysUtils.hpp*
TryEncodeDate Time	bool TryEncodeDateTime(const Word Year, const Word Month, const Word Day, const Word Hour, const Word Min, const Word Sec, const Word MSec, System::TDateTime &Value) Преобразует год Year, месяц Month, день Day в TDateTime, часы Hour, минуты Min, секунды Sec и миллисекунды MSec в TDateTime, возвращая false в случае ошибки	*SysUtils.hpp*
TryEncodeTime	bool TryEncodeTime(Word Hour, Word Min, Word Sec, Word MSec, System::TDateTime &Time) Преобразует часы Hour, минуты Min, секунды Sec и миллисекунды MSec в Time, возвращая false в случае ошибки	*SysUtils.hpp*
YearOf	Word YearOf (const System: :TDateTime AValue) Извлекает из значения AValue год	*DateUtils.hpp*

Функция	Синтаксис / Описание	Файл
YearsBetween	**int YearsBetween(** **const System::TDateTime ANow,** **const System::TDateTime AThen)** **Возвращает число полных лет между двумя значениями даты и времени**	*DateUtils.hpp*
YearSpan	**double YearSpan(** **const System::TDateTime ANow,** **const System::TDateTime AThen)** **Возвращает число лет между двумя значениями даты и времени**	*DateUtils.hpp*
Yesterday	**TDateTime __fastcall Yesterday(void)** **Возвращает дату вчерашнего дня**	*DateUtils.hpp*

Комментарии

Большинство функций данного раздела использует тип **TDateTime** (см. разд. 3.1.7), представляющий собой число с плавающей запятой, целая часть которого соответствует дате, дробная — времени. Некоторые функции используют также тип **TTimeStamp**. Это структура, содержащая дату и время. Имеется еще один формат представления дат и времени, принятый в DOS. Этот формат используется в таких функциях, как **FileAge, FileGetDate, FileSetDate,** в поле **Time** структуры типа **TSearchRec,** применяемой в функциях **FindFirst и FindNext.** Перевод в этот формат значения типа **TDateTime** осуществляется функцией **DateTimeToFileDate.** Наконец, имеется еще системный формат — **TSystemTime,** определенный как тип **_SYSTEMTIME.** Он может требоваться при вызове функций API Windows. Преобразование **TDateTime** в этот формат осуществляется функцией **DateTimeToSystemTime,** а обратное преобразование осуществляется функцией **SystemTimeToDateTime.**

Функции из файла dos.h используют для хранения даты структуру типа **date:**

```
struct date{
   int da_year;      // текущий год
   char da_day;      // день месяца
   char da_mon;      // номер месяца (1 — январь)
};
```

Все данные хранятся в виде целых чисел, что облегчает их дальнейшую обработку. Приведем пример использования такой структуры. Следующие операторы создают структуру D типа **date** и заносят в нее текущую дату:

```
#include <dos.h>
...
struct date D;
getdate(&D);
```

В дальнейшем можно обращаться к полям этой структуры: **D.da_year, D.da_mon, D.da_day.**

Аналогичная структура предусмотрена и для хранения времени:

```
struct time {
   unsigned char ti_min;      // минуты
   unsigned char ti_hour;     // часы

   unsigned char ti_hund;     // сотые доли секунды
   unsigned char ti_sec;      // секунды
};
```

Функция **time** возвращает текущее время в секундах, отсчитанное от 0 часов 1 января 1970 по Гринвичу. Это время может быть преобразовано в строку с нулевым символом в конце, включающую год, месяц, день и т.д., с помощью функции **ctime** с учетом поправок на локальное время. Вид строки:

```
Mon Nov 21 11:31:54 1983\n\0
```

к сожалению, с английскими сокращениями.

Время, возвращаемое функцией **time,** может также с помощью функций **gmtime** (время по Гринвичу) или **localtime** (время с локальной поправкой) преобразовываться в поля структуры типа tm:

```
struct tm {
   int tm_sec;          //  секунды
   int tm_min;          //  минуты
   int tm_hour;         // часы (0 - 23)
   int tm_mday;         // день месяца (1 — 31)
   int tm_mon;          // месяц (0 - 11)
   int tm_year;         // год (календарный минус 1900)
   int tm_wday;         // день недели (0 — 6; 0 — воскресенье
   int tm_yday;         // день года (0 -365)
   int tm_isdst;        // установлен ли 12-часовой формат
};
```

Данная структура может быть преобразована в строку функцией **asctime.** Например, операторы:

```
time_t t = time(NULL);
struct tm *tt = localtime(&t);
char s[80];
strcpy(s, asctime(tt));
```

создают структуру типа tm, заносят в нее текущее время, полученное функцией **time** и преобразованное функцией **gmtime,** после чего формируют строку s. Но учтите, что строка получится аналогичной строке, возвращаемой описанной выше функцией **ctime** — т.е. использующей английские сокращения.

3.3.3 Функции преобразования типов

Функция	Синтаксис / Преобразует	Файл
Bounds	Windows::TRect Bounds(int ALeft, int ATop, int **AWidth,** int AHeight) Координаты ALeft и ATop и размеры AWidth и AHeight в TRect	*Classes.hpp*
CurrTo FMTBCD	**bool CurrToFMTBCD(System::Currency** Curr, Bde::FMTBcd &BCD, int Precision, int Decimals) Значение **Curr** в тип Bde::FMTBcd	*DBCommon.hpp*
FMTBCD ToСигг	bool **FMTBCDToCurr(const** Bde::FMTBcd &BCD, System: **:Currency &Curr)** Значение BCD в тип Currency	*DBCommon.hpp*
Point	TPoint Point(int AX, int AY) Координаты AX и AY в TPoint	*Classes.hpp*

Функция	Синтаксис / Преобразует	Файл
Rect	**Windows::TRect** Rect(int ALeft, int ATop, int ARight, int ABottom) **Windows::TRect** Rect(constTPoint **ATopLeft,** const TPoint ABottomRight) Координаты ALeft, ATop, ARight, ABottom или точки ATopLeft, ABottomRight в TRect	*Classes.hpp*

Комментарии

Функции **CurrToFMTBCD** и **FMTBCDToCurr** осуществляют взаимное преобразование типа **Currency** и типа **Bde::FMTBcd,** используемого для хранения в полях BCD баз данных.

3.4 Строки и символы

3.4.1 Функции обработки символов

Функция	Синтаксис / Описание	Файл
_tolower	int **_tolower(int ch)** Макрос приведения латинской буквы к нижнему регистру (без проверки)	*ctype.h*
_toupper	int **_toupper(int** ch) Макрос приведения латинской буквы к верхнему регистру (без проверки)	*ctype.h*
isalnum	int **isalnum(int** c) Макрос проверки на латинскую букву или цифру	*ctype.h*
isalpha	int isalpha(int c) Макрос проверки на латинскую букву	*ctype.h*
isascii	int isascii(int c) Макрос проверки на символ из набора ASCII	*ctype.h*
iscntrl	int iscntrl(int c) Макрос проверки на управляющий символ	*ctype.h*
isdigit	int isdigit(int c) Макрос проверки на цифру	*ctype.h*
isgraph	int isgraph(int c) Макрос проверки на печатный символ (исключая пробел)	*ctype.h*
islower	int islower(int c) Макрос проверки на латинскую букву в нижнем регистре	*ctype.h*
isprint	int isprint(int c) Макрос проверки на печатный символ (включая пробел)	*ctype.h*
ispunct	int ispunct(int c) Макрос проверки на символ пунктуации (любой печатаемый, кроме латинской буквы, цифры, пробела)	*ctype.h*

Функция	Синтаксис / Описание	Файл
isspace	**int** isspace(int c) Макрос проверки на пробельный символ (пробел, табуляция, новая строка)	*ctype.h*
isupper	int **isupper(int** c) Макрос проверки на латинскую букву в верхнем регистре	*ctype.h*
iswalnum	int **iswalnum(wint_t** c) Макрос проверки на латинскую букву или цифру	*ctype.h*
iswalpha	int **iswalpha(wint_t** c) Макрос проверки на латинскую букву	*ctype.h*
iswascii	int **iswascii(wint_t** c) Макрос проверки на символ из набора ASCII	*ctype.h*
iswcntrl	int **iswcntrl(wint_t** c) Макрос проверки на управляющий символ	*ctype.h*
iswdigit	int **iswdigit(wint_t** c) Макрос проверки на цифру	*ctype.h*
iswgraph	int **iswgraph(wint_t** c) Макрос проверки на печатный символ (исключая пробел)	*ctype.h*
iswlower	int **iswlower(wint_t** c) Макрос проверки на латинскую букву в нижнем регистре	*ctype.h*
iswprint	int iswprint(wint_t c) Макрос проверки на печатный символ (включая пробел)	*ctype.h*
iswpunct	int **iswpunct(wint_t** c) Макрос проверки на символ пунктуации (любой печатаемый, кроме латинской буквы, цифры, пробела)	*ctype.h*
iswspace	int **iswspace(wint_t** c) Макрос проверки на пробельный символ (пробел, табуляция, новая строка)	*ctype.h*
iswupper	int iswupper(wint_t c) Макрос проверки на латинскую букву в верхнем регистре	*ctype.h*
iswxdigit	int **iswxdigit(wint_t** c) Макрос проверки на шестнадцатеричную цифру	*ctype.h*
isxdigit	int isxdigit(int c) Макрос проверки на шестнадцатеричную цифру	*ctype.h*
toascii	int toascii(int c) Макрос преобразования целого в код ASCII (очистка всех битов, кроме 7 младших) — в число от 0 до 127)	*ctype.h*
tolower	int tolower(int ch) Макрос приведения латинской буквы к нижнему регистру, если она в верхнем регистре	*ctype.h*

Функция	Синтаксис / Описание	Файл
toupper	int **toupper(int** ch) Макрос приведения латинской буквы к верхнему регистру, если она в нижнем регистре	*ctype.h*
towlower	int **towlower(wint_t** ch) Макрос приведения латинской буквы к нижнему регистру, если она в верхнем регистре	*ctype.h*
towupper	int **towupper(wint_t** ch) Макрос приведения латинской буквы к верхнему регистру, если она в нижнем регистре	*ctype.h*

Комментарии

Обратите внимание на то, что макросы, распознающие и преобразовывающие буквы, не работают с символами кириллицы. Для кириллицы надо использовать работающие с кириллицей функции строк (см. разд. 3.4.2.2 и 3.4.2.3): перевести символ в строку, преобразовать строку и взять ее первый символ. Например, оператор

```
Key = AnsiUpperCase(Key);
```

приведет символ **Key** типа **char** к верхнему регистру.

3.4.2 Функции обработки строк

3.4.2.1 Функции работы с областями памяти и строками

Функция	Синтаксис / Описание	Файл
_wmemcpv	void * **wmemcpy(void** *dest, const void ***src,** size t n) Копирует n байтов из src в dest; **src** и dest не должны перекрываться в памяти (см. memmove); возвращает dest	*mem.h*
_wmemset	void * _wmemset(void *s, int c, size t n) Заполняет n байтов блока s символом c; возвращает s	*mem.h*
memccpy	void *memccpy(void *dest, const void *src, int c, size t n) Копирует символы из src в dest, пока не встретится символ c или не будет скопировано n символов; возвращает dest	*mem.h*
memchr	void ***memchr(const** void *s, int c, size_t n) Возвращает указатель на первое вхождение символа c в первых n байтах s; если символ не найден, возвращает NULL	*mem.h*
memcmp	int memcmp(const void *sl, const void *s2, size_t n) Сравнивает n символов из **s1** и s2; результат < 0 при **s1** < s2, = 0 при **s1** = s2, > 0 при **s1** > s2	*mem.h*
memcpv	void *memcpy(void *dest, const void *src, size_t n) Копирует n байтов из **src** в dest; **src** и dest не должны перекрываться в памяти (см. memmove); возвращает dest	*mem.h*
memicmp	int **memicmp(const** void *sl, const void *s2, size_t n) Сравнивает, игнорируя регистр (не кириллицу), n символов из **s1** и s2; результат < 0 при **s1** < s2, = 0 при **s1** = s2, > 0 при **s1** > s2	*mem.h*

Функция	Синтаксис / Описание	Файл
memmove	void *memmove(void *dest, const void *src, size_t n) Копирует n байтов из src в dest; src и dest могут перекрываться в памяти (см. тетеру); возвращает dest	mem.h
memset	void *memset(void *s, int c, size_t n) Заполняет n байтов блока s символом c; возвращает s	mem.h
setmem	void setmem(void *dest, unsigned length, char value) Заполняет блок dest размером length байтом value	mem.h

Комментарии

Приведенные в данном разделе функции могут работать как со строками с нулевым символом в конце, так и со строками без нулевого символа, а также с блоками памяти, не являющимися строками.

3.4.2.2 Функции обработки строк с нулевым символом в конце

Функция	Синтаксис / Описание	Файл
AnsiStrComp	int AnsiStrComp(char * S1, char * S2) Сравнивает строки S1 и S2 с учетом регистра; результат < 0 при S1 < S2, = 0 при S1 = S2, > 0 при S1 > S2	SysUtils.hpp
AnsiStrIComp	int AnsiStrIComp(char * S1, char * S2) Сравнивает строки S1 и S2 без учета регистра; результат < 0 при S1 < S2, = 0 при S1 = S2, > 0 при S1 > S2	SysUtils.hpp
AnsiStrLComp	int AnsiStrLComp(char * S1, char * S2, Cardinal MaxLen) Сравнивает до MaxLen символов строк S1 и S2; результат < 0 при S1 < S2, = 0 при S1 = S2, > 0 при S1 > S2	SysUtils.hpp
AnsiStrLIComp	int AnsiStrLIComp(char * S1, char * S2, Cardinal MaxLen) Сравнивает до MaxLen символов строк S1 и S2 без учета регистра; результат < 0 при S1 < S2, = 0 при S1 = S2, > 0 при S1 > S2	SysUtils.hpp
AnsiStrLower	char * AnsiStrLower(char * Str) Возвращает строку, все символы которой приведены к нижнему регистру	SysUtils.hpp
AnsiStrPos	char * AnsiStrPos(char * Str, char * SubStr) Возвращает первое вхождение подстроки SubStr в Str или NULL	SysUtils.hpp
AnsiStrRScan	char * AnsiStrRScan(char * Str, char Chr) Возвращает указатель на последнее вхождение символа Chr в Str или NULL	SysUtils.hpp
AnsiStrScan	char * AnsiStrScan(char * Str, char Chr) Возвращает указатель на первое вхождение символа Chr в Str или NULL	SysUtils.hpp

Функция	Синтаксис / Описание	Файл
AnsiStr Upper	char * AnsiStrUpper(char * Str) Возвращает строку, все символы которой приведены к верхнему регистру	*SysUtils.hpp*
CharTo Oem	BOOL CharToOem(LPCTSTR lpszSrc, LPSTR lpszDst) Переводит строку в текст MS-DOS	*winuser.h*
CharTo OemBuff	BOOL CharToOemBuff(LPCTSTR IpszSrc, LPSTR IpszDst, DWORD cchDstLength) Переводит строку в текст MS-DOS	*winuser.h*
Compare Str	int CompareStr(const System::AnsiString S1, const System::AnsiString S2) Сравнивает строки S1и S2 с учетом регистра; результат < 0 при S1 < S2, = 0 при S1 = S2, > 0 при S1 > S2	***SysUtils.hpp***
Compare Text	int CompareText(const System::AnsiString S1, const System::AnsiString S2) Сравнивает строки S1и S2 без учета регистра; результат < 0 при S1 < S2, = 0 при S1 = S2,.> 0 при S1 > S2	***SysUtils.hpp***
LineStart	char * LineStart(char * Buffer, char * BufPos) Возвращает указатель на начало последней строки в Buffer, кончающейся в позиции BufPos	***SysUtils.hpp***
_mbscpv	unsigned char * mbscpy(unsigned char *dest, const unsigned char *src) Копирует строку src в dest	*mbstring.h*
_mbslwr	unsigned char * mbslwr(unsigned char *s) Преобразует строку к нижнему регистру	*mbstring.h*
_mbsncpy	unsigned char * mbsncpy(unsigned char *dest, const unsigned char *src, size_t maxlen) Копирует до maxlen символов строки src в dest	***mbstring.h***
_mbsupr	unsigned char * mbsupr(unsigned char *s) Преобразует строку к верхнему регистру	***mbstring.h***
OemTo Char	BOOL OemToChar(LPCTSTR IpszSrc, LPSTR IpszDst) Переводит текст MS-DOS в строку	***winuser.h***
OemTo CharBuff	BOOL OemToCharBuff(LPCTSTR IpszSrc, LPSTR IpszDst, DWORD cchDstLength) Переводит текст MS-DOS в строку	***winuser.h***
StrAlloc	char * StrAlloc(Cardinal Size) Динамически выделят блок памяти под строку длиной Size — 1 и возвращает указатель на него; блок должен освобождаться функцией StrDispose	***SysUtils.hpp***
StrBufSize	Cardinal StrBufSize(char * Str) Возвращает максимальное число символов, которые могут разместиться в созданной функцией StrAlloc строке Str	***SysUtils.hpp***

Функция	Синтаксис / Описание	Файл
strcat	char *strcat(char *dest, const char *src) Добавляет строку src в конец строки dest; возвращает указатель на результирующую строку	*string.h*
StrCat	char * StrCat(char * Dest, char * Source) Добавляет строку Source в конец строки Dest; возвращает указатель на результирующую строку	*SysUtils.hpp*
strchr	char *strchr(const char * s, int c) Возвращает указатель на первое вхождение c в s, или NULL	*string.h*
strcmp	int strcmp(const char *sl, const char *s2) Сравнивает строки s1 и s2; результат < 0 при s1 < *s2*, = 0 при s1 = s2, > 0 при s1 > s2	*string.h*
strcmpi	int strcmpi(const char *sl, const char *s2) То же, что stricmp: сравнивает строки s1 и s2 без учета регистра (не кириллицу); результат < 0 при s1 < s2, = 0 при s1 = s2, > 0 при s1 > s2	*string.h*
StrComp	int StrComp(char * Str1, char * Str2) Сравнивает строки S1и S2 с учетом регистра (для кириллицы лучше использовать AnsiStrComp); результат < 0 при S1 < S2, = 0 при S1 = S2, > 0 при S1 > S2	*SysUtils.hpp*
StrCopv	char * StrCopy(char * Dest, char * Source) Копирует Source в Dest и возвращает Dest	*SysUtils.hpp*
strcpy	char *strcpy(char *dest, const char *src) Копирует строку src в dest; возвращает dest	*string.h*
strcspn	size_t strcspn(const char *sl, const char *s2) Возвращает длину начальной части строки s1, не содержащей ни одного из символов строки s2	*string.h*
strdup	char *strdup(const char *s) Выделяет соответствующую область в памяти и копирует в нее строку s; возвращает указатель на эту область	*string.h*
StrECopy	char * StrECopy(char * Dest, char * Source) Копирует Source в Dest и возвращает указатель на конечный нулевой символ Dest	*SysUtils.hpp*
StrEnd	char * StrEnd(char * Str) Возвращает указатель на конечный нулевой символ Str	*SysUtils.hpp*
strerror	char *strerror(int errnum) Возвращает указатель на строку сообщения об ошибке с номером errnum	*string.h*

Функция	Синтаксис / Описание	Файл
stricmp	**int stricmp(const** char *sl, const char *s2) То же, что strcmpi: сравнивает строки **s1** и s2 без учета регистра (не кириллицу); результат < 0 при **s1** < s2, = 0 при **s1** = s2, > 0 при **s1** > s2	*string.h*
StrIComp	int StrIComp(char * **Str1,** char * Str2) Сравнивает строки **S1и** S2 без учета регистра (для кириллицы надо использовать **AnsiStrIComp);** результат < 0 при **S1** < S2, = 0 при **S1** = S2, > 0 при **S1** > S2	*SysUtils.hpp*
StrLCat	**char** * StrLCat(char * Dest, char * Source, Cardinal MaxLen) Копирует до MaxLen символов строки Source в конец строки Dest и возвращает Dest	*SysUtils.hpp*
StrLComp	int **StrLComp(char** * Strl, char * **Str2,** Cardinal MaxLen) Сравнивает до MaxLen символов строк **S1и S2** с учетом регистра (для кириллицы лучше использовать AnsiStrLComp); результат < 0 при **S1** < S2, = 0 при **S1** = S2, > 0 при **S1** > S2	*SysUtils.hpp*
StrLCopy	char * **StrLCopy(char** * Dest, **char** * Source, Cardinal MaxLen) Копирует до MaxLen символов Source в Dest и возвращает указатель на Dest	*SysUtils.hpp*
strlen	size_t strlen(const **char** *s) Возвращает число символов в s, не считая нулевого символа в конце	*string.h*
StrLen	Cardinal **StrLen(char** * Str) Возвращает число символов в Str, не считая нулевого символа в конце	*SysUtils.hpp*
StrLIComp	int StrLIComp(char * **Str1,** char * Str2, Cardinal MaxLen) Сравнивает до MaxLen символов строк **S1и** S2 без учета регистра (для кириллицы надо использовать AnsiStrLIComp); результат < 0 при **S1** < S2, = 0 при **S1** = S2, > 0 при **S1** > S2	*SysUtils.hpp*
StrLower	char * StrLower(char * Str) Возвращает строку, все символы которой приведены к нижнему регистру (для кириллицы надо использовать **AnsiStrLower);**	*SysUtils.hpp*
strlwr	char *strlwr(char *s) Преобразует строку s в нижний регистр (только латинские буквы)	*string.h*
StrMove	char * StrMove(char * Dest, char * Source, Cardinal Count) Копирует Count символов из Source в Dest и возвращает Dest; Source и Dest могут перекрываться в памяти	*SysUtils.hpp*

Функция	Синтаксис / Описание	Файл
strncat	**char *strncat(char** *dest, const **char** *src, size_t maxlen) Копирует до maxlen символов строки src в конец строки dest и добавляет нулевой символ; возвращает dest	*string.h*
strncmp	**int strncmp(const** char *sl, const **char** *s2, size_t maxlen) Сравнивает до maxlen символов строк **s1** и s2; результат < 0 при **s1** < s2, = 0 при **s1** = s2, > 0 при **s1** > s2	*string.h*
strncmpi	int strncmpi(const **char** *sl, const **char** *s2, size_t n) То же, что strnicmp: сравнивает до maxlen символов строк **s1** и s2 без учета регистра (не кириллицу); результат < 0 при **s1** < s2, = 0 при **s1** = s2, > 0 при **s1** > s2	*string.h*
strncpy	**char** *strncpy(char *dest, const char *src, size_t maxlen) Копирует до maxlen символов из **src** в dest; возвращает dest	*stdio.h*
StrNew	char * **StrNew(char * Str)** Динамически размещает в памяти копию **Str** и возвращает указатель на нее	*SysUtils.hpp*
strnicmp	int **strnicmp(const char** *sl, const char ***s2,** size_t maxlen) То же, что strncmpi: сравнивает до maxlen символов строк **s1** и s2 без учета регистра (не кириллицу); результат < 0 при **s1** < s2, = 0 при **s1** = s2, > 0 при **s1** > s2	*string.h*
strnset	char ***strnset(char *s,** int ch, size_t n) Копирует символ ch в первые n символов s	*string.h*
strpbrk	char *strpbrk(const char *sl, const char *s2) Возвращает первое вхождение в **s1** любого из символов строки s2 или NULL	*string.h*
StrPCopy	char * StrPCopy(char * Dest, const **System::AnsiString** Source) Копирует Source в Dest и возвращает Dest	*SysUtils.hpp*
StrPLCopy	char * StrPLCopy(char * Dest, const System::AnsiString Source, Cardinal MaxLen) Копирует до MaxLen символов из Source в Dest и возвращает Dest	*SysUtils.hpp*
StrPos	char * **StrPos(char * Str1,** char * Str2) Возвращает первое вхождение подстроки Str2 в **Str1** или NULL	*SysUtils.hpp*
strrchr	char *strrchr(const char *s, int c) Возвращает последнее вхождение символа **с** в **s** или NULL	*string.h*

Функция	Синтаксис / Описание	Файл
strrev	char ***strrev(char** *s) Инвертирует (переворачивает) строку s кроме нулевого символа	*string.h*
StrRScan	char * **StrRScan(char * Str,** char **Chr)** Возвращает последнее вхождение символа Chr в Str или NULL	*SysUtils.hpp*
StrScan	char * StrScan(char * Str, char Chr) Возвращает первое вхождение символа Chr в Str или NULL	*SysUtils.hpp*
strset	char *strset(char *s, int ch); Заполняет всю строку s до нулевого символа символом ch	*string.h*
strspn	size_t strspn(const char *sl, const char *s2) Возвращает число первых символов строки **s1**, входящих в множество символов строки s2 (последовательность символов безразлична)	*string.h*
strstr	char *strstr(const **char** *sl, const char *s2) Возвращает первое вхождение подстроки s2 в строку **s1** или NULL	*string.h*
strtok	char *strtok(char *sl, const char *s2) Ищет первое вхождение разделителей из строки s2 в строке **s1** и усекает строку **s1**; возможны повторные вызовы	*string.h*
StrUpper	char * **StrUpper(char * Str)** Возвращает строку, все символы которой приведены к верхнему регистру (для кириллицы надо использовать **AnsiStrUpper);**	*SysUtils.hpp*
strupr	char *strupr(char *s) Преобразует строку s в верхний регистр (только латинские буквы)	*string.h*
_wcslwr	wchar_t * _wcslwr(wchar_t *s) Преобразует строку s в нижний регистр	*string.h*
_wcsupr	wchar_t * **wcsupr(wchar_t** *s) Преобразует строку s в верхний регистр	*string.h*

Комментарии

Функции файла *string.h,* распознающие регистр символов **(strcmpi, stricmp, strlwr, strncmpi, strnicmp, strupr),** не позволяют оперировать *с* символами кириллицы, записанными в разных регистрах. Для подобной работы с русскими текстами надо использовать аналогичные функции файла *SysUtils.hpp.* Функции этого файла могут работать с текстами на русском языке и с многобайтными символами.

Операции, выполняемые большинством функций, вероятно, понятны из пояснений в таблице. Поэтому остановимся только на некоторых из них.

Функция **strcat** прибавляет к тексту строки, указанной ее первым параметром, текст строки, указанной вторым параметром. Она возвращает указатель на строку, заданную ее первым параметром и содержащую суммарный текст обеих

строк. Это позволяет делать вложенные вызовы **strcat,** если надо склеить несколько текстов. Функция **strcpy** копирует строку, являющуюся ее вторым параметром, в строку, являющуюся первым параметром и возвращает указатель на результат копирования. Функция **strstr** позволяет искать в строке некоторую заданную последовательность символов. Многочисленные примеры применения функций **strcat, strcpy, strstr и strlen** вы можете посмотреть в гл. 2, в разд. 2.5.1.

Теперь рассмотрим функцию **strtok,** которая работает следующим образом. При своем первом вызове для данной строки **s1** функция ищет первое появление в строке одного из символов, содержащихся в строке s2. Если такой символ найден, то он заменяется на нулевой символ, т.е. строка **s1** усекается на этом символе. Функция возвращает указатель на первый символ усеченной строки **s1.** Далее можно повторно вызывать функцию **strtok,** задавая ей в качестве первого параметра **NULL.** Функция продолжит обработку той же строки **s1** (строку s2 при этом можно сменить), найдет вхождение следующего символа из s2, опять заменит его нулевым символом и вернет указатель на начало нового просмотренного фрагмента строки. Таким образом, получается чтение строки по фрагментам. Приведем пример. Операторы

```
char s[80], *p;
...
p = strtok(s, " ,.");
if (p) Memo1->Lines->Add(p);
while(p)
{
 p = strtok(NULL, " ,.;");
 if (p) Memo1->Lines->Add(p);
}
```

осуществляют поиск в строке s символов — разделителей: пробела, запятой, точки. Обработанные фрагменты строки заносятся в строки окна **Memo1.** Например, если в s занесен текст "Это текст строки, которая анализируется.", то приведенный код выдаст в окно **Memo1** строки:

```
Это
текст
строки
которая
анализируется
```

Если же мы уберем из второго параметра функции **strtok** символ пробела, то результатом будет:

```
Это текст строки
 которая анализируется
```

Функция **LineStart** производит поиск в буфере **Buffer** назад от позиции **BufPos** символа конца строки "\n". Если символ найден, то функция возвращает указатель на него, выделяя таким образом последнюю строку. Обратившись к функции повторно и задав в качестве **BufPos** значение, на 1 меньшее возвращенного, можно найти предпоследнюю строку и т.п. Если символ "\n" не найден, то функция возвращает указатель на начало **Buffer.** Например, код "

```
char *Buf = "Это первая строка\пЭто вторая\nЭто третья",
     *P = StrEnd(Buf);
...
do
{
 P=LineStart(Buf, P-1);
 ...
} while(P != Buf);
```

переберет по очереди, начиная с конца, все строки буфера Buf.

3.4.2.3 функции обработки строк типа AnsiString

Функция	Синтаксис / Описание	Файл
AdjustLine Breaks	**System::AnsiString AdjustLineBreaks(const System::AnsiString S)** Заменяет в S символы конца строки на **CR/LF** — стандартные для Unix	*SysUtils.hpp*
AnsiCompare Str	**int AnsiCompareStr(const System::AnsiString S1, const System::AnsiString S2)** Сравнивает строки **S1** и S2 с учетом регистра; результат < 0 при **S1** < S2, = 0 при **S1** = S2, > О при **S1** > S2	*SysUtils.hpp*
AnsiCompare Text	**int AnsiCompareText(const System::AnsiString S1, const System::AnsiString S2)** Сравнивает строки **S1** и S2 без учета регистра; результат < 0 при **S1** < S2, = 0 при **S1** = S2, > О при **S1** > S2	*SysUtils.hpp*
AnsiExtract QuotedStr	**System::AnsiString AnsiExtractQuotedStr(char * &Src, char Quote)** Возвращает строку Src с удаленными из ее начала и конца символами кавычек, заданными как **Quote,** и с заменой внутри двойных кавычек на одинарные	*SysUtils.hpp*
AnsiLowerCase	**System::AnsiStringAnsiLowerCase(const System::AnsiString S)** Возвращает строку S, приведенную к нижнему регистру (работает с кириллицей)	*SysUtils.hpp*
AnsiPos	**int AnsiPos(const System::AnsiString Substr, const System::AnsiString S)** Возвращает позицию начала подстроки **Substr в S или О**	*SysUtils.hpp*
AnsiQuotedStr	**System::AnsiString AnsiQuotedStr(const System::AnsiString S, char Quote)** Возвращает строку S со вставленными в ее начало и конец символами кавычек, заданными как **Quote,** и с заменой внутри строки одинарных кавычек на двойные	*SysUtils.hpp*
AnsiUpperCase	**System::AnsiString AnsiUpperCase(const System::AnsiString S)** Возвращает строку S, приведенную к верхнему регистру (работает с кириллицей)	*SysUtils.hpp*
IsDelimiter	**bool IsDelimiter(const System::AnsiString Delimiters, const System::AnsiString S, int Index)** Определяет, является ли символ с индексом **Index** в строке S одним из разделителей, указанных в строке **Delimiters**	*SysUtils.hpp*

Функция	Синтаксис / Описание	Файл
IsPathDelimiter	**bool IsPathDelimiter(const System::AnsiString S, int Index);** Определяет, является ли символ с индексом **Index** в строке S обратным слэшем '\', используемым для задания путей к файлам	*SysUtils.hpp*
LastDelimiter	**int LastDelimiter(** **const System::AnsiString Delimiters,** **const System::AnsiString S)** Возвращает индекс последнего вхождения в строку S одного из разделителей, указанных в строке **Delimiters**	*SysUtils.hpp*
LowerCase	**System::AnsiString LowerCase(** **const System::AnsiString S)** Возвращает строку S, приведенную к нижнему регистру (для кириллицы используйте **AnsiLowerCase**)	*SysUtils.hpp*
QuotedStr	**System::AnsiString QuotedStr(** **const System::AnsiString S)** Возвращает строку S со вставленными в ее начало и конец символами одинарных кавычек и с заменой внутри строки одинарных кавычек на двойные (для многобайтных символов используйте **AnsiQuotedStr**)	*SysUtils.hpp*
StringReplace	**System::AnsiString StringReplace(** **const System::AnsiString S,** **const System::AnsiString OldPattern,** **const System::AnsiString NewPattern,** **TReplaceFlags Flags)** Возвращает строку S с заменой подстроки **OldPattern** на **NewPattern**; **Flags** управляет заменами подстрок	*SysUtils.hpp*
Trim	**System::AnsiString Trim(** **const System::AnsiString S)** Возвращает строку S с удаленными начальными и конечными пробельными и управляющими символами	*SysUtils.hpp*
TrimLeft	**System::AnsiString TrimLeft(** **const System::AnsiString S)** Возвращает строку S с удаленными начальными пробельными и управляющими символами	*SysUtils.hpp*
TrimRight	**System::AnsiString TrimRight(** **const System::AnsiString S)** Возвращает строку S с удаленными конечными пробельными и управляющими символами	*SysUtils.hpp*

Функция	Синтаксис / Описание	Файл
UpperCase	System::AnsiString UpperCase(const System::AnsiString S) Возвращает строку S, приведенную к верхнему регистру (для кириллицы используйте AnsiUpperCase)	*SysUtils.hpp*
WrapText	System::AnsiString WrapText(const System::AnsiString Line, const System::AnsiString BreakStr, const TSysCharSet &BreakChars, int MaxCol) Возвращает текст **Line,** разбитый на строки длиной до **MaxCol** вставкой символов **BreakStr** и заменой на них символов множества **BreakChars**	*SysUtils.hpp*

Комментарии

Помимо функций, содержащихся в данной таблице, посмотрите в разд. 3.1.6 описание класса AnsiString. В нем вы найдете много удобных методов работы со строками типа AnsiString.

Все функции, оперирующие со строками типа AnsiString, учитывают локализацию и поэтому могут с равным успехом работать как для латинских букв, так и для кириллицы. В этом их большое преимущество перед многими функциями, работающими со строками типа **char *.**

Несколько замечаний о приведенных в таблице функциях. В функциях **IsDelimiter и IsPathDelimiter** индексы отсчитываются от 1 (вопреки утверждениям встроенной справки C++Builder). Соответственно 1 — это первый символ строки, 2 — второй и т.д.

Функция **IsDelimiter** удобна для просмотра всех символов строки и замены каких-то одних символов на другие. Например, код

```
AnsiString S, Delimiters;
Delimiters = "'";
S = ...;
for(int i = 1; i <= StrLen(S.c_str()); i++)
  if(IsDelimiter(Delimiters,S,i))
    S[i] = '"';
```

заменит в строке S все символы одинарных кавычек на двойные кавычки.

В этой функции в строке **Delimiters** не обязательно должны быть именно разделители. В нее могут быть занесены любые символы. Например, если приведенный код изменить следующим образом:

```
AnsiString S, Delimiters;
Delimiters = "123456789";
S = ...;
for(int i = 1; i <= StrLen(S.c_str()); i++)
  if(IsDelimiter(Delimiters,S,i))
    S[i] -= 1;
```

то все символы цифр в строке, кроме 0, будут уменьшены на 1.

Функция **StringReplace** возвращает строку S с заменой подстроки **OldPattern** на NewPattern. Если параметр **Flags** не включает флаг rfReplaceAll, то функция заменят только первое вхождение подстроки **OldPattern.** Если параметр **Flags** включает флаг rfIgnoreCase, то операции выполняются без учета регистра. Например, оператор

```
S1 = StringReplace(S, OldPattern, NewPattern,
                   TReplaceFlags()<<rfReplaceAll);
```

поместит в строку S1 текст строки S с заменой в ней всех подстрок **OldPattern** на **NewPattern.**

Функция **WrapText** разбивает заданный текст **Line** на строки. В качестве символов конца строки используются символы, заданные параметром **BreakStr.** Параметр **MaxCol** задает максимальное количество символов в строке. Разбиение на строки производится вставкой **BreakStr** после одного из символов, имеющихся в множестве **BreakChars.** Вставка производится после того из символов в текущей строке, который обеспечивает ее максимальную длину в пределах **MaxCol.** Если ни одного символа из **BreakChars** не встретилось, длина строки может превысить **MaxCol.** Например, операторы

```
TSysCharSet bchars;
bchars << ' ' << '.' << ',' << ';' << '+' << '-';
AnsiString S, S1;
S = ...;
S1 = WrapText(S, "\n\r", bchars, 10);
```

обеспечивают запись в **S1** текста S, разбитого на строки длиной до 10 символов, причем разбиение проводится после пробелов и знаков пунктуации. Если в S записать текст: "Этот тест показывает разбиение на строки, в частности — на символах + и –.", то результат будет следующим:

```
"Этот тест "
"показывает "
"разбиение "
"на строки,"
" в "
"частности "
"- на "
"символах +"
" и - ."
```

Можно заметить, что вторая строка содержит 11 символов, включая пробел, т.е. ее размер больше заданного.

Конечно, в этом примере указана очень маленькая длина строки и поэтому разбиение выглядит некрасиво. При нормальной для печати длине строк разбиение получается лучше.

3.5 Потоки и файлы

3.5.1 Атрибуты и флаги файлов, стандартные файлы

Файлы могут иметь следующие атрибуты, определенные в **dos.h:**

FA_RDONLY	только для чтения
FA_HIDDEN	невидимый
FA_SYSTEM	системный
FA_LABEL	метка тома
FA_DIREC	каталог
FA_ARCH	архивный

Еще один альтернативный набор констант атрибутов приведен в разд. 3.5.6. Атрибуты объединяются в одно слово операцией ИЛИ (|).

При открытии файла доступ к нему определяется следующими флагами (определены в файле *fcntl.h*):

*O*_RDONLY	файл отрыт только для чтения
O_WRONLY	файл отрыт только для записи
O_RDWR	файл отрыт для чтения и записи
0_CREAT	создание нового файла
O_TRUNC	если файл существует, он урезается до 0
O_BINARY	двоичный файл
O_TEXT	текстовый файл
O_ NOINHERIT	файл не передается в дочерний процесс
O_NDELAY	не используется, введен для совместимости с UNIX
O_APPEND	файл отрыт для добавления в конец, при каждой операции вывода указатель файла автоматически устанавливается на конец
0_CREA	если файл существует, то этот флаг не действует, если файл создается, то его флаги доступа задаются специальным параметром mode, принимающим значения, указанные в приведенной ниже таблице
O_EXCL	используется только вместе с O_CREA и означает, что, если файл уже существует, возвращается ошибка

Файлы могут открываться в следующих режимах mode (определены в файле sys\stat.h):

S_IWRITE	разрешение записи
S_IREAD	разрешение чтения
S_IREAD S_IWRITE	разрешение записи и чтения

Файлы могут иметь следующие флаги совместного доступа нескольких приложений (определены в файле *share.h*):

SH_COMPAT	Установка режима совместного доступа. Объединяется с другими флагами (например, SH_COMPAT | SH_DENWR). Происходит ошибка, если файл уже открыт с другим режимом доступа
SH_DENWR	Запрещает запись, разрешает повторное открытие файла только для чтения
SH_DENYNO	Разрешает доступ для чтения и записи (оставлен наряду с SH_DENYNONE для обратной совместимости)
SH_DENYNONE	Разрешает доступ для чтения и записи. Разрешает повторное открытие файла, но только с тем же SH_COMPAT
SH_DENYRD	Запрещает чтение, разрешает повторное открытие файла только для записи
SH_DENYRW	Доступ к файлу обеспечивает только текущий дескриптор

Флаги могут соединяться в одно слово операцией ИЛИ (|). Из флагов **SH_DENYRD, SH_DENYNO** может быть задан только один.

Имеется и другой набор констант режимов, в которых могут быть открыты файлы и которые определяют доступ к файлам других приложений (файл **SysUtils.hpp):**

Имя константы	Значение	Режим
fmOpenRead	$0000	открыть только для чтения
fmOpenWrite	$0001	открыть только для записи
fmOpenReadWrite	$0002	открыть для чтения и записи
fmShareCompat	$0000	открыть совместимым с FCB
fmShareExclusive	$0010	запрет другим приложениям читать и записывать в файл
fmShareDenyWrite	$0020	запрет другим приложениям записывать в файл
fmShareDenyRead	$0030	запрет другим приложениям читать из файла
fmShareDenyNone	$0040	полный доступ к файлу других приложений

В языках С и С++ файл рассматривается как поток (stream), представляющий собой последовательность считываемых или записываемых байтов.

В С++Builder могут использоваться два подхода к работе с файлами. Первый заключается в том, что информация о потоке (файле) заносится в структуру типа **FILE,** определенную в файле **stdio.h,** и файл оказывается связанным с этой структурой. Второй подход связывает файл с дескриптором (handle) — целым значением, характеризующим размещение информации о файле во внутренних таблицах системы.

В начале работы любой программы автоматически открывается три потока со своими дескрипторами:

поток	дескриптор	
stdin	0	стандартный входной поток — обычно клавиатура
stdout	1	стандартный выходной поток — обычно экран
stderr	2	стандартный поток сообщений об ошибках

В чистом виде эти потоки используются только в консольных приложениях. Но с помощью некоторых функций, описанных в последующих разделах, они могут быть перенаправлены в файлы и использоваться в этом случае в приложениях Windows.

3.5.2 Управление потоками и файлами, описываемыми структурами FILE

Управление файлами при подходе, описываемом структурами **FILE,** осуществляется следующими функциями.

Функция	Синтаксис / Описание	Файл
_fdopen	FILE *_fdopen(int handle, char * mode) Связывает файл с дескриптором handle, открываемый в режиме mode, с потоком и возвращает указатель на связываемую с потоком структуру типа FILE или NULL	*stdio.h*
_fileno	int _fileno(FILE *stream) Возвращает дескриптор потока stream	*stdio.h*
_flushall	int _flushall(void) Очищает буферы всех входных и выходных потоков, записывая в выходные потоки содержимое их буферов; возвращает число открытых и закрытых потоков	*stdio.h*
_fsopen	FILE * fsopen(const char *filename, const char *mode, int shflag) Открывает файл filename совместного доступа, определяемого параметрами shflag и mode; возвращает указатель на связываемую с ним структуру типа FILE или NULL	*stdio.h, share.h*
fclose	int fclose(FILE *stream) Закрывает поток stream	*stdio.h*
fflush	int fflush(FILE *stream) Очищает буфер выходного потока stream, сбрасывая его содержимое в поток; возвращает 0 при успешном завершении и EOF при ошибке	*stdio.h*
fopen	FILE *fopen(const char *filename, const char *mode) Открывает файл с именем filename в режиме mode и возвращает указатель на связываемую с ним структуру типа FILE или NULL	*stdio.h*
freopen	FILE *freopen(const char *filename, const char *mode, FILE *stream) Связывает с открытым потоком stream файл с именем filename в режиме mode и возвращает указатель на связываемую с ним структуру типа FILE или NULL	*stdio.h*
setbuf	void setbuf(FILE *stream, char *buf) Задает буфер buf для потока stream вместо буфера по умолчанию	*stdio.h*
setvbuf	int setvbuf (FILE *stream, char *buf, int type, size_t size) Задает буфер buf размера size для потока stream вместо буфера по умолчанию	*stdio.h*
tmpfile	FILE *tmpfile(void) Открывает временный двоичный файл для записи и возвращает указатель на связываемую с ним структуру типа FILE или NULL	*stdio.h*

Комментарии

Открывается файл функцией **fopen,** в которую передается как параметр строка с именем файла **filename.** Аргумент **mode** указывает на строку, которая определяет режим открытия. Она может содержать спецификаторы:

r	открыть файл только для чтения
r+	открыть существующий файл для обновления — чтения и записи
a	открыть или создать файл для записи данных в конец файла
a+	открыть или создать файл для чтения или записи в конец файла
w	создать файл для записи; если такой файл уже существует, он будет перезаписан
w+	создать файл для обновления — чтения и записи; если такой файл уже существует, он будет перезаписан

К указанным спецификаторам в конце или перед символом "+" может добавляться символ "t" — текстовый файл, или "b" — бинарный, двоичный файл. Например, "rt", "rb", "r+t", "r+b" и т.д. Если ни символ "t", ни символ "b" не указаны, то тип открываемого файла определяется значением глобальной переменной **_fmode,** определенной в файле *fcntl.h.* Она может принимать значения O_TEXT – текстовый файл (по умолчанию) или **O_BINARY** — двоичный файл.

Если открытие файла прошло успешно, функция **fopen** возвращает указатель на связываемую с потоком структуру типа **FILE.** Если произошла ошибка, то возвращается **NULL.** Типичная процедура открытия файла имеет вид:

```
•FILE *F;
if ( (F = fopen ("Test.txt", "rt")) == NULL)
{
 ShowMessage("Файл не удается открыть");
 return;
}
```

После того, как файл открыт, с ним связывается указатель, определяющий текущую позицию чтения и записи. При каждой операции чтения и записи этот указатель автоматически смещается на величину прочитанного или записанного поля. Кроме того, указатель может смещаться программно с помощью функций **fseek и rewind,** описанных в разд. 3.5.4. Однако если файл открыт для обновления — чтения и записи, то при работе с ним надо учитывать следующее:

- вывод (запись) не может следовать сразу за вводом (чтением) без предварительной установки указателя явным образом с помощью функций **fseek** или **rewind**
- ввод не может следовать сразу за выводом без предварительной установки указателя явным образом с помощью функций **fseek** или **rewind** — в противном случае ввод может неожиданно выдать признак конца файла

Функция **_fsopen** открывает файл **filename** совместного доступа для нескольких процессов, определяемого параметрами **shflag и mode.** Флаги совместного доступа, из которых может формироваться **shflag,** см. в разд. 3.5.1. Параметр **mode** аналогичен такому же параметру функции **fopen.** При работе с этой функцией в DOS предварительно должна быть загружена программа **SHARE.EXE.**

Функция **freopen** связывает с открытым потоком **stream** файл с именем **filename** в режиме **mode.** Отличие от функции **fopen** заключается только в том, что поток **stream** уже был ранее открыт. Это позволяет, в частности, переназначить стандартный поток. Например, оператор

```
FILE *F = freopen("output.txt", "wt", stdout);
```

перенаправляет стандартный выходной поток **stdout** в текстовый файл "output.txt". Все последующие выводы в поток **stdout** будут в действительности направляться в этот файл.

Функция **_fdopen** осуществляет связь между файлами, открытыми с дескрипторами (об этом подходе см. в разд. 3.5.3), и потоками типа **FILE.** Например, оператор

```
FILE * stream = fdopen(handle, "w");
```

связывает ранее открытый файл с дескриптором **handle** со структурой потока stream.

Обратное преобразование - - получение дескриптора файла, связанного со структурой **FILE,** осуществляет функция **_fileno.** В приведенном ниже примере создается файл "Test.txt" для записи и определяется его дескриптор.

```
FILE *stream;
int handle;
stream = fopen("Test.txt", "w"); // создание файла
handle = _fileno(stream);        // определение дескриптора
...
fclose(stream);                  // закрытие файла
```

Открываемые рассмотренными функциями потоки буферизуются, т.е. обмен информацией происходит не непосредственно с файлами, а с промежуточными буферами, расположенными в оперативной памяти. Информация переписывается из буфера в файл только при переполнении буфера, или при закрытии файла, или функциями **fflush и _flushall.** Первая из них действует на буфер указанного выходного потока, вторая — на буферы всех входных и выходных потоков. Для входного потока функция очищает буфер, а для выходного — немедленно сбрасывает все содержимое в поток, после чего буфер очищается. Потоки остаются открытыми и буферы готовы к приему новой информации.

Программа автоматически осуществляет буферизацию всех потоков. Однако этим процессом можно управлять функциями **setbuf и setvbuf.** Если в функции **setbuf** параметр **buf** задать равным **NULL,** то буферизация потока производиться не будет. Это может замедлить работу с потоком, но зато обеспечит немедленную передачу информации без ожидания того, чтобы буфер переполнился. Если же **buf** указывает на массив символов, то именно этот массив будет использоваться для полной буферизации потока **stream** вместо буфера по умолчанию. Размер буфера может достигать значения **BUFSIZ,** определенного в файле **stdio.h.** Например:

```
char outbuf[BUFSIZ];
setbuf(F, outbuf);
```

Функция **setbuf** должна вызываться сразу поле открытия потока или сразу после вызова функции **fseek** (см. разд. 3.5.4), устанавливающей позицию указателя потока. В противном случае вызов **setbuf** может приводить к непредсказуемым результатам.

Функция **setvbuf** предоставляет более богатые возможности по управлению процессом буферизации. В этой функции задание параметра **buf = NULL** приводит к выделению в динамически распределяемой памяти с помощью функции **malloc** места для буфера размером **size.** Этот буфер автоматически освобождает память при закрытии соответствующего потока. Размер открываемого буфера ограничен сверху константой UINT_MAX, определенной в файле *limits.h.*

Параметр **type** может принимать следующие значения:

_IOFBF	Полная буферизация файла. Когда буфер ввода пуст, очередная операция ввода пытается заполнить весь буфер. При выводе выдача содержимого в файл производится после того, как буфер заполнится до отказа.

_IOLBF	Буферизация строк. Когда буфер ввода пуст, очередная операция ввода пытается, как и в предыдущем случае, заполнить весь буфер. Однако при выводе выдача содержимого в файл производится после того, как в потоке появляется символ новой строки.
_IONBF	Небуферизованный ввод/вывод. При этом параметры **buf** и size игнорируются.

При успешном завершении функция **setvbuf** возвращает 0.

Файлы, открытые рассмотренными ранее функциями **fopen и freopen,** должны закрываться функцией **fclose.** При этом автоматически открытые буферы потоков освобождают память. Но если буферы назначались явным образом функциями **setbuf и setvbuf** с параметром buf отличным от **NULL,** то они автоматически не освобождают память.

Функция **tmpfile** открывает временный двоичный файл для записи в режиме (w+b) и возвращает указатель на связываемую с ним структуру типа **FILE.** При неудаче возвращается **NULL.** Если после создания временного файла программа не изменяет текущий каталог, то при завершении программы временный файл автоматически удаляется с диска.

Подробное рассмотрение работы с файлами, описываемыми структурами **FILE,** см. в гл. 2 в разд. 2.10.2.

3.5.3 Управление потоками и файлами, связанными с дескрипторами

Управление файлами осуществляется следующими функциями.

Функция	**Синтаксис** / Описание	Файл
_creat	int _creat(const char *path, **int** mode) Создает новый или переписывает существующий файл в режиме mode; возвращает дескриптор или −1	*io.h, sys\stat.h*
_rtl_close	int _rtl_close(int handle) Закрывает поток с дескриптором handle	*io.h*
_rtl_creat	int **_rtl_creat(const** char *path, int attrib) Создает новый или переписывает существующий файл path с атрибутами attrib; возвращает дескриптор или -1	*io.h, dos.h*
_rtl_open	int _rtl_open(const char ***filename,** int oflags) Открывает существующий файл filename для чтения или записи с атрибутами oflags; возвращает дескриптор или -1	*io.h, dos.h*
_sopen	int _sopen(char *path, int access, int **shflag** [, int **mode]**) Открывает файл path совместного доступа, определяемого параметрами access, shflag, mode; возвращает дескриптор или -1	*fcntl.h, sys\stat.h, share.h, io.h, stdio.h*
close	int close(int handle) Закрывает поток с дескриптором handle	*io.h*
creatnew	int **creatnew(const** char *path, int attrib) Аналогична _rtl_creat, но выдает ошибку, если файл существует	*io.h, dos.h*

Функция	Синтаксис / Описание	Файл
creattemp	int **creattemp(char** *path, int **attrib)** Создает временный файл с уникальным именем и атрибутами attrib в каталоге path	*io.h, dos.h*
dup	int dup(int handle) Создает и возвращает дубликат дескриптора handle	*io.h*
dup2	int dup2(int oldhandle, int newhandle) Создает и возвращает дубликат newhandle дескриптора oldhandle	*io.h*
FileClose	void FileClose(int Handle) Закрывает файл с дескриптором Handle	*SysUtils.hpp*
FileCreate	int FileCreate(const **System::AnsiString** FileName) Создает файл FileName и возвращает его дескриптор в случае успеха или –1	*SysUtils.hpp*
FileOpen	int FileOpen(const **System::AnsiString** FileName, int Mode) Открывает файл FileName в режиме Mode и возвращает его дескриптор или -1	*SysUtils.hpp*
lock	int **lock(int** handle, long offset, long length) Блокирует в файле handle length байтов, начиная с позиции offset от чтения или записи другими процессами	*io.h*
locking	int locking(int handle, int **cmd,** long length) Блокирует или разблокирует в файле handle length байтов, начиная с текущей позиции для доступа других процессов	*io.h, sys\locking.h*
setmode	int **setmode(int** handle, int amode) С помощью параметра amode задает и возвращает тип открытого файла с дескриптором handle: O_BINARY — двоичный, **O_TEXT** — текстовый	*io.h*
umask	unsigned **umask(unsigned** mode) Задает маску режима чтения/записи mode, принимаемую по умолчанию функциями open и **creat:** S_IWRITE, S_IREAD или **S_IREAD|S_IWRITE;** возвращает предыдущую маску	*io.h*
unlock	int **unlock(int** handle, long offset, long length) Разблокирует в файле handle length байтов, начиная с позиции offset, для чтения или записи другими процессами	*io.h*

Комментарии

Функция **_creat** создает новый или переписывает существующий файл в режиме **mode.** Параметр **path** указывает имя файла или имя с путем к нему. Вид файла — текстовый или двоичный, задается глобальной переменной **_fmode** – **O_TEXT** или **O_BINARY.** Если файл существует и для него установлен атрибут записи, то длина файла усекается до 0. Если же файл существует и имеет атрибут

только для чтения, то функция **_creat** выдает ошибку, а файл сохраняется неизменным.

Режим, в котором создается файл, определяется параметром **mode,** который может принимать значения, определенные в файле *sys\stat.h* и указанные в разд. 3.5.1.

При успешном завершении возвращается дескриптор созданного файла. При ошибке возвращается -1, а значение **errno** (см. разд. 3.1.4.1) может иметь значения **EACCES, ENOENT, EMFILE.**

В настоящее время функция **_creat** считается устаревшей и вместо нее рекомендуется использовать **_rtl_creat.** Она действует подобно функции **_creat,** но всегда создает двоичный файл и позволяет своим параметром attrib установить операцией ИЛИ (|) атрибуты: **FA_RDONLY** — только для чтения, **FA_HIDDEN** — невидимый, **FA_SYSTEM** — системный (подробнее об атрибутах см. в разд. 3.5.1).

Функция **creatnew** аналогична функции **_rtl_creat** во всем, кроме того, что возвращает -1 в случае, если файл с данным именем уже существует.

Функция **_rtl_open** открывает существующий файл **filename** для чтения или записи с атрибутами **oflags.** Таблица возможных атрибутов приведена в разд. 3.5.1. При успешном завершении возвращается дескриптор открытого файла и его указатель устанавливается на начало файла. При ошибке возвращается -1, а значение **errno** (см. разд. 3.1.4.1) может иметь значения **EINVACC, EACCES,** ENOENT, **EMFILE.**

Функция **_sopen** открывает файл **path** совместного доступа, определяемого параметрами **access, shflag, mode.** Параметр **access** задает совокупность O_... флагов доступа, перечисленных в разд. 3.5.1. Если среди этих флагов задан **O_CREA,** то режим открытия файла определяется параметром **mode** (см. разд. 3.5.1). Параметр **shflag** определяет флаги совместного доступа к файлу нескольких приложений. Значения этих флагов приведены в разд. 3.5.1. При успешном завершении функция возвращает дескриптор открытого файла и его указатель устанавливается на начало файла. При ошибке возвращается -1, а **errno** (см. разд. 3.1.4.1) может принимать значения **EINVACC, EACCES, ENOENT, EMFILE.**

Функция **creattemp** создает временный файл с уникальным именем и атрибутами **attrib** в каталоге **path.** Вид файла — текстовый или двоичный, определяется значением глобальной переменной **_fmode (O_TEXT** или **O_BINARY).** Параметр **attrib** может равняться нулю или принимать уже рассмотренные значения **FA_HIDDEN, FA_RDONLY** или **FA_SYSTEM** (см. разд. 3.5.1).

Число файлов одновременно открытых перечисленными функциями, не должно превышать **HANDLE_MAX.**

Функции **close, _rtl_close и FileClose** закрывают файл, открытый ранее функциями **creat, creatnew, creattemp, dup, dup2, open, _rtl_creat, _rtl_open, FileOpen.** При этом в выходной файл не записывается автоматически признак конца файла Ctrl-Z. Если этот символ требуется, вам надо предварительно записать его явным образом.

При успешном завершении функций они возвращают 0. При ошибке возвращают значение -1 и задают глобальной переменной **errno** (см. разд. 3.1.4.1) значение **EBADF.**

Таким образом, стандартная схема работы с файлами, связанными с дескрипторами, следующая:

```
int hout = open("output.txt", O_CREAT I O_WRONLY,S_IWRITE);
...
close(hout);
```

Например, операторы

```
int handle;
if ((handle = open ("Test.txt", O_CREAT | O_TEXT)) == -1)
{
 ShowMessage("Файл не удается создать");
```

```
    return;
}
...
close(handle);
```

пытаются создать новый текстовый файл, а в случае неудачи (функция **open** вернула -1) отображают сообщение об ошибке.

Функции **FileOpen, FileCreate и FileClose** дают альтернативный подход к открытию и закрытию файлов, связанных с дескрипторами. Функция **FileOpen** открывает файл в режиме **Mode,** задаваемом константами **fmShare** (см. разд. 3.5.1). В дальнейшем с этими файлами можно работать с помощью функций **FileRead, FileWrite, FileSeek,** описанных в разд. 3.5.4.

При совместном доступе к файлам нескольких приложений помимо установки флагов доступа может использоваться блокировка и деблокировка отдельных областей файла с помощью функций **lock, unlock, locking.** При работе с этими функциями в DOS предварительно должна быть загружена программа SHARE.EXE. В функции **locking** режим работы определяется параметром cmd:

LK_LOCK	Блокировать область. Если не удалось, то прежде, чем отказаться от блокировки, делается новая попытка через 10 секунд.
LK_RLCK	То же, что LK_LOCK.
LK_NBLCK	Блокировать область. Если не удалось, то происходит отказ от попытки блокировки.
LK_NBRLCK	То же, что LK_NBLCK.
LK_UNLCK	Разблокировать ранее заблокированную область файла.

При успешном завершении функции возвращается 0. При неудаче возвращается -1, а значение **errno** (см. разд. 3.1.4.1) может иметь значения **EACCES, EBADF, EDEADLOCK** — невозможность блокировки, несмотря на повторную попытку через 10 секунд (при **cmd** равном **LK_LOCK** или **LK_RLCK), EINVAL** — ошибка в cmd или не загружена программа **SHARE.EXE.**

Функции **dup** и **dup2** позволяют оперировать с дескрипторами файлов. Функция **dup** создает и возвращает дубликат (псевдоним) дескриптора **handle.** Дубликат связан с тем же открытым файлом или устройством, что и исходный дескриптор, имеет тот же режим доступа (только чтение, только запись, чтение и запись) и имеет тот же указатель позиции в файле. Изменение указателя в одном из дескрипторов приводит к синхронному сдвигу указателя в другом дескрипторе. Функция **dup2** создает и возвращает аналогичный функции **dup** дубликат **newhandle** дескриптора **oldhandle.** Функции **dup** и **dup2** могут использоваться, в частности, для перенаправления стандартных потоков. Например, операторы

```
int hout = open("output.txt", O_CREAT | O_WRONLY,S_IWRITE);
dup2(hout,1);
```

перенаправляют стандартный выходной поток **stdout** (его дескриптор равен 1 — см. разд. 3.5.1) в файл "output.txt".

3.5.4 Функции ввода/вывода

Функция	Синтаксис / Описание	Файл
_fgetchar	**int fgetchar(void)** **Вводит символ из потока stdin**	*stdio.h*

Функция	Синтаксис / Описание	Файл
_fgetwchar	wint t fgetwchar(void) Вводит символ из потока stdin	*stdio.h*
_fputchar	int fputchar(int c) Выводит символ с в поток stdout, то же, что fputc(c, stdout); при ошибке возвращает EOF	*stdio.h*
_fputwchar	wint t fputwchar(int c) Выводит символ с в поток stdout, то же, что fputc(c, stdout); при ошибке возвращает EOF	*stdio.h*
_getw	int _getw(FILE *stream) Вводит целое число из потока stream	*stdio.h*
_getws	wchar_t *_getws(wchar_t *s) Читает строку из стандартного входного потока stdin	*stdio.h*
cgets	char *cgets(char *str) Читает строку символов с консоли	*conio.h*
clearerr	void clearerr(FILE *stream) Очищает индикаторы ошибок и конца файла потока stream	*stdio.h*
cprintf	int cprintf(cbnst char *format [, argument, ...] Выводит на экран список аргументов argument по формату format (см. разд. 3.1.3.1)	*conio.h*
cputs	int cputs(const char *str) Выводит строку на экран; возвращает последний символ	*conio.h*
cscanf	int cscanf(char *format [, address, ...]) Вводит данные с консоли в список аргументов по адресам argument по формату format; возвращает число успешно введенных полей или EOF при конце файла	*conio.h*
eof	int eof(int handle) Возвращает 0 при достижении конца потока (файла), связанного с дескриптором handle	*io.h*
feof	int feof(FILE *stream) Возвращает 0 при достижении конца потока (файла) stream	*stdio.h*
ferror	int ferror(FILE *stream) Проверяет ошибки ввода/вывода потока stream и возвращает 0 при отсутствии ошибки	*stdio.h*
fgetc	int fgetc(FILE *stream) Вводит символ из потока stream и возвращает его обратно	*stdio.h*
fgetpos	int fgetpos(FILE *stream, fpos_t *pos) Заносит в pos текущую позицию файла stream; при успехе возвращает 0	*stdio.h*

Функция	Синтаксис / Описание	Файл
fgets	**char *fgets(char** *s, **int** n, FILE ***stream)** Вводит в s и возвращает строку до n символов из потока stream	*stdio.h*
FileRead	int FileRead(int Handle, void *Buffer, int Count) Выводит из файла с дескриптором Handle, открытого функциями FileOpen или FileCreate, Count байтов в буфер Buffer; возвращает число прочитанных байтов или −1	*SysUtils.hpp*
FileSeek	int FileSeek(int Handle, int Offset, int Origin) Перемещает указатель файла с дескриптором Handle, открытого функциями FileOpen или FileCreate, на Offset байтов от позиции Origin; возвращает 0 при успешном завершении	*SysUtils.hpp*
FileWrite	int **FileWrite(int** Handle, const void *Buffer, int Count) Вводит в файл с дескриптором Handle, открытый функциями FileOpen или FileCreate, Count байтов из буфера Buffer; возвращает число записанных байтов или -1	*SysUtils.hpp*
fprintf	int fprintf(FILE *stream, const char *format [, argument, ...]) Выводит в файл stream список аргументов argument по формату format; возвращает число успешно записанных байтов или EOF при ошибке	*stdio.h*
fputc	int fputc(int c, FILE ***stream)** Выводит символ с в поток stream	*stdio.h*
fputs	int fputs(const char *s, FILE *stream) Выводит строку s в поток stream; при ошибке возвращает EOF	*stdio.h*
fputwc	wint_t **fputwc(wint_t** c, FILE *stream) Выводит символ с в поток stream	*stdio.h*
fputws	int fputws(const **wchar_t** *s, FILE ***stream)** Выводит строку s в поток stream; при ошибке возвращает EOF	*stdio.h*
fread	size_t fread(void *ptr, size_t size, size_t n, FILE *stream) Неформатированное чтение из stream в ptr n элементов данных размером size каждый; возвращает число успешно прочитанных байтов (n * size)	*stdio.h*
fscanf	int fscanf(FILE *stream, const char *format [, address, ...]) Вводит данные из файла stream в список аргументов по адресам argument по формату format; возвращает число успешно введенных полей или EOF при конце файла	*stdio.h*

Функция	Синтаксис / Описание	Файл
fseek	int fseek(FILE *stream, long offset, int fromwhere) Перемещает указатель файла stream на offset байтов от позиции fromwhere; возвращает 0 при успешном завершении	*stdio.h*
fsetpos	int fsetpos(FILE *stream, const fpos_t *pos) Устанавливает указатель потока stream в позицию pos	*stdio.h*
ftell	long int ftell(FILE *stream) Возвращает текущую позицию файла stream	*stdio.h*
fwrite	size_t fwrite(const void *ptr, size t size, size t n, FILE *stream) Неформатированная запись из ptr в stream n элементов данных размером size каждый; возвращает число успешно записанных байтов (n * size)	*stdio.h*
getc	int getc(FILE *stream) Вводит символ из потока stream	*stdio.h*
getch	int getch(void) Вводит символ с консоли без эхо на экране	*conio.h*
getchar	int getchar(void) Вводит символ из stdin; то же, что getc(stdin)	*stdio.h*
getche	int getche(void) Вводит символ с консоли с эхо на экране	*conio.h*
getpass	char *getpass(const char *prompt) Вводит пароль с консоли до восьми символов после печати на экране приглашения prompt; возвращает введенную строку	*conio.h*
gets	char *gets(char *s) Вводит строку из stdin	*stdio.h*
kbhit	int kbhit(void) Проверяет нажатие клавиши консоли; если ни одна клавиша не нажата — возвращает 0	*conio.h*
lseek	long lseek(int handle, long offset, int fromwhere) Перемещает указатель файла с дескриптором handle на offset байтов от позиции fromwhere; возвращает 0 при успешном завершении	*io.h*
perror	void perror(const char *s) Выводит в стандартный выходной поток ошибок сообщение s	*stdio.h*
printf	int printf(const char *format [, argument, ...]) Выводит в стандартный поток stdout список аргументов argument по формату format (см. разд. 3.1.2.1)	*stdio.h*
putc	int putc(int c, FILE *stream) Выводит символ c в поток stream	*stdio.h*

Функция	Синтаксис / Описание	Файл
putch	int putch(int c) Выводит символ с на экран	*conio.h*
putchar	int putchar(int c) Макрос, выводящий символ с в stdout; эквивалентен putc(c, stdout)	*stdio.h*
puts	int puts(const char *s) Выводит строку s в stdout и добавляет симв'ол новой строки	*stdio.h*
_putws	int _putws(const wchar_t *s) Выводит строку s в stdout и добавляет символ новой строки	*stdio.h*
puttext	int puttext(int left, int top, int right, int bottom, void *source) Выводит содержимое блока памяти source на экран в текстовом режиме в прямоугольник с координатами left, top, right, bottom	*conio.h*
putw	int _putw(int w, FILE *stream) Выводит в поток stream целое w	*stdio.h*
read	int read(int handle, void *buf, unsigned len) Чтение из файла с дескриптором handle в буфер buf len байтов	*io.h*
scanf	int scanf(const char *format [, address, ...]) Вводит данные из потока stdin в список аргументов по адресам argument по формату format (см. разд. 3.1.3.2)	*stdio.h*
sprintf	int sprintf(char *buffer, const char *format [, argument, ...]) Выводит в строку buffer список аргументов argument по формату format (см. разд. 3.1.3.1)	*conio.h*
sscanf	int sscanf(const char *buffer, const char *format [, address, ...]) Вводит данные из строки buffer в список аргументов по адресам argument по формату format	*conio.h*
tell	long tell(int handle) Возвращает текущую позицию файла с дескриптором handle	*io.h*
ungetc	int ungetc(int c, FILE *stream) Возвращает символ ch в поток stream, чтобы он стал следующим символом для чтения	*stdio.h*
ungetch	int ungetch(int ch) Возвращает символ ch на консоль, чтобы он стал следующим символом для чтения	*conio.h*

Функция	Синтаксис / Описание	Файл
vfprintf	int vfprintf(FILE *stream, const char *format, va_list arglist) Выводит в файл stream список аргументов arglist по формату format (см. разд. 3.1.3.1)	stdio.h
vfscanf	int vfscanf(FILE *stream, const char *format, va_list arglist) Вводит данные из потока stream в список адресов аргументов arglist по формату format (см. разд. 3.1.3.2)	stdio.h
vprintf	int vprintf(const char *format, va_list arglist) Выводит в stdout список аргументов arglist по формату format (см. разд. 3.1.3.1)	stdarg.h
vscanf	int vscanf(const char *format, va_list arglist) Вводит данные из потока stdin в список адресов аргументов arglist по формату format (см. разд. 3.1.3.2)	stdarg.h
vsprintf	int vsprintf(char *buffer, const char *format, va_list arglist) Выводит в строку buffer список аргументов arglist по формату format (см. разд. 3.1.3.1)	stdarg.h
vsscanf	int vsscanf(const char *buffer, const char *format, va_list arglist) Вводит данные из буфера buffer в список адресов аргументов arglist по формату format (см. разд. 3.1.3.2)	stdarg.h
write	int write(int handle, void *buf, unsigned len) Выводит в файл с дескриптором handle из буфера buf len байтов	io.h

Комментарии

Для работы с текстовыми файлами чаще всего используются функции **scanf** для чтения и **printf** для записи.

Функции **printf, sprintf, vfprintf, vprintf, vsprintf** производят форматированный вывод данных из списка указанных в них аргументов. Строка форматирования подробно рассмотрена в разд. 3.1.3.1. Функции возвращают число успешно записанных байтов или EOF при ошибке.

Функции **fscanf, scanf, cscanf, sscanf, vfscanf, vscanf, vsscanf** производят форматированный ввод данных в список адресов аргументов. Строка форматирования подробно рассмотрена в разд. 3.1.3.2.

Функции **getchar, _fgetchar, cgets, cprintf, _fputchar, putch, puttext, getch, getche, cputs, getpass, cscanf, kbhit, ungetc, ungetch, vprintf, vscanf** не могут использоваться в приложениях с графическим интерфейсом для Win32.

В приложениях с графическим интерфейсом для Win32 при использовании функций **scanf, gets** должен быть перенаправлен поток **stdin,** а при использовании функций **printf, putchar, puts** должен быть перенаправлен поток **stdout.**

Функции **getchar, _fgetchar, getc, fgetc** возвращают читаемый символ, преобразованный в целое без знака.

Функции **getchar, _fgetchar, getc, gets, fgets, fgetc, fputc** при ошибке преобразования или при окончании файла возвращают EOF.

Функции **getc, fgetc, getchar** после чтения возвращают символ в поток. При этом функции **getc** увеличивают указатель потока на 1, подготавливая чтение следующего символа.

Функция **_getw** возвращает целое, прочитанное из потока. Поток (файл) должен быть открыт в текстовом режиме. Функции **putw, puts** записывают соответственно целое и строку в поток. При ошибке преобразования или при окончании файла все эти функции возвращают **EOF.** Поскольку EOF является допустимым возвращаемым значением, для проверки конца файла или ошибки преобразования надо использовать функции **feof и ferror.**

Функция gets читает последовательность символов до символа конца строки, который не помещает в возвращаемую строку, заменяя его нулевым символом.

Функция **fgets** читает последовательность символов до заданного числа символов п или до символа конца строки, который помещает в возвращаемую строку, помещая после него нулевой символ.

Функция **puttext,** используемая только в консольных приложениях, выводит содержимое блока памяти, на который указывает **source,** на экран в текстовом режиме в прямоугольник с координатами **left, top, right, bottom.** Координаты левого верхнего угла (1,1). Каждой позиции на экране соответствуют 2 байта, первый из которых — символ, а второй — атрибуты вывода. Функция возвращает ненулевое значение при успешном выводе и 0 — при ошибке.

Функции fseek и **lseek** перемещают указатель файла на **offset** байтов от позиции **fromwhere.** Для текстового файла параметр **offset** должен быть равен 0 или соответствовать допустимому значению, возвращенному функцией **ftell.** Параметр **fromwhere,** определяющий точку, относительно которой производится смещение **offset,** может принимать значения:

SEEK_SET	0	начало файла
SEEK_CUR	1	текущая позиция файла
SEEK_END	2	конец файла

Функции возвращают 0 при успешном завершении.

Функции **FileRead, FileWrite и FileSeek** используются для работы с файлами, открытыми функциями **FileOpen** или **FileCreate.**

3.5.5 Функции обработки имен файлов

Функция	Синтаксис / Описание	Файл .
_mktemp	**char *_mktemp(char *template)** Генерирует, заносит в **template** и возвращает уникальное имя файла, которое может в дальнейшем использоваться для создания временных файлов	*dir.h*
ChangeFileExt	**System::AnsiString ChangeFileExt(const System::AnsiString FileName, const System::AnsiString Extension)** Возвращает имя файла **FileName** с измененным расширением на **Extension;** сам файл не переименовывается	*SysUtils.hpp*

Функция	Синтаксис / Описание	Файл
ExpandFile Name	System::AnsiString ExpandFileName(const System::AnsiString FileName) Расширяет имя файла **FileName,** добавляя к нему текущие путь и диск; проверка существования такого файла не проводится	*SysUtils.hpp*
ExpandUNC FileName	System::AnsiString ExpandUNCFileName(const System::AnsiString FileName) Расширяет имя файла **FileName,** добавляя к нему текущие путь и том в формате UNC: "\\<servername>\<sharename>", если том указывает на сеть	*SysUtils.hpp*
ExtractFileDir	System::AnsiString ExtractFileDir(const System::AnsiString FileName) Извлекает из **FileName** и возвращает путь к файлу	*SysUtils.hpp*
ExtractFile Drive	System::AnsiString ExtractFileDrive(const System::AnsiString FileName) Возвращает диск файла **FileName** (например, "c:") или в формате UNC: "\\<servername>\<sharename>", если путь указывает на сеть	*SysUtils.hpp*
ExtractFileExt	System::AnsiString ExtractFileExt(const System::AnsiString FileName) Возвращает расширение файла **FileName**	*SysUtils.hpp*
ExtractFile Name	System::AnsiString ExtractFileName(const System::AnsiString FileName) Возвращает имя файла, извлеченное из **FileName,** т.е. конец строки после последнего обратного слэша или двоеточия	*SysUtils.hpp*
ExtractFile Path	System::AnsiString ExtractFilePath(const System::AnsiString FileName) Возвращает путь к файлу, извлеченный из **FileName,** включая последний обратный слэш или двоеточие, отделяющие путь от имени	*SysUtils.hpp*
Extract RelativePath	System::AnsiString ExtractRelativePath(const System::AnsiString BaseName, const System::AnsiString DestName) Возвращает относительный путь файла **DestName** относительно каталога **BaseName,** включая форматы вида "..\"	*SysUtils.hpp*
ExtractShort PathName	System::AnsiString ExtractShortPathName(const System::AnsiString FileName) Возвращает путь и имя файла **FileName,** преобразовывая имена в формат 8.3	*SysUtils.hpp*
MatchesMask	bool MatchesMask(const System::AnsiString Filename, const System::AnsiString Mask) **Проверяет Filename** на использование маски **Mask**	*Masks.hpp*

Функция	Синтаксис / Описание	Файл
MinimizeName	**System::AnsiString MinimizeName(const System:: AnsiString Filename, Graphics::TCanvas * Canvas, int MaxLen)** **Минимизирует имя Filename, сокращая путь до размера, вмещающегося при изображении на канве Canvas в MaxLen пикселов**	*filectrl.hpp*
ProcessPath	**void ProcessPath(const System::AnsiString EditText, char &Drive, System::AnsiString &DirPart, System::AnsiString &FilePart)** **Разделяет путь процесса EditText на драйвер Drive, путь DirPart и имя FilePart**	*filectrl.hpp*
tmpnam	**char *tmpnam(char *s)** **Создает уникальное имя файла, которое может в дальнейшем использоваться для создания временных файлов**	*stdio.h*

Комментарии

Все функции (кроме **tmpnam и _mktemp)** возвращают строку типа **AnsiString** (см. разд. 2.5.2 и 3.1.7), содержащую результат преобразования имени файла **FileName.** Могут работать с многобайтными символами. Если **FileName** не содержит пути или расширения, то соответствующие функции **ExtractFile...** возвращают пустую строку.

Не забывайте, что обратный слэш в строке пути к файлу должен повторяться дважды. Например:

```
"C:\\Program Files\\Bcb.exe"
```

Функция **ExtractFileDir** возвращает путь к файлу в том виде, который нужен для передачи в функции **CreateDir, GetCurrentDir, RemoveDir, SetCurrentDir** (см. разд. 3.5.6).

Функция **ExtractShortPathName** возвращает путь и имя файла **FileName,** сокращая имена каталогов и файлов до формата 8.3. Например, строка

```
C:\Program Files\Borland\CBuilder\Bin\Bcb.exe
```

будет возвращена как

```
C:\Progra~1\Borland\CBuilder\Bin\Bcb.exe
```

Функция **MatchesMask** проверяет **Filename** на использование маски **Mask.** Маска может содержать обычные алфавитно-цифровые символы, множества, символы "*" — любое количество любых символов и "?" — любой один символ. Множества заключаются в квадратные скобки [ ]. В скобках без пробелов записываются возможные символы или диапазоны в виде <символ>-<символ>. Например, "a-c". Если первый символ в множестве — восклицательный знак "!", то это множество содержит символы, которые не должны встречаться. Сравнение с маской призводится без учета регистра. Функция **MatchesMask** возвращает **true,** если **Filename** соответствует маске, **false,** если не соответствует и генерирует исключение при синтаксически неверной маске. Например, маске

```
"[a-b]:\\test\\*.*"
```

будут соответствовать все имена файлов, расположенных на дисководах a: или b: в каталоге "test".

Функция **MinimizeName** минимизирует имя **Filename,** сокращая путь до размера, вмещающегося при изображении на канве **Canvas** в **MaxLen** пикселов. Вместо выброшенных частей пути изображаются точки. Например, оператор

```
Label1->Caption = MinimizeName("C:\\Program Files\\Bcb.exe",
                               Label1->Canvas, 100);
```

приведет к появлению в метке Label1 надписи:

```
C:\...\Bcb.exe
```

Функции **tmpnam** и **_mktemp** создают уникальное имя файла, которое может в дальнейшем использоваться для создания временных файлов. Последовательное обращение к функции **tmpnam** может создавать до **TMP_MAX** = 65 535 уникальных имен. Параметр s этой функции должен задаваться или равным **NULL,** или указывать на массив размером не менее **L_tmpnam** символов (эта константа определена в *stdio.h*). В случае параметра равного **NULL,** функция **tmpnam** сама создает необходимый объект с именем файла и возвращает указатель на него.

Если вы создаете затем временный файл с именем, сгенерированным **tmpnam или _mktemp,** то должны сами позаботиться о его удалении в конце работы программы. Автоматически он не удаляется.

3.5.6 Управление каталогами и файлами на дисках

Функция	Синтаксис / Описание	Файл
_getdcwd	**char** * **_getdcwd(int** drive, **char *buffer,** int buflen) Заносит в буфер buffer размером buflen текущий каталог диска drive (0 — текущий диск, 1 — A и т.д.); возвращает указатель на buffer или NULL; при buffer = NULL создает буфер и возвращает указатель на него	*direct.h*
_rmdir	int **_rmdir(const char** *path) Удаляет каталог path (пустой, не текущий и не корневой); возвращает 0 при успехе или -1	*dir.h*
_rtl_chmod	int **_rtl_chmod(const** char *path, int func [, int attrib]) При func = 0 возвращает текущие атрибуты файла, при func = 1 устанавливает файлу атрибуты attrib	*io.h, dos.h*
_unlink	int _unlink(const char *filename) Удаляет с диска файл filename; возвращает 0 или -1	*io.h*
_waccess	int **_waccess(const** wchar_t *filename, int **amode)** Определяет, существует ли файл filename и какие операции с ним доступны; режим работы задается параметром amode	*io.h*
_wrtl_chmod	int **_wrtl_chmod(const wchar_t** *path, int func, ...) При func = 0 возвращает текущие атрибуты файла, при func = 1 устанавливает файлу атрибуты attrib	*io.h, dos.h*
access	int access(const char ***filename,** int amode) Определяет, существует ли файл filename и какие операции с ним доступны; режим работы задается параметром amode	*io.h*

Функция	Синтаксис / Описание	Файл
chdir	int **chdir(const char *path)** Задает path в качестве текущего каталога; возвращает 0 при успехе или -1	*dir.h*
chmod	int **chmod(const** char *path, int amode) Изменяет режим доступа amode к файлу path; возвращает 0 при успехе или −1; amode содержит одно или оба значения **S_IWRITE** и S_IREAD	*io.h*
chsize	int chsize(int handle, long size) Изменяет размер файла с дескриптором handle, открытого для записи, до size байтов; возвращает 0 или -1	*io.h*
CreateDir	**bool** CreateDir(const System::AnsiString Dir) Создает каталог Dir и возвращает true в случае успеха	*SysUtils.hpp*
DeleteFile	bool DeleteFile(const **System::AnsiString** FileName) Удаляет файл FileName с диска и возвращает true в случае успеха	*SysUtils.hpp*
Directory **Exists**	**bool DirectoryExists(const System::AnsiString** Name) Определяет, существует ли каталог Name	*SysUtils.hpp*
DiskFree	int DiskFree(Byte Drive) Возвращает число свободных байтов на диске Drive или −1, если Drive ошибочный (Drive =0 — текущий диск, 1 — A, 2 — B и т.д.)	*SysUtils.hpp*
DiskSize	int DiskSize(Byte Drive) Возвращает размер в байтах диска Drive или −1, если Drive ошибочный (Drive =0 — текущий диск, 1 — A, 2 — B и т.д.)	*SysUtils.hpp*
FileAge	int FileAge(const System: :AnsiString FileName) Возвращает дату создания файла FileName или -1, если такого файла нет	*SysUtils.hpp*
FileDateTo DateTime	**System::TDateTime FileDateToDateTime(** int FileDate) Возвращает в формате типа **TDateTime** дату и время FileDate, заданные в формате DOS	*SysUtils.hpp*
FileExists	bool FileExists(const System::AnsiString FileName) Определяет, существует ли файл FileName	*SysUtils.hpp*
FileGetAttr	int FileGetAttr(const System::AnsiString FileName) Возвращает атрибуты файла FileName	*SysUtils.hpp*
FileGetDate	int FileGetDate(int Handle) Возвращает дату создания файла с дескриптором Handle или -1, если такого файла нет	*SysUtils.hpp*

Функция	Синтаксис / Описание	Файл
filelength	long filelength(int handle) Возвращает длину в байтах файла с дескриптором handle; при ошибке возвращает -1	*io.h*
FileSearch	System::AnsiString FileSearch(const System:: AnsiString Name, const System::AnsiString DirList) Ищет в списке каталогов DirList файл Name; возвращает полный путь к файлу или пустую строку	*SysUtils.hpp*
FileSetAttr	int FileSetAttr(const System::AnsiString FileName, int Attr) Устанавливает файлу FileName атрибуты Attr; возвращает 0 или код ошибки	*SysUtils.hpp*
FileSetDate	int FileSetDate(int Handle, int Age) Устанавливает дату Age файлу с дескриптором Handle; возвращает 0 или код ошибки	*SysUtils.hpp*
FindClose	void FindClose(TSearchRec &F) Завершает последовательность поиска функциями FindFirst и FindNext со структурой F и освобождает память	*SysUtils.hpp*
FindFirst	int FindFirst(const System::AnsiString Path, int Attr, TSearchRec &F) Начинает поиск файлов по шаблону Path с атрибутами Attr; заносит результат в F; возвращает 0 или код ошибки	*SysUtils.hpp*
findfirst	int findfirst(const char _FAR *__path, struct ffblk _FAR *___ffblk, int____attrib) Начинает поиск файлов по шаблону__path с атрибутами___attrib; заносит результат в___ffblk; возвращает 0 при успехе или -1	*dir.h*
_wfindfirst	int _wfindfirst(const wchar_t *pathname, struct _wffblk *ffblk, int attrib) Начинает поиск файлов по шаблону pathname с атрибутами attrib; заносит результат в _wffblk; возвращает 0 при успехе или -1	*dir.h*
FindNext	int FindNext(TSearchRec &F) Продолжает поиск файлов, начатый функцией FindFirst со структурой F; заносит результат в F; возвращает 0 или код ошибки	*SysUtils.hpp*
findnext	int findnext(struct ffblk _FAR * __ffblk) Продолжает поиск файлов, начатый функцией findfirst со структурой___ffblk; возвращает 0 при успехе или -1	*dir.h*
_wfindnext	int _wfindnext(struct _wffblk *ffblk) Продолжает поиск файлов, начатый функцией _wfindfirst со структурой ffblk; возвращает 0 при успехе или -1	*dir.h*

Функция	Синтаксис / Описание	Файл
fnmerge	void **fnmerge(char** *path, const **char *drive,** const char **dir,** const char *name, const **char** *ext) Формирует строку path пути к файлу из его отдельных составляющих: диска drive, каталога dir, имени файла name и расширения ext	*dir.h*
fnsplit	int fnsplit(const char *path, char **drive,** char **dir,** char *name, char *ext) Разделяет строку path пути к файлу на его отдельные составляющие: диск drive, каталог dir, имя файла name и расширение ext	*dir.h*
Force Directories	void **ForceDirectories(System::AnsiString Dir)** Создает каталог Dir и все промежуточные родительские каталоги, если они отсутствуют	*FileCtrl.hpp*
fstat	int fstat(int handle, struct stat *statbuf) Заносит в структуру statbuf информацию об открытом файле с дескриптором handle; возвращает 0 или -1	*sys\stat.h*
getcurdir	int getcurdir(int drive, char **directory)** Заносит в directory текущий каталог диска drive (0 — текущий диск, 1 — A и т.д.) без имени диска и начального символа "\"	*dir.h*
GetCurrent Dir	System::AnsiString **GetCurrentDir()** Возвращает текущий каталог	*SysUtils.hpp*
getcwd	char ***getcwd(char *buf,** int buflen) Возвращает и сохраняет в буфере buf размером buflen полный путь к текущему каталогу, включая диск; возвращает указатель на buf или NULL; при buf = NULL создает буфер и возвращает указатель на него	*dir.h*
getdisk	int getdisk(void) Возвращает текущий диск: 0 — A, 1 — B и т.д.	*dir.h*
getftime	int **getftime(int** handle, struct ftime ***ftimep)** Читает время и дату создания файла handle в структуру **ftimep;** возвращает 0 или -1	*io.h*
GetSystem Directory	**UINT** GetSystemDirectory(LPTSTR **lpBuffer,** **UINT** uSize) Функция API Windows, заносит в буфер IpBuffer размером uSize системный каталог Windows	–
GetWindows Directory	UINT **GetWindowsDirectory(LPTSTR** IpBuffer, UINT uSize) Функция API Windows, заносит в буфер IpBuffer размером uSize каталог Windows	–

Функция	Синтаксис / Описание	Файл
isatty	int isatty(int handle) Возвращает ненулевое значение, если файл с дескриптором handle связан с одним из устройств: терминал, консоль, принтер, последовательный порт	*io.h*
mkdir	int **mkdir(const** char *path) Создает каталог path; возвращает 0 при успехе или -1	*dir.h*
remove	int remove(const char *filename) Макрос, удаляет с диска файл filename; возвращает 0 или -1	*stdio.h*
RemoveDir	**bool RemoveDir(const System::AnsiString** Dir) Удаляет с диска каталог **Dir**	*SysUtils.hpp*
rename	int **rename(const** char ***oldname,** const char ***newname)** Переименовывает файл **oldname,** давая ему новое имя **newname;** может использоваться для перемещения файла без изменения диска; возвращает 0 или -1	*stdio.h*
RenameFile	bool RenameFile(const **System::AnsiString** OldName, const **System::AnsiString NewName)** Переименовывает файл OldName, давая ему новое имя NewName; если файл с именем NewName уже существует или нет файла OldName, возвращается false	*SysUtils.hpp*
searchpath	char ***searchpath(const** char *file) Ищет файл file в каталогах, указанных в переменной окружения PATH; возвращает полный путь к файлу или NULL	*dir.h*
SetCurrent Dir	bool **SetCurrentDir(const** System::AnsiString **Dir)** Задает **Dir** в качестве текущего каталога	*SysUtils.hpp*
setdisk	int setdisk(int drive) Устанавливает в качестве текущего диск drive: 0 — A, 1 — B и т.д.; возвращает число доступных дисков	*dir.h*
setftime	int setftime(int handle, struct ftime ***ftimep)** Устанавливает время и дату создания файла handle по данным структуры **ftimep;** возвращает 0 или -1	*io.h*
stat	int stat(const char *path, struct stat *statbuf) Заносит в структуру statbuf информацию об открытом файле path; возвращает 0 или -1	*sys\stat.h*

Комментарии

Функции **fstat и stat** заносят в структуру типа **stat** информацию об открытом файле. Структура имеет поля:

st_mode	битовая маска режима файла
st_dev	номер диска файла или дескриптор, если файл на устройстве

st_mode	битовая маска режима файла
st_rdev	то же, что st_dev
st_nlink	константа 1
st_size	размер файла в байтах
st_atime	время последнего открытия (в Windows) или изменения (в DOS)
st_mtimc	то же, что st_atime
st_ctime	то же, что st_atime

Маска **st_mode** содержит информацию о режиме открытого файла и включает в себя следующие биты.

Должен быть установлен один из следующих битов:

S_IFCHR	если дескриптор ссылается на устройство
S_IFREG	если дескриптор ссылается на обычный файл

Должен быть установлен один или оба следующих битов:

S_IWRITE	пользователю разрешена запись в файл
S_IREAD	пользователю разрешено чтение из файла

Системы HPFS и NTFS учитывают следующее различие между полями **st_atime, st_mtime и st_ctime:**

st_atime	время последнего доступа к файлу
st_mtime	время последней модификации файла
st_ctime	время создания файла

Следующий пример демонстрирует работу функции **stat:**

```
#include <stdio.h>
#include <time.h>
#include <sys\stat.h>
...
struct stat statbuf;
FILE *stream;
if ((stream = fopen("TEST.TXT", "r+t")) == NOLL)
{
  ShowMessage("Невозможно открыть файл");
  return;
}
stat("TEST.TXT", &statbuf);          // чтение информации
fclose(stream);

// отображение информации:
if (statbuf.st_mode & S_IFCHR)
   Memo1->Lines->Add("Дескриптор устройства");
if (statbuf.st_mode & S_IFREG)
   Memo1->Lines->Add("Ссылка на файл");
if (statbuf.st_mode & S_IREAD)
   Memo1->Lines->Add("Разрешено чтение из файла");
if (statbuf.st_mode & S_IWRITE)
   Memo1->Lines->Add("Разрешена запись в файл");
```

```
Memo1->Lines->Add("Файл расположен на диске " +
                  AnsiString((char)('A'+statbuf.st_dev)));
Memo1->Lines->Add("Размер файла в байтах: " +
                  IntToStr(statbuf.st_size));
Memo1->Lines->Add("Последний раз файл был открыт " +
                  AnsiString(ctime(&statbuf.st_ctime)));
```

Функции **remove** и **_unlink** удаляют с диска указанный файл. Если файл открыт, то перед удалением его надо закрыть. Файл с атрибутом только для чтения не может быть удален. Сначала надо изменить его атрибут функциями **chmod** или **_rtl_chmod.**

Функции **access** и **_waccess** определяют разрешенный доступ к файлу, заданному параметром filename. Различаются ени только типом строки, содержащей имя файла. Функции проверяют, существует ли файл, и если существует, то разрешено ли чтение, запись или выполнение этого файла.

Параметр amode определяет, что именно должно проверяться:

06	Проверка разрешения чтения и записи
04	Проверка разрешения чтения
02	Проверка разрешения записи
01	Проверка, является ли файл выполняемым
00	Проверка существования файла

В DOS, OS/2 и Windows все существующие файлы имеют разрешение чтения. Поэтому значения **amode** 00 и 04 дают одинаковый результат. В DOS разрешение записи подразумевает и разрешение чтения. Поэтому 06 и 02 также дают одинаковый результат.

Если в качестве **filename** задан не файл, а каталог, то функции просто проверяют, существует ли каталог.

Функции возвращают 0, если файл имеет запрошенный уровень доступа. В противном случае возвращается 1, а глобальная переменная **errno** устанавливается в одно из следующих состояний:

ENOENT путь или файл не найден
EACCES ошибка доступа

Функция **FileSearch** ищет в списке каталогов **DirList** файл **Name.** Список **DirList** представляет собой перечень обычным образом записанных путей к каталогам, разделенных точками с запятой. При успешном поиске функция возвращает имя файла с полным путем к нему (если файл найден в текущем каталоге, возвращается только имя файла). Если файл не найден, возвращается пустая строка.

Функция **searchpath** ищет файл **file** в списке каталогов, указанных в переменной окружения PATH. Возвращает полный путь к файлу или **NULL.**

Функция **FileExists** определяет, существует ли указанный файл.

Функция **fnmerge** формирует строку **path** пути к файлу из его отдельных составляющих: диска **drive,** каталога **dir,** имени файла **name** и расширения ext. В итоге получается строка вида **drive:\path\name.ext.** Каждая из составляющих пути **(dir, path, name, ext)** может быть задана равной **NULL.** Тогда эта составляющая в результирующий путь не включается. Функция **fnsplit** осуществляет обратную операцию: разделяет полный путь к файлу на его составляющие.

Функция **DirectoryExists** определяет, существует ли каталог Name. Если **Name** содержит полный путь, то проверяется наличие именно указанного каталога. В противном случае **Name** воспринимается как путь относительно текущего каталога.

Функции **CreateDir** и **ForceDirectories** создают переданный им как параметр каталог **Dir.** Функция **ForceDirectories** отличается от других подобных функций тем, что она может создать не только конечный каталог, но одновременно и его родительские каталоги, если они отсутствуют. Например, оператор

```
ForceDirectories ("C:\\Test\\Test1");
```

создаст не только каталог **Test1,** но и его родительский каталог **Test,** если он отсутствует. Проверить, создан ли нужный каталог этой функцией, можно с помощью функции **DirectoryExists.**

Функции **getcurdir** и **GetCurrentDir** позволяют определить текущий каталог на заданном или текущем диске.

Функция API Windows **GetSystemDirectory,** заносит в буфер строку, характеризующую системный каталог Windows. Это тот каталог, в котором размещены файлы библиотек, драйверов, шрифтов. Приложение не должно создавать какие-то файлы в системном каталоге. Создавать свои файлы можно в каталоге, возвращаемом другой аналогичной функцией — **GetWindowsDirectory,** дающей путь к каталогу Windows. Этот каталог содержит файлы приложений Windows, файлы инициализации .ini и файлы справок **.hlp.** В этом каталоге вы можете хранить файлы инициализации и файлы справок своего приложения. Если приложение создает другие файлы, которые вы хотите хранить, не допуская к ним других пользователей, то помещайте их в каталог, указанный в переменной окружения **HOMEPATH.** При соответствующей установке этот каталог различен для всех пользователей.

Параметр **lpBuffer** функций **GetSystemDirectory** и **GetWindowsDirectory** является указателем на строку с нулевым символом в конце, в которую передается найденный путь. Этот путь записывается без заключительного обратного слэша "\", если только каталог не является корневым.

Параметр uSize указывает максимальный размер буфера в символах. Его величина должна быть не менее значения **MAX_PATH.**

При успешном выполнении функции копируют путь в **lpBuffer** и возвращают число символов в строке, не считая последнего нулевого. Если длина строки больше, чем **uSize,** то возвращенное значение позволяет узнать требуемый размер буфера.

Если функция не смогла успешно завершиться, то она возвращает нулевое значение. В этом случае узнать причину отказа можно, вызвав **GetLastError.**

Приведем пример. Если ваш системный каталог Windows назван WINDOWS\SYSTEM и расположен на диске C:, то операторы

```
Char s[MAX_PATH];
GetSystemDirectory(s,MAX_PATH);
```

занесут в s путь: C:\WINDOWS\SYSTEM. Полученный путь можно, например, использовать для проверки, имеются ли на компьютере пользователя нужные библиотеки, драйверы и шрифты.

Функции **FileGetAttr, FileSetAttr, _rtl_chmod** позволяют определять или устанавливать атрибуты файла (см. описание атрибутов в разд. 3.5.1). Устанавливая атрибуты их можно объединять в одно слово атрибутов операцией поразрядного ИЛИ. Возвращенные функциями **FileGetAttr** и **_rtl_chmod** атрибуты можно проверять с помощью операции поразрядного И.

Функция **_rtl_chmod** позволяет определить или установить атрибуты файла (см. их описание в разд. 3.5.1). При **func** = 0 функция возвращает слово текущих атрибутов файла **path, а при func** = 1 устанавливает файлу **path** атрибуты **attrib.** Например, следующий оператор устанавливает для файла, заданного строкой **SFile,** атрибут «невидимый»:

```
_rtl_chmod(SFile, 1, FA_HIDDEN);
```

Возвращенные функциями FileGetAttr и **_rtl_chmod** атрибуты можно проверять с помощью операции поразрядного И.

Следующие два оператора добавляют к атрибутам файла, заданного строкой **SFile,** атрибут «невидимый»:

```
int attrib = _rtl_chmod(SFile, 0);
_rtl_chmod(SFile, 1, attrib | FA_HIDDEN);
```

Следующие операторы определяют и отображают в окне редактирования **Memo1** атрибуты файла **SFile:**

```
int attrib = _rtl_chmod(SFile, 0);
if(attrib == -1)
{
  Memo1->Lines->Add("Ошибка номер " + IntToStr(errno));
  return;
}
if (attrib & FA_RDONLY)
  Memo1->Lines->Add(SFile + " — файл только для чтения");
if (attrib & FA_HIDDEN)
  Memo1->Lines->Add(SFile + " — невидимый файл");
if (attrib & FA_SYSTEM)
  Memo1->Lines->Add(SFile + " — системный файл");
if (attrib & FA_DIREC)
  Memo1->Lines->Add(SFile + " — каталог");
if (attrib & FA_ARCH)
  Memo1->Lines->Add(SFile + " — архивный файл");
```

Функции **FileAge, FileGetDate и FileSetDate** оперируют с данными о времени создания файла в формате DOS. В этом же формате хранится значение поля Time структуры типа **TSearchRec,** используемой в функциях **FindFirst и FindNext.** Преобразование этого формата в тип **TDateTime** может осуществляться функцией **FileDateToDateTime.** См. также разд. 3.3.2, посвященный преобразованиям форматов дат и времени.

Функция **getftime** читает время и дату создания файл, заданного своим дескриптором **handle** (он может быть определен функцией fileno), и заносит их в структуру типа ftime, на которую указывает параметр ftimep. Функция **setftime** выполняет обратную задачу: задает файлу время и дату в соответствии с данными, записанными в структуру **ftimep.** Файл должен быть доступен для записи. В противном случае произойдет ошибка **EACCES.** После установки времени и даты в файл ничего нельзя записывать, пока он не закрыт. Иначе установка будет изменена.

При успешном завершении функции возвращают 0. В противном случае возвращается **-1,** а глобальная переменная **errno** устанавливается в одно из следующих состояний:

EACCES ошибка доступа
EBADF ошибочный номер файла
EINVFNC ошибочный номер функции

Структура типа **ftime** имеет вид:

```
struct ftime {
   unsigned ft_tsec: 5;      // пары секунд
   unsigned ft_min: 6;       // минуты
   unsigned ft_hour: 5;      // часы
   unsigned ft_day: 5;       // день
   unsigned ft_month: 4;     // месяц
   unsigned ft_year: 7;      // год — 1980
   };
```

Следующий код в качестве примера определяет дату создания файла "Test.txt", уменьшает день на 1 (делает дату вчерашней — предполагается, что день не равен 1) и устанавливает файлу эту измененную дату:

```
FILE *stream;
std::ftime ft;
char buffer[80];
if ((stream= fopen("TEST.TXT", "r+t")) == NULL)
(
 ShowMessage("Невозможно открыть файл для записи");
 return;
}
getftime(fileno(stream), &ft);                    // чтение даты
sprintf(buffer, "Дата создания файла: %u/%u/%u",
        ft.ft_day, ft.ft_month, ft.ft_year+1980);
ShowMessage(buffer);
ft.ft_day--;                                      // изменение даты
setftime(fileno(stream), &ft);                    // установка даты
fclose(stream);
```

3.6 Управление процессами

3.6.1 Функции управления текущим процессом

Функция	Синтаксис / Описание	Файл
_c_exit	void _c_exit(void) Выполняет все действия, аналогичные функции exit, по закрытию файлов и очистке буферов, но не вызывает функций окончания и не прерывает выполнение программы	*process.h*
_cexit	void _cexit(void) Выполняет все действия, аналогичные функции exit, по закрытию файлов, очистке буферов и вызову функций окончания, но не прерывает выполнение программы	*process.h*
_exit	void _exit(int status) Завершает выполнение программы, но в отличие от exit не сбрасывает буферы, не закрывает файлы и не вызывает функции окончания; status — устанавливаемый код завершения	*stdlib.h*
abort	void abort(void) Аварийное завершение программы	*stdlib.h*
Abort	void Abort(void) Генерирует исключение EAbort	*SysUtils.hpp*
atexit	int atexit(void (_USERENTRY * **func)(void))** Регистрирует функцию окончания func; при успехе возвращает 0	*stdlib.h*
exit	void exit(int status) Завершает выполнение программы, закрывая все открытые файлы, сбрасывая выходные буферы в соответствующие потоки, и вызывая все зарегистрированные функцией atexit функции окончания; status - устанавливаемый код завершения	*stdlib.h*

Функция	Синтаксис / Описание	Файл
raise	int raise(int sig) Генерирует сигнал sig; при успешной генерации возвращает 0	signal.h
signal	void (USERENTRY *signal(int sig, void (_USERENTRY *func) (int sig[, int subcode])))(int) Задание обработчика func сигнала sig	signal.h

Комментарии

Функция **atexit** регистрирует в системе функцию окончания. Эта функция будет вызываться при завершении работы программы функциями **exit** и **_cexit**. Обычно в этой функции предусматривается «зачистка мусора» — освобождение динамически распределенной памяти, уничтожение временных файлов, разрыв соединений с базами данных и т.п. (см. подробнее в разд. 1.12.2.). Например, оператор

```
atexit(Exit1);
```

регистрирует функцию окончания, которую вы можете записать в виде:

```
void Exit1(void)
{
  ...
}
```

Всего в приложении может быть зарегистрировано до 32 функций окончания. При завершении приложения они срабатывают в последовательности, обратной последовательности их регистрации, т.е. последняя зарегистрированная функция будет вызываться первой.

Семейство функций **exit** выполняет операции по завершению работы приложения. Наиболее полное завершение выполняет функция **exit.** Прежде, чем завершить приложение, она закрывает все открытые файлы, сбрасывает выходные буферы в соответствующие потоки, вызывает все зарегистрированные функцией **atexit** функции окончания. Параметр **status** функции **exit** — это устанавливаемый код завершения.

Приведенная ниже таблица показывает, какие из этих операций выполняются другими функциями семейства **exit,** а какие нет.

Функция	Закрытие файлов	Сброс буферов	Вызов функций окончания	Завершение приложения
exit	+	+	+	+
_exit	–	–	–	+
_cexit	+	+	+	–
_c_exit	+	+	-	–

3.6.2 Функции выполнения порождаемых процессов

Функция	Синтаксис / Описание	Файл
CreateProcess	bool __fastcall CreateProcess(const char * lpApplicationName, char * lpCommandLine, _SECURITY ATTRIBUTES * lpProcessAttributes, _SECURITY_ATTRIBUTES * lpThreadAttributes, bool bInheritHandles, unsigned long dwCreationFlags, void * lpEnvironment, const char * lpCurrentDirectory, STARTUPINFO * lpStartupInfo, PROCESS_INFORMATION * lpProcessInformation) Порождает дочерний процесс	*winbase.h*
cwait	int cwait(int *statloc, int pid, int action); Обеспечивает ожидание завершения указанного порожденного процесса, заносит в statloc статус завершения, возвращает ID порожденного процесса или -1	*process.h*
execl	int execl(char *path, char *argO, *argl, ..., *argn, NULL) Выполняет порожденный процесс path с аргументами argO — argn	*process.h*
execle	int execle(char *path, char *argO, *argl, ..., *argn, NULL, char **env) Выполняет порожденный процесс path с аргументами arg0 — argn и с окружением env	*process.h*
execlp	int execlp(char *path, char *argO,*argl, ..., *argn, NULL) Выполняет порожденный процесс path с аргументами argO — argn, с поиском в PATH	*process.h*
execlpe	int execlpe(char *path, char *argO, *argl, ..., *argn, NULL, char **env) Выполняет порожденный процесс path с аргументами argO — argn, с поиском в PATH и с окружением env	*process.h*
execv	int execv(char *path, char *argv[]) Выполняет порожденный процесс path с аргументами argv[]	*process.h*
execve	int execve(char *path, char *argv[], char **env) Выполняет порожденный процесс path с аргументами argv[] и с окружением env	*process.h*
execvp	int execvp(char *path, char *argv[]) Выполняет порожденный процесс path с аргументами argv[], с поиском в PATH	*process.h*
execvpe	int execvpe(char *path, char *argv[], char **env) Выполняет порожденный процесс path с аргументами argv[], с поиском в PATH и с окружением env	*process.h*

Функция	Синтаксис / Описание	Файл
Find Executable	HINSTANCE FindExecutable(LPCTSTR lpFile, LPCTSTR lpDirectory, LPTSTR lpResult) Возвращает имя и путь приложения, связанного с указанным файлом	*Shell-API.h*
ShellExecute	HINSTANCE ShellExecute(HWND hwnd, LPCTSTR lpOperation, LPCTSTR IpFile, LPCTSTR lpParameters, LPCTSTR IpDirectory, INT nShowCmd) Открывает или печатает указанный файл или открывает указанную папку	*Shell-API.h*
spawnl	int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL) Выполняет в режиме mode порожденный процесс path с аргументами argO — argn	*process.h, stdio.h*
spawnle	int spawnle(int mode, char *path, char *argO, argl, ..., argn, NULL, char *envp[]) Выполняет в режиме mode порожденный процесс path с аргументами argO — argn и с окружением envp	*process.h, stdio.h*
spawnlp	int spawnlp(int mode, char *path, char *argO, arg1, ..., argn, NULL) Выполняет в режиме mode порожденный процесс path с аргументами argO — argn, с поиском в PATH	*process.h, stdio.h*
spawnlpe	int spawnlpe(int mode, char *path, char *argO, argl, ..., argn, NULL, char *envp[]) Выполняет в режиме mode порожденный процесс path с аргументами argO — argn, с поиском в PATH и с окружением envp	*process.h, stdio.h*
spawnv	int spawnv(int mode, char *path, char *argv[]) Выполняет в режиме mode порожденный процесс path с аргументами argv[]	*process.h, stdio.h*
spawnve	int spawnve(int mode, char *path, char *argv[], char *envp[]) Выполняет в режиме mode порожденный процесс path с аргументами argv[] и с окружением envp	*process.h, stdio.h*
spawnvp	int spawnvp(int mode, char *path, char *argv[]) Выполняет в режиме mode порожденный процесс path с аргументами argv[], с поиском в PATH	*process.h, stdio.h*
spawnvpe	int spawnvpe(int mode, char *path, char *argv[], char *envp[]) Выполняет в режиме mode порожденный процесс path с аргументами argv[], с поиском в PATH и с окружением envp	*process.h, stdio.h*

Функция	Синтаксис / Описание	Файл .
system	int system(const char *command) Выполняет команду command операционной системы и возвращается в приложение	*stdlib.h*
_wsystem	int **_wsystem(const** wchar_t ***command)** Выполняет команду command операционной системы и возвращается в приложение	*stdlib.h*
wait	int wait(int *statloc) Обеспечивает ожидание завершения одного или более порожденных процессов, заносит в statloc статус завершения, возвращает ID порожденного процесса или -1	*process.h*
WinExec	**UINT WinExec(** LPCSTR **lpCmdLine,** **UINT uCmdShow)** Выполняет указанное приложение	*winbase.h*

3.6.3 Сообщения об ошибках при запуске внешних програм

Если перечисленные в таблице разд. 3.6.2 функции API Windows **CreateProcess. FindExecutable. ShellExecute. WinExec** и некоторые другие возвращают значение, меньшее или равное 32, это указывает на ошибку. Значения ошибок означают следующее (в Windows 95, 98, 2000 и NT для некоторых из этих ошибок имеются именованные константы):

Значение	Именованная константа	Пояснение
0		Системе не хватает памяти, выполняемый файл испорчен или произошло ошибочное перераспределение памяти.
2	ERROR_FILE_NOT_FOUND	Файл не найден.
3	ERROR_PATH_NOT_FOUND	Путь не найден.
5	SE_ERR_ACCESSDENIED	Была попытка динамически связаться с задачей, была ошибка многопроцессорного выполнения или ошибка защиты сети.
6		Библиотека требует отдельных сегментов данных для каждой задачи.
8	SE_ERR_OOM	Недостаточно памяти для запуска приложения.
10		Ошибочная версия Windows.
11	ERROR_BAD_FORMAT	Ошибочный выполняемый файл. Или это не приложение Windows, или ошибка в .exe файле.
12		Приложение спроектировано для другой операционной системы.

Значение	Именованная константа	Пояснение
13		Приложение спроектировано для MS-DOS 4.0.
14		Неизвестный тип выполняемого файла.
15		Попытка запустить приложение, работающее только на более ранних версиях Windows.
16		Попытка запустить второй экземпляр приложения, содержащего сегменты данных, не помеченные «только для чтения».
19		Попытка запустить архивированный файл. Файл должен быть разархивирован, прежде чем его можно будет загрузить.
20		Ошибочный файл одной из DLL, требуемой для приложения.
21		Приложение требует 32-битного расширения Windows.
26	SE_ERR_SHARE	Нарушение права доступа.
27	SE_ERR_ ASSOCINCOMPLETE	Файл, связанный с указанной операцией не полный или ошибочный.
28	SE_ERR_DDETIMEOUT	Транзакция DDE не может быть выполнена из-за нехватки времени.
29	SE_ERR_DDEFAIL	Транзакция DDE закончилась ошибкой
30	SE_ERR_DDEBUSY	Транзакция DDE не может быть выполнена, поскольку выполняется другая транзакция DDE.
31	SE_ERR_NOASSOC	Нет приложения, связанного с файлом указанного типа, или нет файла, связанного с указанной операцией.
32	SE_ERR_DLLNOTFOUND	Не найдена библиотека DLL.

3.7 Функции различного назначения

3.7.1 Функции динамического распределения памяти

Функция	Синтаксис / Описание	Файл
_msize	size_t **_msize(void** *block) Возвращает размер блока с указателем **block,** выделенного ранее функциями malloc, **calloc,** realloc; только для 32-разрядных приложений	*malloc.h*
_new_handler	typedef void (*pvf)(); pvf _new_handler Указатель на функцию, вызываемую при невозможности выделить память операцией new	*new.h*

Функция	Синтаксис / Описание	Файл
alloca	void *alloca(size_t size) Выделяет пространство размером size в стеке; возвращает указатель на него или NULL	*malloc.h*
AllocMem	void * AllocMem(Cardinal Size) Динамически выделяет область памяти размером Size байтов и возвращает указатель (void *) на выделенную область; эта область в дальнейшем может быть освобождена процедурой **FreeMem**	*SysUtils.hpp*
calloc	void *calloc(size_t nitems, size_t size) Выделяет память под nitems элементов размером size каждый; возвращает указатель на выделенный блок памяти или NULL	*stdlib.h*
free	void **free(void *block)** Освобождает блок памяти block, выделенный ранее функциями calloc, malloc, realloc	*stdlib.h*
GetMemory Manager	void **GetMemoryManager(TMemoryManager** &MemMgr) Возвращает указатель MemMgr на функции пользователя, выделяющие и освобождающие память	*System.hpp*
malloc	void *malloc(size_t size) Выделяет блок памяти размером size; возвращает указатель на этот блок или NULL	*stdlib.h* или *alloc.h*
realloc	void *realloc(void *block, size_t size) Изменяет размер блока block, выделенного ранее функциями malloc, calloc, realloc, на size; возвращает указатель на выделенный блок памяти или NULL	*stdlib.h*
set_new_ handler	typedef void (new * new_handler)(); new_handler set_new_handler(**new_handler** my_handler) Устанавливает функцию **my_handler**, которая будет вызываться при невозможности выделить память операцией new	*new.h*
SetMemory Manager	void **SetMemoryManager(** const TMemoryManager &MemMgr) Устанавливает параметром MemMgr функции пользователя, выделяющие и освобождающие память	*System.hpp*
SysFreeMem	extern PACKAGE int SysFreeMem(void * P) Освобождает память, выделенную под блок с указателем P заказным диспетчером памяти	*System.hpp* или *ShareMem.hpp*

Функция	Синтаксис / Описание	Файл
SysGetMem	extern PACKAGE void * SysGetMem(int Size) Выделяет блок памяти размером Size, если введен заказной диспетчер памяти; возвращает указатель на блок или NULL	*System.hpp или ShareMem.hpp*
SysRealloc Mem	extern PACKAGE void * SysReallocMem(void * P, **int** Size) Изменяет размер блока с указателем P до размера Size, если введен заказной диспетчер памяти; возвращает указатель на блок или NULL	*System.hpp или ShareMem.hpp*
THeapStatus	**System::THeapStatus**GetHeapStatus(void) Заносит информацию о состоянии heap в структуру типа TheapStatus	*System.hpp или ShareMem.hpp*

Комментарии

Для функций, в которых в приведенной таблице указано два заголовочных файла — *System.hpp* или *ShareMem.hpp,* файл *System.hpp* надо подключать, если динамически распределяется глобальная область памяти, а файл *ShareMem.hpp* надо подключать, если динамически распределяется область памяти, которую могут совместно использовать различные процессы.

Для динамического распределения выделяется специальная область памяти — heap. Динамическое распределение памяти в этой области может производиться несколькими способами: с помощью библиотечных функций **malloc, calloc, realloc, free** или с помощью операций **new** и **delete** (см. в гл. 1, в разд. 1.11).

Функция **THeapStatus** заносит информацию о состоянии памяти в структуру типа **THeapStatus,** содержащую поля:

TotalAddrSpace	Общее текущее адресное пространство в байтах, доступное программе. Увеличивается по мере увеличения динамически распределяемой памяти.
TotalUncommitted	Общее число байтов в TotalAddrSpace, которое не выделено для своппируемого файла.
TotalCommitted	Общее число байтов в TotalAddrSpace, которое выделено для своппируемого файла. Справедливо соотношение TotalUncommitted + TotalCommitted = TotalAddrSpace.
TotalAllocated	Объем в байтах динамически выделенной в программе области памяти.
TotalFree	Полное число байтов, доступное для программы. Если это число превышается и доступно достаточно виртуальной памяти, то OS увеличивает доступное адресное пространство. Соответственно увеличивается и TotalAddrSpace.
FreeSmall	Число байтов небольших блоков памяти, которые могут быть еще выделены вашей программе.
FreeBig	Число байтов больших блоков памяти, которые могут быть еще выделены вашей программе. Большие свободные блоки могут создаваться объединением смежных малых свободных блоков или динамическим выделением большого блока.

Unused	Общее число байтов, которые не могут использоваться программой. Справедливо соотношение: Unused + **FreeBig** + **FreeSmall** = **TotalFree.**
Overhead	Число байтов, требуемое диспетчером динамически распределяемой памяти для управления всеми блоками.
HeapErrorCode	Индикатор внутреннего состояния динамически распределяемой памяти.

Поля **TotalAddrSpace, TotalUncommitted и TotalCommitted** относятся к памяти, выделенной для программы системой. А поля **TotalAllocated и TotalFree** относятся к области динамически распределяемой памяти (в дальнейшем для краткости будем называть ее heap). Так что для проверки возможностей динамического выделения памяти надо ориентироваться на **TotalAllocated и TotalFree.**

Функция **SetMemoryManager** позволяет пользователю заменить функции, выделяющие и освобождающие память, своими собственными. Эти функции пользователя задаются полями параметра **MemMgr** типа **TMemoryManage.** Структура типа **TMemoryManage** имеет поля:

GetMem	Указывает на функцию, выделяющую в памяти блок с заданным числом байтов **Size** и возвращающую указатель на выделенный блок (аналог функции **malloc).** Параметр **Size** функции **GetMem** не должен быть равен нулю. Если **GetMem** не может выделить блок заданного размера, она должна возвращать **NULL.**
FreeMem	Указывает на функцию, освобождающую блок памяти, на который указывает ее параметр (аналог функции free). Параметр функции **FreeMem** не должен быть равен **NULL.** Если **FreeMem** успешно освободила память, она должна возвращать 0. В противном случае должно возвращаться ненулевое значение.
ReallocMem	Указывает на функцию, которая изменяет размер блока, на который указавает ее параметр, до заданной новой величины Size (аналог функции **realloc).** Указатель, передаваемый в функцию **ReallocMem,** не должен быть равен **NULL,** а параметр **Size** не должен быть равен 0. Функция **ReallocMem** должна изменить размер блока, при необходимости переместив его на новое место, если нельзя обеспечить требуемый размер на прежнем месте. Информация, хранившаяся в прежнем блоке, должна быть сохранена, но вновь выделяемое пространство может не инициализироваться. Функция должна возвращать указатель на блок или **NULL,** если изменить размер блока невозможно.

Функции пользователя, которые устанавливаются функцией **SetMemoryManager,** могут оперировать с объектами, их конструкторами и деструкторами, строками и т.п. Функция **GetMemoryManager** возвращает структуру типа **TMemoryManager,** содержащую указатели на установленные функции. Через поля этой структуры можно обращаться к установленным функциям.

Приведем пример. Пусть вы хотите вести учет числа обращений к функциям динамического распределения памяти и учет объемов выделяемой и освобождаемой памяти. Это можно сделать следующим кодом:

```
#include <malloc.h>

TMemoryManager*  mmNew;
TMemoryManager*  mmOld;
```

```
long alloc, dealloc, Nalloc, Ndealloc;

...
void * _fastcall NewGetMem(int Size)
{
 alloc += Size;
 Nalloc++;
 return mmOld->GetMem(Size);
}

int _fastcall NewFreeMem(void *p)
{
 dealloc = _msize(p);
 Ndealloc ++;
 return mmOld->FreeMem(p);
}

void * _fastcall NewReallocMem(void *p, int Size)
 (
 alloc += Size;
 dealloc = _msize(p);
 Nalloc++;
 Ndealloc ++;
 return mmOld->ReallocMem(p, Size);
}

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
  mmNew = new TMemoryManager();
  mmOld = new TMemoryManager();
  mmNew->GetMem = NewGetMem;
  mmNew->FreeMem = NewFreeMem;
  mmNew->ReallocMem = NewReallocMem;
  GetMemoryManager(*mmOld);
  SetMemoryManager(*mmNew);
}

void__fastcall TForm1::Button1Click(TObject *Sender)
{
 Label1->Caption = "Nalloc = "+IntToStr(Nalloc)+
                   "Ndealloc = "+IntToStr(Ndealloc)+
                   " выделено " +IntToStr(alloc)
                   " освобождено "  +IntToStr(dealloc);
}
```

В этом примере при щелчке на кнопке **Button1** в метке **Label1** отображается сообщение о динамическом распределении памяти.

3.7.2 Функции вызова диалоговых окон с сообщениями

Функция	Синтаксис / Описание	Файл
CreateMessageDialog	**Forms::TForm* CreateMessageDialog(const AnsiString Msg, TMsgDlgType DlgType, TMsgDlgButtons Buttons)** **Создает (но не отображает) диалоговое окно типа DlgType, с кнопками Buttons, с сообщением Msg**	*Dialogs.hpp*

Функция	Синтаксис / Описание	Файл
InputBox	**AnsiString InputBox(constAnsiString ACaption, const AnsiString APrompt, const AnsiString ADefault)** Предлагает пользователю диалоговое окно с заголовком **ACaption,** с предложением **APrompt** пользователю что-то написать и с окошком редактирования, в котором предварительно загружено начальное значение текста **ADefault**	*Dialogs.hpp*
InputQuery	**bool InputQuery(constAnsiString ACaption, const AnsiString APrompt, AnsiString &Value)** Предлагает пользователю диалоговое окно с заголовком **ACaption,** с предложением **APrompt** пользователю что-то написать и с окошком редактирования, в котором предварительно загружено начальное значение текста **ADefault.** Возвращаемое значение показывает, нажал ли пользователь кнопку OK	
LoginDialog	**bool LoginDialog(constAnsiString ADatabaseName, AnsiString &AUserName, AnsiString &APassword)** Вызывает стандартное окно Windows (нерусифицированное), содержащее запрос имени и пароля пользователя для доступа к базам данных. Параметр **ADatabaseName** является строкой, содержащей имя базы данных, параметр **AUserName** — строка, содержащая имя пользователя, параметр **APassword** — строка, содержащая введенный пароль	*DBLogDlg.hpp*
LoginDialogEx	**bool LoginDialogEx(const AnsiString ADatabaseName, AnsiString &AUserName, AnsiString &APassword, bool NameReadOnly)** То же, что **LoginDialog,** но имеет параметр **NameReadOnly,** который можно задать равным **true,** чтобы запретить изменения имени пользователя в диалоге	*DBLogDlg.hpp*
MessageDlg	**int MessageDlg(constAnsiString Msg, TMsgDlgType DlgType, TMsgDlgButtons Buttons, int HelpCtx)** отображает диалоговое окно типа **DlgType** с кнопками **Buttons,** с темой справки **HelpCtx,** с сообщением Msg	*Dialogs.hpp*

Функция	Синтаксис / Описание	Файл
MessageDlgPos	int MessageDlgPos(constAnsiString Msg, TMsgDlgType DlgType, TMsgDlgButtons Buttons, int HelpCtx, int X, int Y) То же, что MessageDlg, но с заданной позицией окна (X, Y)	*Dialogs.hpp*
SelectDirectory	bool SelectDirectory(const AnsiString Caption, const WideString Root, AnsiString &Directory) bool SelectDirectory(AnsiString &Directory, TSelectDirOpts Options, int HelpCtx) Предоставляет пользователю возможность выбрать каталог с помощью стандартного диалога	*FileCtrl.hpp*
ShowMessage	void ShowMessage(const System::AnsiString Msg) Отображает диалоговое окно с сообщением Msg	*Dialogs.hpp*
ShowMessage Fmt	void ShowMessageFmt(constAnsiString Msg, const System::TVarRec Params, const int Params_Size) Отображает диалоговое окно с сообщением по формату Msg, применяемому к параметрам Params	*Dialogs.hpp*
ShowMessage Fmt	void ShowMessagePos(constAnsiString Msg, int X, int Y) Отображает диалоговое окно с сообщением Msg в позиции, левый верний угол которой задается координатами X и Y	*Dialogs.hpp*

Комментарии

Помимо перечисленных функций см. также метод **MessageBox** компонента **Application,** отображающий, пожалуй, наиболее удачный вариант диалогового окна.

Функции **LoginDialog и LoginDialogEx** вызывают стандартные окна Windows, содержащие запрос имени и пароля пользователя для доступа к базам данных. Параметр **ADatabaseName** является строкой, содержащей имя базы данных, с которой требуется соединиться. Параметр **AUserName** -- строка, содержащая имя пользователя. При вызове функции **LoginDialog** в этот параметр можно записать имя пользователя по умолчанию, и оно будет отображаться в диалоге в окошке User Name. А по окончании диалога в параметре AUserName содержится имя, указанное пользователем, или имя по умолчанию, если пользователь его не изменял. Параметр **APassword** — строка, содержащая пароль, введенный пользователем в процессе диалога.

Функция **LoginDialog** возвращает **true,** если пользователь завершил диалог, щелкнув в нем на кнопке OK. В этом случае программа может пытаться открыть базу данных с указанными именем и паролем пользователя или предварительно проверить, зарегистрирован ли такой пользователь, правильный ли указан пароль и какой уровень доступа разрешен этому пользователю. Если пользователь прервал диалог, то возвращается **false.**

В диалоговом окне, отображаемом функцией **LoginDialog,** пользователь может изменять имя, указываемое в окошке User Name. Другая функция Login-

DialogEx — позволяет запретить изменение имени. Для этого надо положить равным **true** параметр **NameReadOnly.**

Существенным недостатком функций **LoginDialog и LoginDialogEx** являются нерусифицированные диалоговые окна. Так что в большинстве приложений вряд ли стоит использовать эти функции. Окна, отображаемые этими функциями, легко создать самому, причем с русскими надписями.

3.7.3 Функции воспроизведения звуков

Функция	Синтаксис / Описание	Файл
Beep	extern PACKAGE void Beep(void) Функция C++Builder, воспроизводит стандартный звуковой сигнал	*SysUtils.hpp*
Beep	BOOL Beep(DWORD dwFreq, DWORD **dwDuration);** Функция API Windows, только для Windows NT/2000/XP, воспроизводит звуковой сигнал с частотой dwFreq Герц и длительностью dwDuration миллисекунд	–
MessageBeep	BOOL MessageBeep(UINT uType); Функция API Windows, воспроизводит звуковой сигнал типа uType	–
PlaySound	BOOL PlaySound(LPCSTR **pszSound,** HMODULE **hmod,** DWORD fdwSound) Функция API Windows, воспроизводит звук указанного волнового файла, или звука системного события, или звука из ресурса	*mmsystem.hpp*

Комментарии

Функция C++Builder **Beep** воспроизводит стандартный звуковой сигнал, вызывая функцию **MessageBeep** API Windows с нулевым параметром. При этом воспроизводится стандартный звуковой сигнал, установленный в Windows, если компьютер имеет звуковую карту и стандартный сигнал задан (он устанавливается в «Панели управления» после щелчка на пиктограмме Звук). Если звуковой карты нет или стандартный сигнал не установлен, звук воспроизводится через динамик компьютера.

Воспроизведение асинхронное, т.е. приложение продолжает выполняться во время воспроизведения звука.

Функция **Beep** API Windows, примененная в Windows NT/2000/XP, синхронно воспроизводит звук простого тона через динамик и не возвращается до окончания звука. В Windows NT параметр **dwFreq** задает частоту звука в герцах. Он может иметь значения в диапазоне от 37 до 32,767 (от 0x25 до 0x7FFF). Параметр **dwDuration** устанавливает длительность звука в миллисекундах.

Воспроизведение синхронное: функция не возвращается до окончания воспроизведения звука.

Все сказанное относится только к Windows NT/2000/XP. В Windows 95 и 98 параметры игнорируются и функция становится подобной функции **Beep** C++Builder. Отличие этих функций остается только в том, что **Beep** C++Builder ничего не возвращает, а **Beep** API Windows при успешном выполнении возвращает ненулевое значение. При аварийном завершении она возвращает нуль. Тогда более развернутую информацию об ошибке можно получить вызовом функции **GetLastError.**

Компилятор автоматически разбирается, какая именно из функций **Beep** использована в программе, по наличию или отсутствию параметров.

Функция **MessageBeep** воспроизводит звуковой сигнал указанного типа. Звуки, соответствующие различным типам сигналов, хранятся в реестре в разделе [sounds] и устанавливаются пользователем с помощью программы «Панель управления» щелчком на пиктограмме Звук.

Целый без знака параметр **uType** функции **MessageBeep** определяет воспроизводимый звук. Для него предопределены следующие константы:

Значение	Звук
OxFFFFFFFF	**Стандартный звук через динамик**
MB_ICONASTERISK	**Звездочка**
MB_ICONEXCLAMATION	**Восклицание**
MB_ICONHAND	**Критическая ошибка**
MB_ICONQUESTION	**Вопрос**
MB_OK	**Стандартный звук**

При успешном завершении функция возвращает ненулевое значение (**true**). Если функция вернула нулевое значение, то получить информацию об ошибке можно с помощью вызова **GetLastError.**

После инициализации воспроизведения звука функция **MessageBeep** возвращает управление в точку вызова и воспроизведение звука производится асинхронно.

Если функция **MessageBeep** не нашла указанный тип звука, она пытается воспроизвести стандартный звук. Если и он не установлен или если компьютер не снабжен звуковой картой, то звук воспроизводится через динамик компьютера.

Функция **PlaySound** API Windows воспроизводит звук указанного волнового файла, или звука системного события, или звука из ресурса.

Параметр **pszSound** представляет собой строку с нулевым символом в конце и определяет воспроизводимый звук. В зависимости от значений флага **fdwSound (SND_FILENAME, SND_ALIAS** или **SND_RESOURCE)** параметр **pszSound** может определять имя волнового файла, псевдоним системного события или идентификатор ресурса. Если ни один из этих флагов не указан, функция ищет в реестре Windows или в файле WIN.INI указанное имя звука. Если звук найден, то он воспроизводится. Если звук не найден, то параметр **pszSound** интерпретируется как имя файла.

Звук, указанный параметром **pszSound,** должен помещаться в доступную память и должен подходить для установленного драйвера устройства воспроизведения волновых файлов. Функция **PlaySound** ищет файл звука в следующих каталогах: текущем, каталоге Windows, системном каталоге Windows, каталогах, перечисленных в переменной среды PATH, в списке каталогов, предоставляемых сетью. Более подробно последовательность поиска в каталогах рассмотрена в документации по функции **OpenFile.**

Если указанный звук не находится, функция **PlaySound** воспроизводит системный звук по умолчанию. Если функция не может найти и его, то воспроизведения не будет, а вернется значение **false.**

Если параметр **pszSound** задан равным 0, то воспроизведение любого волнового файла прерывается. Для прерывания воспроизведения звука, не связанного с волновым файлом, надо указывать **SND_PURGE** в параметре **fdwSound.**

Параметр **hmod** используется только при параметре **fdwSound** равном **SND_RESOURCE.** В этом случае hmod является дескриптором выполняемого фай-

ла, содержащего ресурс, который должен загружаться. В противном случае значение **hmod** задается равным 0.

Параметр **fdwSound** задает флаги воспроизведения звука. Флаги могут комбинироваться друг с другом операцией ИЛИ "|". Возможны следующие значения флагов:

SND_ALIAS	Параметр **pszSound** определяет псевдоним системного события в реестре Windows или в файле WIN.INI. Нельзя использовать совместно с SND_FILENAME и SND_ RESOURCE.
SND_ALIAS_ID	Параметр **szSound** является предопределенным идентификатором звука.
SND_APPLICATION	Звук воспроизводится с использованием установок приложения.
SND_ASYNC	Звук воспроизводится асинхронно и функция **PlaySound** возвращается немедленно после начала воспроизведения. Чтобы прекратить асинхронное воспроизведение волнового файла, надо вызвать **PlaySound** с параметром pszSound, равным 0.
SND_FILENAME	Параметр pszSound является именем файла.
SND_LOOP	Воспроизведение звука постоянно повторяется, пока не вызовется PlaySound с параметром pszSound, равным 0. Одновременно надо указать флаг SND_ASYNC асинхронного воспроизведения звука.
SND_MEMORY	Файл звука события загружен в память. В этом случае параметр pszSound должен указывать на образ звука в памяти.
SND_NODEFAULT	Звук события, кроме звука по умолчанию. Если указанный звук не найден, PlaySound вернется, не воспроизводя звук по умолчанию.
SND_NOSTOP	Если заданный звук не может быть воспроизведен, поскольку ресурсы, необходимые для воспроизведения, заняты воспроизведением другого звука, функция PlaySound немедленно вернет false, не воспроизводя заданного звука. Если данный флаг не указан, функция PlaySound пытается остановить воспроизведение другого звука, чтобы устройство могло быть использовано для воспроизведения нового звука.
SND_NOWAIT	Если драйвер занят, функция сразу вернется без воспроизведения заданного звука.
SND_PURGE	Останавливается воспроизведение любых звуков, вызванных в данной задаче. Если pszSound не 0, останавливаются все экземпляры указанного звука. Если pszSound равен 0, то останавливаются все звуки, связанные с данной задачей. Отдельно надо указать дескриптор для остановки событий SND_RESOURCE.
SND_RESOUECE	Параметр pszSound является идентификатором ресурса. Параметр hmod должен указывать на источник ресурса.
SND_SYNC	Синхронное воспроизведение звука события. Функция PlaySound возвращается только после окончания воспроизведения.

Функция **PlaySound** при успешном выполнении возвращает **true,** в противном случае — **false.**

3.7.4 Некоторые вспомогательные функции C++ и C++Builder

Функция	Синтаксис / Описание	Файл
ARRAYSIZE	**ARRAYSIZE(const void *a)** Макрос возвращает число элементов массива a	*sysdefs.h*
bsearch	**void *bsearch(const void *key, const void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp) (const void *, const void *))** Выполняет двоичный поиск по ключу key в массиве (таблице) **base** из **nelem** элементов по **width** байт каждый с помощью функции **fcmp;** возвращает адрес элемента или О	*stdlib.h*
EXISTING ARRAY	**EXISTINGARRAY(const void *a)** Макрос возвращает индекс последнего элемента массива a	*sysdefs.h*
getenv	**char *getenv(const char *name)** Возвращает или удаляет переменную окружения **name**	*stdlib.h*
GetLongHint	**extern PACKAGE System::AnsiString GetLongHint(const System::AnsiString Hint)** Возвращает вторую часть строки формата, используемого в свойствах компонентов **Hint**	*Controls.hpp*
GetShort Hint	**extern PACKAGE System::AnsiString GetShortHint(const System::AnsiString Hint)** Возвращает первую часть строки формата, используемого в свойствах компонентов **Hint**	*Controls.hpp*
lfind	**void *lfind(const void *key, const void *base, size_t *num, size_t width, int (_USERENTRY *fcmp) (const void *, const void *))** Выполняет линейный поиск по ключу key в массиве (таблице) **base** из **num** записей по **width** байт в каждый с помощью функции **fcmp;** возвращает адрес элемента или О	*stdlib.h*
lsearch	**void *lsearch(const void *key, void *base, size_t *num, size_t width, int(_USERENTRY *fcmp) (const void *, const void *))** Выполняет линейный поиск по ключу key в массиве (таблице) **base** из **num** записей по **width** байт в каждый с помощью функции **fcmp;** если элемент не найден, он добавляется в таблицу; возвращает адрес элемента	*stdlib.h*

Функция	Синтаксис / Описание	Файл
OPENARRAY	OPENARRAY(type **arg1, ..., type arg19)** Макрос обеспечивает передачу в функцию открытого массива, содержащего до 19 элементов	*sysdefs.h*
ParamCount	extern PACKAGE int__fastcall ParamCount(void); Возвращает число параметров командной строки	*System.hpp*
ParamStr	extern PACKAGE AnsiString__fastcall **ParamStr(int** Index); Возвращает параметр с индексом Index командной строки	*System.hpp*
putenv	int putenv(const **char** *name) Устанавливает переменную окружения name	*stdlib.h*
qsort	void **qsort(void** *base, size t **nelem,** size t width, int (_USERENTRY *fcmp) (const void *, const void *)) Выполняет быструю сортировку в массиве (таблице) base из nelem элементов по width байт каждый с помощью функции fcmp	*stdlib.h*
Shortcut	extern PACKAGE **TShortCut ShortCut(Word** Key, **Classes::TShiftState** Shift) Создает структуру, используемую для задания комбинации «горячих» клавиш Key и Shift разделу меню	*Menus.hpp*
Shortcut ToText	extern PACKAGE System::AnsiString ShortCutToText(TShortCut Shortcut) Преобразует структуру Shortcut, содержащую комбинацию «горячих» клавиш раздела меню, в строку текста	*Menus.hpp*
swab	void swab(char ***from,** char *to, int nbytes) Копирует nbytes байтов строки **from** в строку to, меняя местами каждую пару смежных байтов	*stdlib.h*
TextTo Shortcut	extern PACKAGE TShortCut **TextToShortCut(** System::AnsiString Text) Создает из строки текста Text структуру, используемую для задания комбинации «горячих» клавиш разделу меню	*Menus.hpp*
va_arg	type va_arg(va_list ap, type) Макрос возвращает текущий аргумент списка переменной длины типа type и переводит указатель ap на следующий аргумент; предварительно указатель должен быть установлен с помощью va_start или va_arg	*stdarg.h*
va_end	void va_end(va_list ap) Макрос обеспечивает завершение передачи в функцию списка аргументов произвольной длины, обработанного макросами va_start и va_arg	*stdarg.h*

Функция	Синтаксис / Описание	Файл
va_start	void va_start(va_list ap, lastfix) Макрос устанавливает ap на первую переменную, передаваемую в функции, использующие списки аргументов произвольной длины; lastfix — последний переданный в функцию обязательный аргумент	*stdarg.h*

Комментарии

Макросы **EXISTINGARRAY, ARRAYSIZE, OPENARRAY** используются при передаче в функции массивов. Описание способов работы с этими макросами см. в разд. 2.11.3. Примеры использования макросов приведены также в описании функции **Format в гл. 4.**

Функции **bsearch, lfind, lsearch, qsort** предназначены для поиска и сортировки в массивах (таблицах). Во всех этих функциях параметр **base** указывает на начало массива, параметр **nelem** или **num** определяет число элементов, параметр **width** определяет число байтов, занимаемых элементом, а параметр **fcmp** указывает на функцию сравнения, которую вы должны определить и которая сигнализирует о результатах сравнения двух элементов, заданных своими указателями.

Функция **bsearch** осуществляет двоичный поиск (дихотомию) элемента, соответствующего ключу key. Подобный поиск самый быстрый, но он требует, чтобы элементы массива были расположены в порядке возрастания критерия поиска. Функция сравнения **fcmp** в данном случае получает как параметры два указателя: ***elem1 и *elem2.** Функция должна провести сравнение значений, на которые они указывают, и вернуть результат сравнения:

< 0	при *eleml < *elem2
== 0	при *eleml == *elem2
> 0	при *eleml > *elem2

Здесь знак < означает, что элемент ***elem1** расположен в массиве раньше элемента ***elem2,** знак > означает, что элемент ***eleml** расположен в массиве после элемента ***elem2,** а знак равенства означает, что элементы равны. Эта оговорка существенна, поскольку элементами могут быть не только числа, но и объекты любого типа, например, записи со множеством полей.

Функция **qsort** выполняет быструю сортировку массива по критерию, заданному функцией сравнения. Сама функция сравнения используется такая же, как описанная выше.

Ниже приведен пример использования функций **qsort и bsearch.** В примере объявлен массив **array** неупорядоченных целых чисел. Функция **fcmp** — это функция сравнения, одинаковая для **qsort и bsearch.** При щелчке на кнопке **Buttonl** массив сначала упорядочивается функцией **qsort,** а затем в нем ищется с помощью **bsearch** элемент, соответствующий указанному пользователем в окне **Edit1.**

```
#include <malloc.h>
#include <stdlib.h>

int array[] = {800,123,512,627,933,145};
...

int fcmp (const void *p1, const void *p2)
{ return (*(int*)p1 - *(int*)p2); }

void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
 int key = StrToInt(Edit1->Text);
 int *elem;
 qsort(array, ARRAYSIZE(array),sizeof(int), fcmp);
 elem = (int *) bsearch (&key, array, ARRAYSIZE(array),
                         sizeof(int), fcmp);
 if(elem == 0) ShowMessage("Элемент " + IntToStr(key) +
                           " не найден");
 else ShowMessage("Индекс элемента " + IntToStr(*elem) +
                  " равен " + IntToStr(elem- array));
}
```

Функции **lfind и lsearch** выполняют в массиве линейный поиск. Он медленнее, чем дихотомия, но может применяться к неупорядоченным массивам. Функции различаются тем, что **Isearch,** если элемент не обнаружен, добавляет его в конец массива. Функции поиска в **Ifind и Isearch** отличаются от рассмотренных выше. Они должны возвращать нуль при совпадении элементов и ненулевое значение, если элементы различны.

Ниже приведен пример использования функции **Isearch,** похожий на рассмотренный ранее. В примере объявлен массив **array** неупорядоченных целых чисел. Функция **fcmp1** — это функция сравнения. При щелчке на кнопке **Button1** в массиве ищется элемент, соответствующий указанному пользователем в окне **Edit1**. Если элемент не найден, он добавляется в конец массива.

```
#include <malloc.h>
#include <stdlib.h>
int array[10] = {800,123,512,627,933,145};
unsigned Narray = 6;
...

int fcmp1 (const void *p1, const void *p2)
{ return(*(int*)p1 != *(int*)p2) ; }

void _fastcall TForm1 : :Button1Click(TObject *Sender)
{
 int key = StrToInt(Edit1->Text);
 int *elem;
elem = (int *) Isearch (&key, array, &Narray,
                        sizeof(int), fcmp1);
 ShowMessage("Индекс элемента " + IntToStr(key) +
             " равен " +IntToStr(elem-array));
}
```

Функции **GetShortHint и GetLongHint** возвращают соответственно первую и вторую части строки формата

```
<текст первой части>|<текст второй части>
```

Строки такого вида, в частности, задаются в свойстве компонентов **Hint.**

Функции **getenv и putenv** позволяют работать с переменными окружения. Переменные окружения представляют собой строки таблицы параметров окружения в виде **name=string\0.** Функция **getenv** ищет или удаляет указанную переменную окружения name. Если в функции **getenv** задать имя переменной окружения, она вернет указатель на строку, содержащую значение этой переменной. Например, оператор

```
Label1->Caption = getenv("PATH");
```

отображает в метке **Labell** содержание строки переменной PATH.

Имена переменных DOS и OS/2 должны записываться в верхнем регистре. Остальные переменные могут записываться как в верхнем, так и в нижнем регистрах.

Если переменная окружения с этим именем отсутствует, то возвращается **NULL.** Если в функции **getenv** задать параметр **name** в виде "name=", то переменная name будет удалена из окружения. Например, оператор

```
getenv("PATH=");
```

очистит переменную PATH.

Функция **putenv** устанавливает переменную окружения. Параметр **name** задается в виде " **name=string**". Например:

```
putenv("PATH=c:\\temp");
```

Функции **Shortcut, ShortCutToText и TextToShortCut** используются для задания свойства **Shortcut** раздела меню, определяющего соответствующую этому разделу комбинацию «горячих» клавиш. Функция **Shortcut** упаковывает параметр Key, определяющий виртуальный код клавиши, и параметр **Shift,** задающий комбинацию вспомогательных клавиш типа Shift, Ctrl и Alt, в значение типа **TShortCut,** эквивалентное типу **Word.** Функция **TextToShortCut** создает аналогичное значение из строки текста. Функция **Shortcut** выполняется быстрее, но зато функцию **TextToShortCut** удобнее использовать в диалоге, когда комбинацию «горячих» клавиш задает пользователь с помощью окна редактирования.

Функция **ShortCutToText** позволяет получить текстовое описание значения **Shortcut** типа **TShortCut.** Эту функцию удобно использовать для вывода пользователю принятой в разделе меню комбинации «горячих» клавиш, если ему предоставляется возможность изменять эту комбинацию.

Рассмотрим примеры использования этих трех функций. Оператор

```
MOpen->ShortCut = ShortCut('O', TShiftState() << ssCtrl);
```

задает разделу меню с именем **MOpen** «горячие» клавиши Ctrl-O. Оператор

```
MOpen->ShortCut = ShortCut('O',TShiftState()<<ssCtrl<<ssAlt);
```

задает тому же разделу комбинацию Ctrl-Alt-O.

Еще один пример. Приведенные ниже процедуры обеспечивают задание пользователем комбинации «горячих» клавиш для раздела меню, названного в программе **Open.** Первая процедура с помощью **ShortCutToText** задает начальное значение текста в окне редактирования, равное исходной комбинации клавиш. А вторая — с помощью функции **TextToShortCut** изменяет комбинацию на заданную пользователем. Пользователь может задать, например, комбинацию Ctrl-O или как "^O", или как "Ctrl+O".

```
void__fastcall TForm1::Button1Click(TObject *Sender)
{
 Edit1->Text = ShortCutToText (MOpen->ShortCut);
}
//---------------------------------------------------------------
void__fastcall TForm1::Button2Click(TObject *Sender)
{
 MOpen->ShortCut = TextToShortCut(Edit1->Text);
}
```

Функции **ParamStr и ParamCount** позволяют работать с командной строкой. Функции **ParamStr** возвращает параметр командной строки с указанным индексом **Index.** Нулевым параметром командной строки является имя выполняемого файла приложения вместе с полным путем к нему. Таким образом, выражение **ParamStr(O)** вернет, например, строку "D:\\TEST\\PROJECT1.EXE", т.е. имя файла, приведенное к верхнему регистру. Из этого имени можно извлечь путь к выполняемому файлу. Это очень часто требуется, если программа использует какие-то другие файлы, расположенные в том же каталоге, в котором располагается выполняемый файл.

Если при запуске приложения в него через командную строку переданы какие-то параметры, то эти параметры могут быть прочитаны соответственно выра-

жениями **ParamStr(1), ParamStr(2)** и т.п. При этом регистр параметров будет тем, который использован при запуске программы. Так что при чтении параметров желательно программно приводить их к верхнему или нижнему регистрам.

Функция **ParamCount** возвращает число параметров, переданных через командную строку. Она позволяет организовывать циклы по параметрам командной строки. Например, код

```
for (int i=1;i<=ParamCount();i++)
    if (LowerCase(ParamStr(i)) == "-e")
      ...
```

обеспечивает выполнение некоторых действий (обозначены многоточием), если среди параметров командной строки встретится "-e" или "-E".

Следует отметить, что в файле объявлена переменная **CmdLine** типа **(char** *). Эта переменная содержит полный текст командной строки, в котором параметры отделены друг от друга пробелами. Регистр всех параметров в строке **CmdLine,** включая нулевой, соответствует тому, который использовался при запуске приложения на выполнение. Еще один альтернативный способ работы с командной строкой рассмотрен в гл. 1, в разд. 1.2.2.

3.7.5 Некоторые вспомогательные функции API Windows

Функция	Синтаксис / Описание
CloseWindow	**BOOL CloseWindow(HWND hWnd)** **Сворачивает, не уничтожая, окно, указанное дескриптором hWnd**
DestroyWindow	**BOOL DestroyWindow(HWND hWnd)** **Уничтожает окно, указанное дескриптором hWnd, и всех его потомков, освобождает отведенную память**
EnableWindow	**BOOL EnableWindow(HWND hWnd, BOOL bEnable)** **Делает доступным (при bEnable = true) или недоступным (при bEnable = false) окно, указанное дескриптором hWnd**
FindWindow	**HWND FindWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName)** **Возвращает дескриптор окна класса IpClassName с заголовком IpWindowName**
GetLastError	**DWORD GetLastError(VOID)** **Возвращает код последней ошибки**
GetNextWindow	**HWND GetNextWindow(HWND hWnd, UINT wCmd)** **Возвращает дескриптор следующего за hWnd или предыдущего окна в Z-последовательности**
GetWindow	**HWND GetWindow(HWND hWnd, UINT wCmd)** **Возвращает дескриптор окна, находящегося с указанным окном hWnd в указанном соотношении wCmd**
GetWindowText	**int GetWindowText(HWND hWnd, LPTSTR lpString, int nMaxCount)** **Копирует текст, связанный с окном или оконным элементом hWnd, в буфер IpString размера nMaxCount**

3.8 Работа с сообщениями Windows

Работа с сообщениями Windows рассмотрена в гл. 1, в разд. 1.14. Ниже приводятся справочные сведения по функциям, которые использовались в гл. 1.

Функция	Синтаксис / Описание	Файл
PostMessage	**BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)** Помещает указанное в ней сообщение окну в очередь сообщений потока, и возвращается, не дожидаясь окончания обработки этого сообщения	*winuser.h*
RegisterWindowMessage	**UINT RegisterWindowMessage(LPCTSTR lpString)** Определяет новое окно сообщения с гарантированной уникальностью его в системе, которое может использоваться в функциях **SendMessage и PostMessage**	*winuser.h*
SendMessage	**LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)** Посылает указанное в ней сообщение окну и не возвращается, пока это сообщение обрабатывается	*winuser.h*

Глава 4

Описания функций

В гл. 3 были приведены краткие описания функций C, C++, C++Builder и API Windows. В данной главе дается описание тех из них (более 300), которые наиболее часто используются или которые требуют развернутых пояснений и примеров. Полное описание всех функций невозможно в рамках данной книги. Может быть, в недалеком будущем мною будет подготовлен ряд относительно небольших книг, содержащих полное описание стандартных библиотек функций C, C++, C++Builder и API Windows. Отмечу также, что значительно большее число описаний функций имеется в [2].

Ряд типов данных в этой главе помечен в тексте подчеркиванием. Например, **TDateTime.** Это означает, что подробные описания этих типов вы можете найти в соответствующих главах книги. Аналогично подчеркиванием выделены функции, подробно рассмотренные в данной главе.

abort — функция завершения выполнения

Завершает выполнение приложения в случае ошибки

Синтаксис

```
#include <stdlib.h>
void abort(void);
```

Описание

Функция **abort** вызывает завершение выполнения приложения, свидетельствуя о появившейся ошибке времени выполнения. В родительский процесс или в операционную систему возвращается код завершения 3.

При вызове функции **abort** она, в свою очередь, вызывает **raise(SIGABRT),** генерируя тем самым сигнал **SIGABRT** (см. разд. «signal и raise — функции работы с сигналами», а также гл. 1, разд. 1.13. Если в приложении нет обработчика этого сигнала, то функция **abort** пишет **в stderr** (см. разд. 2.10.2.4) сообщение «Abnormal program termination» и затем завершает работу приложения вызовом функции **exit** с кодом 3.

Abort — функция генерации исключения

Генерирует исключение **EAbort**

Заголовочный файл *SysUtils.hpp*

Объявление

```
extern PACKAGE void __fastcall Abort(void);
```

Описание

Функция **Abort** применяется для прерывания вычислений в процедурах и функциях (особенно, глубоко вложенных) без сообщения об ошибке. Процедура генерирует специальное «молчаливое исключение» **EAbort,** срабатывающее как любое другое исключение, но не вызывающее сообщения об ошибке. Функция **Abort** прерывает текущую процедуру и все вызвавшие ее процедуры, передавая управление на самый верх — в конец последнего из блоков **try ... finally.** Таким образом, это наиболее простой выход из глубоко вложенных процедур. Впрочем, можно при необходимости перехватить исключение на каком-то промежуточном

уровне, предусмотрев на нем блок try ... catch и вставив соответствующий оператор обработки:

```
catch(EAbort&)
{
 ...
}
```

abs и другие функции вычисления модуля

Функции вычисления модуля

Заголовочные файлы *math.h, stdlib.h*

Синтаксис

```
#include <stdlib.h>
int abs(int x);
long labs(long int x);

tinclude <math.h>
double fabs(double x);
long double fabsl(long double x);
```

Описание

Функции возвращают значение модуля аргумента соответствующего типа. Функции **fabs** и **fabsl** принимают и возвращают действительные значения. Функции **abs** и **labs** принимают и возвращают целые значения. Если abs вызывается из приложения, в которое подключен файл *stdlib.h,* то она развертывается как макрос. Если это нежелательно и вы хотите использовать именно функцию abs, надо после директивы

```
#include <stdlib.h>
```

включить директиву

```
#undef abs
```

Функция abs возвращает значение в диапазоне от 0 до INT_MAX (описана в файле *limit.h*).

AnsiCompareStr и другие функции сравнения строк

Сравнивают две строки

Заголовочный файл *SysUtils.hpp*

Объявления

```
extern PACKAGE int__fastcall
    AnsiCompareStr(const AnsiString S1, const AnsiString S2);
extern PACKAGE int__fastcall
    AnsiCompareText(const AnsiString S1, const AnsiString S2);
```

Описание

Функции сравнивают две строки **S1** и S2 типа AnsiString. Возвращают значение < 0, если **S1** < S2, 0, если **S1** = S2, и > 0, если **S1** > S2.

Сравнение строк осуществляется по символам, начиная с первого. Если очередные символы не равны друг другу, то строка, в которой символ больше, считается больше другой строки, и функция возвращает соответствующее значение. Сравнение символов кириллицы производится в соответствии с русским алфавитом. Считается, что латинские символы меньше символов кириллицы, символы цифр меньше символов букв, символы пунктуации (включая пробел) меньше символов цифр.

Если в процессе сравнения оказывается, что в одной строке символы закончились, а в другой еще имеются, строка с меньшим числом символов считается меньшей.

Функция **AnsiCompareText** проводит сравнение, не учитывая регистр, в котором набраны символы. Т.е. символы "а" и "А" считаются равными. Причем, в отличие от функции **CompareText.** это распространяется на символы кириллицы.

Функция **AnsiCompareStr** учитывает регистр, в котором набраны символы. Заглавная буква (символ, набранный в верхнем регистре) считается больше аналогичной прописной (набранной в нижнем регистре). Так что строка "а" считается меньше строки "А".

Примеры

Следующие операторы обеспечивают различные действия (обозначены многоточиями) в зависимости от сравнения текстов в окнах **Edit1** и **Edit2.**

```
if (AnsiCompareStr (Edit1->Text, Edit2->Text) < 0)
  ...
else if (AnsiCompareStr (Editl->Text, Edit2->Text) == 0)
  ...
else ...
```

Аналогичный оператор можно записать и для функции **AnsiCompareText.**

В приведенной ниже таблице даны различные варианты строк S1 и S2 и результаты их сравнения

Editl->Text	Edit2->Text	AnsiCompareStr	AnsiCompareText
строка 1	строка 2	<0	<0
Строка	Строка	0	0
Строка	строка	> 0	0
Edit	edit	> 0	0
Строка	СтрокИ	< 0	< 0

AnsiCompareText — сравнение строк без учета регистра

Сравнивает две строки без учета регистра.

См. разд. «AnsiCompareStr и другие функции сравнения строк».

AnsiLowerCase и другие функции преобразования строки к нижнему регистру

Преобразуют строку к нижнему регистру.

Заголовочные файлы *SysUtils.hpp, string.h, mbstring.h*

Синтаксис

```
extern PACKAGE AnsiString___fastcall
                          AnsiLowerCase(const AnsiString S);
extern PACKAGE char *___fastcall
                          AnsiStrLower(char * Str);
extern PACKAGE AnsiString___fastcall
                          LowerCase(const AnsiString S);
extern PACKAGE char *___fastcall StrLower(char * Str);

char *strlwr(char *s);
wchar_t * wcslwr(wchar_t *s);
unsigned char *_mbslwr(unsigned char *s);
```

Описание

Функции возвращают строку Str или s типа **(char** *) или строку **S** типа **AnsiString,** преобразованную к нижнему регистру. Функции **StrLower, LowerCase, strlwr, _wcslwr, _mbslwr** работают только с латинскими символами и непри-

менимы к символам кириллицы. Для русских текстов должны использоваться функции **AnsiStrLower и AnsiLowerCase.** Функция **AnsiLowerCase** применима также к многобайтным символам.

При работе с компонентами VCL обычно удобнее использовать функции **AnsiLowerCase и LowerCase,** работающие со строками типа **AnsiString.**

Примеры

Пусть в окне редактирования **Edit1** записан текст: "Hi! Привет!". Ниже приведена таблица, в которой показаны результаты, возвращаемые различными функциями.

Выражение	Результат
AnsiLowerCase(Edit1->Text)	hi! привет!
LowerCase(Edit1->Text)	hi! Привет!
AnsiStrLower((Edit1->Text).c_str())	hi! привет!
StrLower((Edit1->Text).c_str())	hi! Привет!

Вы можете видеть, что функции **AnsiLowerCase, AnsiStrLower** работают нормально, а функции **LowerCase, StrLower** не преобразует русский текст. Можно также видеть, что при работе со свойствами компонентов VCL функции **AnsiLowerCase, LowerCase** удобнее, поскольку не требуют приведения типов.

См. также примеры в разд. «AnsiPos и другие функции поиска подстроки».

AnsiPos и другие функции поиска подстроки

Возвращают позицию первого вхождения заданной подстроки в строку.

Заголовочный файл *SysUtils.hpp*

Синтаксис

```
extern PACKAGE int___fastcall
        AnsiPos(const AnsiString Substr, const AnsiString S);
extern PACKAGE char *___fastcall
                    AnsiStrPos(char * Str, char * SubStr);
extern PACKAGE char *___fastcall
                StrPos(const char * Str1, const char * Str2);
```

Описание

Функции **AnsiStrPos и StrPos** возвращают указатель на первое вхождение подстроки **SubStr (STR2)** в строку **Str (STR1).** Возвращаемое значение — указатель на первый символ найденной подстроки. Если **SubStr** нет в **Str,** возвращается **NULL.**

Если строки содержат символы кириллицы, надежнее использовать функцию **AnsiStrPos,** чем **StrPos.** А в случае многобайтных символов работает только **AnsiStrPos.**

Функция **AnsiPos** тоже осуществляет поиск первого вхождения подстроки, но возвращает индекс первого вхождения подстроки, отсчитанный от 1. Например, если строка S начинается с подстроки **Substr,** вернется 1. Если подстрока не найдена, возвращается 0.

Обратите внимание на разную последовательность аргументов в вызовах функций: в **AnsiStrPos и StrPos** первым аргументом указывается строка, а в **AnsiPos** — подстрока.

Примеры

Пусть вам надо произвести какие-то действия, если текст, хранящийся в строке, записанной в окне **Edit2,** встречается в тексте, записанном в окне **Editl.** Сравнение надо производить независимо от регистра, в котором набраны тексты.

Поиск текста одной строки в другой производится функцией **AnsiPos.** Но, чтобы сделать поиск нечувствительным к регистру, надо сначала привести обе строки к одному регистру, как это сделано с помощью функции **AnsiLowerCase** в следующем примере:

```
if (AnsiPos(AnsiLowerCase(Edit2->Text),
            AnsiLowerCase(Edit1->Text)) > 0)
...
```

или с помощью функции **AnsiUpperCase** в следующем:

```
if (AnsiPos(AnsiUpperCase(Edit2->Text),
            AnsiUpperCase(Edit1->Text)) > 0)
...
```

В приведенных примерах функцию **AnsiPos** можно заменить функцией **AnsiStrPos или StrPos:**

```
if (AnsiStrPos(AnsiLowerCase(Edit1->Text).c_str(),
               AnsiLowerCase(Edit2->Text).c_str()) != NULL)
...
```

Результат будет тем же самым. Обратите внимание на то, что в функциях **AnsiStrPos** и **StrPos,** в отличие от **AnsiPos,** подстрока передается вторым параметром, а строка — первым.

Следующий оператор отображает в метке Labell часть текста окна **Edit1,** начинающуюся с первого вхождения в него подстроки из окна **Edit2:**

```
Labell->Caption =
     AnsiStrPos(AnsiStrLower((Edit1->Text).c_str()),
                AnsiStrLower((Edit2->Text).c_str()));
```

Если вхождения подстроки не найдено, в метку ничего не заносится.

Во всех приведенных примерах вместо функций **AnsiStrLower и AnsiStrUpper** можно использовать функции **AnsiLowerCase. LowerCase. StrLower. AnsiUpperCase. UpperCase. StrUpper.** но с учетом ограничений, свойственных этим функциям.

AnsiStrComp — сравнение строк

Сравнивает две строки с учетом регистра.

Заголовочный файл *SysUtils.hpp*

Синтаксис

```
extern PACKAGE int __fastcall AnsiStrComp(char * S1, char * S2);
```

Описание

Функция сравнивает две строки **S1** и S2 с учетом регистра. Возвращает значение < 0, если **S1** < S2, 0, если **S1** = S2, и > 0, если **S1** > S2.

Сравнение строк осуществляется по символам, начиная с первого. Если очередные символы не равны друг другу, то строка, в которой символ больше, считается больше другой строки, и функция возвращает соответствующее значение. Заглавные буквы (символы, набранные в верхнем регистре) считаются больше прописных (набранных в нижнем регистре). Сравнение символов кириллицы производится в соответствии с русским алфавитом. Считается, что латинские символы меньше символов кириллицы, символы цифр меньше символов букв, символы пунктуации (включая пробел) меньше символов цифр. Если в процессе сравнения оказывается, что в одной строке символы закончились, а в другой еще имеются, строка с меньшим числом символов считается меньшей.

Функция применима к русским текстам, в то время, как аналогичная функция **StrComp** может на русских текстах давать сбои.

AnsiStrIComp — сравнение строк

Сравнивает две строки без учета регистра.

Заголовочный файл *SysUtils.hpp*

Синтаксис

```
extern PACKAGE int __fastcall AnsiStrIComp(char * S1, char * S2);
```

Описание

Функция сравнивает две строки **S1** и S2 без учета регистра. Возвращает значение < 0, если **S1** < S2, 0, если **S1** = S2, и > 0, если **S1** > S2. Функция работает как с латинскими символами, так и с кириллицей. Этим она отличается от функции **StrIComp**, работающей только с латинскими символами.

Сравнение строк осуществляется по символам, начиная с первого. Регистр, в котором набраны символы, не учитывается. Если очередные символы не равны друг другу, то строка, в которой символ больше, считается больше другой строки, и функция возвращает соответствующее значение. Сравнение символов кириллицы производится в соответствии с русским алфавитом. Считается, что латинские символы меньше символов кириллицы, символы цифр меньше символов букв, символы пунктуации (включая пробел) меньше символов цифр. Если в процессе сравнения оказывается, что в одной строке символы закончились, а в другой еще имеются, строка с меньшим числом символов считается меньшей.

См. также функцию **AnsiCompareText**, которая во многих отношениях удобнее функции **AnsiStrlComp.**

Пример

Следующие операторы обеспечивают различные действия (обозначены многоточиями) в зависимости от сравнения текстов в окнах **Edit1** и **Edit2.**

```
String S1 = Edit1->Text;
String S2 = Edit2->Text;
switch (AnsiStrIComp(S1.c_str(), S2.c_str()))
{
  case -1: ...;
           break;
  case 0:  ...;
           break;
  case 1:  ...;
}
```

Приведенный выше текст можно было бы упростить, удалив из него переменные S1 и S2 и заменив оператор **switch** следующим:

```
switch (AnsiStrIComp((Edit1->Text).c_str(), (Edit2->Text).c_str()))
```

В приведенной ниже таблице даны различные варианты строк S1 и S2 и результаты их сравнения

S1	S2	Возвращаемое значение
"строка 1"	"строка 2"	-1
"строка "	"строка"	+1
"Строка 1"	"строка 1"	0
"строка"	"строки"	-1

AnsiStrLower — преобразование строки к нижнему регистру

Преобразует строку к нижнему регистру.

См. разд. «AnsiLowerCase и другие функции преобразования строки к нижнему регистру».

AnsiStrPos — поиск подстроки

Возвращает указатель на позицию первого вхождения заданной подстроки в строку.

См. разд. «AnsiPos и другие функции поиска подстроки».

AnsiStrUpper — преобразование строки к верхнему регистру

Преобразует строку к верхнему регистру.

См. разд. «AnsiUpperCase и другие функции преобразования строки к верхнему регистр».

AnsiToOem — макрос перевода строки в текст DOS

Устаревший, оставленный для совместимости с 16-разрядными приложениями вариант перевода строки текста в текст MS-DOS. Сейчас реализован в виде макроса, использующего более современную функцию **CharToOem.** которую и следует вызывать при необходимости перевода строк (см. разд. «CharToOem и другие функции перевода строки в текст DOS»).

AnsiUpperCase и другие функции преобразования строки к верхнему регистру

Преобразуют строку к верхнему регистру.

Заголовочные файлы *SysUtils.hpp*,*string.h*, *mbstring.h*.

Синтаксис

```
extern PACKAGE char *__fastcall AnsiStrUpper(char * Str);
extern PACKAGE AnsiString__fastcall
                        AnsiUpperCase(const AnsiString S);
extern PACKAGE char *__fastcall StrUpper(char * Str);
extern PACKAGE AnsiString__fastcall
                        UpperCase(const AnsiString S);

char *strupr(char *s);
wchar_t *_wcsupr(wchar_t *s);
unsigned char *_mbsupr(unsigned char *s) ;
```

Описание

Функции возвращают строку **Str** или s типа (char *), или строку S типа AnsiString, преобразованную к верхнему регистру. Функции **StrUpper, UpperCase, strupr, _wcsupr, _mbsupr** работают только с латинскими символами и неприменимы к символам кириллицы. Для русских текстов должны использоваться функции **AnsiStrUpper и AnsiUpperCase.** Функция **AnsiUpperCase** применима также к многобайтным символам.

При работе с компонентами VCL обычно удобнее использовать функции **AnsiUpperCase и Uppercase,** работающие со строками типа **AnsiString.**

Примеры

Пусть в окне редактирования **Edit1** записан текст: "Hi! Привет!". Ниже приведена таблица, в которой показаны результаты, возвращаемые различными функциями.

Выражение	Результат
AnsiUpperCase(Edit1->Text)	HI! ПРИВЕТ!
UpperCase(Edit1->Text)	HI! Привет!
AnsiStrUpper((Edit1->Text).c_str())	HI! ПРИВЕТ!
StrUpper((Edit1->Text).c_str())	HI! Привет!

Вы можете видеть, что функции AnsiUpperCase, **AnsiStrUpper** работают нормально, а функции UpperCase, **StrUpper** не преобразует русский текст. Можно также видеть, что при работе со свойствами компонентов VCL функции **AnsiUpperCase, UpperCase** удобнее, поскольку не требуют приведения типов.

См. также примеры в разд. «AnsiPos и другие функции поиска подстроки».

assert — макрос диагностики

Обеспечивает диагностику при отладке.

Синтаксис

```
#include <assert.h>
void assert(int test);
```

Описание

Макрос **assert** используется в программах для диагностики. Если при расширении макроса значение параметра **test** ложно (равно нулю), то **assert** выдает в стандартный файл ошибок stderr сообщение:

```
Assertion failed: test, file <имя файла>, line <номер строки>
```

После этого макрос **assert** производит вызов функции **abort.**

Если в исходном файле перед директивой

```
#include <assert.h>
```

появляется директива препроцессора

```
#define NDEBUG
```

то все последующие макросы **assert** игнорируются. Таким образом, вы можете ввести в свое приложение какие-то проверки, необходимые для отладки, а затем в окончательном файле отключить их приведенной выше директивой.

Пусть, например, вы хотите проверять, не окажется ли в результате какой-то ошибки введенный вами указатель P равным **NULL.** Тогда вы в соответствующем месте кода можете ввести оператор:

```
assert(P == NULL);
```

Если при выполнении этого оператора значение P окажется равным **NULL,** то будет отображено диалоговое окно, содержащее сообщение об ошибке, и приложение завершит работу.

Bounds и другие функции формирования прямоугольной области

Формируют прямоугольную область типа **TRect.**

Заголовочные файлы *Types.hpp, Classes.hpp.*

Синтаксис

```
#include <Types.hpp>
struct TRect
{
 int left, top, right, bottom;
};
```

```
struct TPoint
(
 int  x;
 int  y;
};

extern PACKAGE Types::TRect___fastcall
          Bounds(int ALeft, int ATop, int AWidth, int AHeight);

#include <Classes.hpp>
extern PACKAGE TRect___fastcall
        Rect(int ALeft, int ATop, int ARight, int ABottom);
. extern PACKAGE TRect___fastcall
        Rect (const TPoint ATopLeft, const TPoint ABottomRight) ;
```

Описание

Функции возвращают структуру типа **TRect.** используемую во многих функциях C++Builder и Windows. Она определяет в пикселах размеры и размещение различных окон и областей. В качестве системы координат принимается система координат родительского окна или экрана. За начало координат всегда принимается левый верхний угол родительского окна или экрана.

Функция **Bounds** формирует **TRect** из координат X (ALeft) и Y (ATop) левого верхнего угла области, из ее ширины **(AWidth)** и высоты (AHeight). Первая форма функции **Rect** вместо ширины и высоты задает координаты X **(ARight)** и Y (ATop) правого нижнего угла области. А вторая форма функции **Rect** формирует область заданием двух точек типа **TPoint,** определяющих ее левый верхний и правый нижний углы. Задавать значения **TPoint** обычно удобно функцией **Point.**

В C++Builder рассматриваемые функции применяются, в частности, для задания значений таким свойствам компонентов, как **BoundsRect, ClientRect и др.**

Примеры

Приведенный ниже оператор размещает окно текстового редактора **Memo1** на его родительской панели **Panel1,** оставляя слева, внизу и справа зазор в 10 пикселов (для более приятного вида), а сверху — зазор 40 пикселов (например, для размещения заголовка окна):

```
Memo1->BoundsRect = Rect(10,40,Panel1->ClientWidth-10,
                         Panel1->ClientHeight-10);
```

То же самое можно сделать оператором:

```
Memo1->BoundsRect = Bounds(10,40,Panel1->ClientWidth-20,
                           Panel1->ClientHeight-50);
```

Ниже приведен ряд операторов, иллюстрирующих функции **Rect и Point.** а также применение типов **TRect и TPoint:**

```
TRect R, R1, R3;
R = Rect(10,100,20,200);
Rl = R;

TPoint P1, P2;
P1 = Point(10, 100) ;
P2 = Point (10, 200);
R3 = TRect(P1, P2);

TRect R2(P1, P2);

R.Left = 15;
int W = R.Width(); <
if(R1 != R) ...
```

calloc — функция выделения памяти

Функция выделяет память под заданное число объектов.
См. разд. «malloc и другие функции динамического распределения памяти».

ceil — округление действительного числа

Округляет действительное число до целого значения.
См. разд. «Ceil и другие функции округления действительных чисел».

Ceil и другие функции округления действительных чисел

Функции округления действительных чисел до целых значений.

Заголовочные файлы *math.h* и *math.hpp*.

Синтаксис

```
#include <math.h>
double ceil(double x) ;
double floor(double x);
long double ceill (long double x);
long double floorl(long double x);

ttinclude<math.hpp>
extern PACKAGE int___fastcall Ceil(Extended X);
extern PACKAGE int___fastcall Floor(Extended X);
```

Описание

Функции **ceil, ceill, Ceil, floor, floorl, Floor** округляют значения своих аргументов, являющихся действительными числами различных типов. Округление производится в разные стороны: **ceil, ceill и Ceil** округляют в сторону положительной бесконечности (до минимального целого числа, не меньшего, чем значение аргумента); **floor, floorl и Floor** округляют в сторону отрицательной бесконечности (до максимального целого числа, не большего, чем значение аргумента). Обратите внимание, что результат выполнения всех этих функций — не целое значение, а действительное, округленное до целого.

Ниже приведены примеры округления:

Функция	X = 3.5	X = -3.5	X = 3
ceil, ceill, Ceil	4	-3	3
floor, floorl, Floor	3	-4	3

ceill — округление действительного числа

Округляет действительное число до целого значения.
См. разд. «Ceil и другие функции округления действительных чисел».

cgets — ввод строки из потока

Вводит строку из стандартного потока.
См. разд. «fputs и другие функции ввода/вывода строк».

CharToOem, CharToOemBuff — функции перевода строки в текст DOS

Функции API Windows, переводят строку в текст MS-DOS.

Заголовочный файл *winuser.h.*

Синтаксис

```
#include <system.hpp>
BOOL CharToOem(
    LPCTSTR lpszSrc,     // исходная строка
    LPSTR lpszDst         // результат перевода
   );
BOOL CharToOemBuff(
    LPCTSTR IpszSrc,     // исходная строка
    LPSTR IpszDst,       // результат перевода
    DWORD cchDstLength  // число символов
   );
```

Описание

Функции применяются для перевода строки текста в формате ASCII («просто текст») в строку формата «текст MS-DOS». Это требуется, в частности, в консольных приложениях для вывода на экран русских текстов. Необходим подобный перевод и в случаях, когда в окно редактирования загружен русский текст и его надо сохранить в файле в формате DOS.

Параметр IpszSrc — указатель на строку, которую надо перекодировать. Параметр **IpszDst** — указатель на строку, в которую заносится перекодированный текст. Параметр **cchDstLength** в функции **CharToOemBuff** определяет число перекодированных символов, которые заносятся в результирующую строку. Если это число меньше числа символов в исходной строке, то остальные символы не заносятся в результирующую строку.

Имеются два варианта функций, работающие с кодами ANSI и с многобайтными кодами Unix. В случае, если работа идет с кодами ANSI, адреса исходной и результирующей строк могут совпадать, т.е. параметры **IpszSrc** и **IpszDst** могут указывать на одну строку.

Функции всегда возвращают ненулевое значение.

Имеется также функция **OemToChar.** которая осуществляет обратное преобразование.

Примеры

Ниже приведен пример консольного приложения, демонстрирующий вывод и ввод сообщений в кодировке DOS. Для вывода русских текстов используется функция **CharToOem.**

```
#include <stdio.h>
#include <system.hpp>

int main()
{
   char S1[20], S2[20];
   CharToOem("Введите Ваше имя:\n", S1);
   printf(S1);
   scanf("%20s", S2);
   CharToOem("Привет, ", S1);
   printf(strcat(strcat(S1,S2), "!!!\n"));
   // Чтобы не закрылось окно DOS
   fflush(stdin);
   getchar();
   return 0;
}
```

В следующем примере, текст из окна **RichEdit1** сохраняется в формате DOS в файле, указанном пользователем.

```
#include <stdio.h>
...
if (SaveDialog1->Execute())
{
  char *S = (char *) malloc(sizeof(RichEdit1->Text)+1);
  CharToOem((RichEdit1->Text).c_str(),S);
  FILE *F;
  F = fopen((SaveDialog1->FileName).c_str(), "wt");
  fprintf(F,"%s", S);
  fclose(F);
  free(S);
}
```

Первый оператор в структуре if вызывает диалог сохранения файла. Если пользователь выбрал в нем файл, то далее отводится память под строку S, необходимая для хранения в ней текста окна **RichEditl.** Следующий оператор заносит в нее перекодированный текст. Дальнейшие операторы создают текстовый файл с заданным именем, заносят в него перекодированный текст и закрывают файл. Последний оператор освобождает память.

_clear87 и другие функции очистки слова состояния FPU

Очищают слово состояния FPU.

Заголовочный файл *float.h*.

Синтаксис

```
#include <float.h>
unsigned int _clear87 (void);
unsigned int _clearfp (void);
```

Описание

Функции **_clear87 и _clearfp** очищают слово состояния FPU (см. разд. 1.9.3), отображающее состояние после выполнения операций с плавающей запятой. Это слово содержит следующие флаги:

Флаг	Описание	Биты
IE	Исключение при ошибочных операциях	0
DE	Исключение ненормализованных операций	1
ZE	Исключение деление на нуль	2
OE	Исключение переполнения	3
UE	Исключение потери порядка	4
PE	Исключение точности	5
SF	Ошибка стека	6
ES	Состояние ошибочного суммирования	7
C0	Условный код 0 (CF)	8
C1	Условный код 1	9
C2	Условный код 2 (PF)	10
ST	Вершина стека	11-13
C3	Условный код 3 (ZF)	14
BF	Флаг занятости FPU	15

Слово состояния отражает характер результата текущей операции с плавающей запятой. Например, если при выполнении приложения встретилось деление переменной с плавающей запятой на нуль, флаг ZE слова состояния переключится в 1, свидетельствуя об этой ошибочной операции. Вот такие установленные биты и сбрасываются функциями **_clear87 и _clearfp.**

Обе функции идентичны. Функция **_clearfp** определена только для совместимости с Microsoft.

Функции возвращают значение слова состояния до того, как оно очищено. Возвращенное значение можно использовать для анализа результатов выполнения арифметических операций с плавающей запятой. Это имеет смысл применять в случае, если до возникновения ошибки генерация исключений замаскирована функцией **_control87.** Например, оператором:

```
_control87(0x3F, 0x3F);
```

Тогда при возникновении ошибок операций с плавающей запятой исключения не генерируются и о наличии ошибок можно судить, проверяя отдельные биты возвращенного слова операцией И. Например, оператор

```
if ( _clear87() & 0x4) ShowMessage("Деление на нуль");
```

прореагирует на произошедшее в предшествующих операциях деление на нуль и очистит слово. А оператор

```
if (_clear87() & 0x3D)
  ShowMessage("Ошибка операции с плавающей запятой");
```

прореагирует на любую ошибку операций с плавающей запятой и очистит слово от последствий ошибок.

_clearfp — очистка слова состояния FPU

Очищает слово состояния FPU.

См. разд. «_clear87 и другие функции очистки слова состояния FPU».

CompareDate и другие функции сравнения дат и времени

Сравнивают два значения дат и времени.

Заголовочный файл *DateUtils.hpp.*

Синтаксис

```
#include <DateUtils.hpp>
typedef int TValueRelationship;

extern PACKAGE Types::TValueRelationship___fastcall
               CompareDate(const System::TDateTime A,
                           const System::TDateTime B);
extern PACKAGE Types::TValueRelationship___fastcall
          CompareDateTime(const System::TDateTime A,
                           const System::TDateTime B);
extern PACKAGE Types::TValueRelationship___fastcall
               CompareTime(const System::TDateTime A,
                           const System::TDateTime B);
```

Описание

Функции **CompareDate, CompareDateTime, CompareTime** сравнивают два значения даты и времени A и B типа **TDateTime.** но по-разному. Функция **CompareDate** сравнивает только даты, игнорируя время. Так что два значения, относящихся к одной и той же дате, но различающихся по времени, считаются одинаковыми. Функция **CompareDateTime** сравнивает и дату, и время. При этом осуществляется полное сравнение, причем более достоверное, чем сравнение A и B как действитель-

ных чисел. Функция **CompareTime** сравнивает только время, игнорируя дату. Так что она считает равными значение, которые совпадают по времени вплоть до миллисекунды, но относятся к разным датам. И может, например, указать, что A > B, если время A больше B, хотя при этом дата A может быть меньше B.

Все функции возвращают:

Значение	Именованная константа	Условие
-1	**LessThanValue**	A <B
0	EqualsValue	A = B
+1	**GreaterThan**Value	A > B

CompareDateTime — сравнение дат и времени

Сравнивает два значения дат и времени.

См. разд. «CompareDate и другие функции сравнения дат и времени».

CompareText — сравнение строк

Сравнивает две строки без учета регистра.

Заголовочный файл *SysUtils.hpp.*

Синтаксис

```
extern PACKAGE int __fastcall CompareText(const AnsiString S1,
                                          const AnsiString S2);
```

Описание

Функция сравнивает две строки **S1** и S2 типа **AnsiString.** Для латинских текстов сравнение происходит без учета регистра. Для кириллицы это не работает. Возвращает значение > 0, если **S1** = S2. В остальных случаях возвращается 0.

Например, выражение

```
CompareText(Edit1->Text, Edit2->Text)
```

вернет 0, если в окнах **Edit1** и **Edit2** записан одинаковый текст или он не содержит символов кириллицы и различается только регистром. Например, "Edit" и "edit". В остальных случаях вернется положительное значение. Но если тексты содержат символы кириллицы в разных регистрах (например, "Окно" и "окно"), то вернется положительное число, т.е. строки будут признаны разными.

Для сравнения русских текстов следует использовать функцию **AnsiStrComp**.

CompareTime — сравнение значений времени

Сравнивает два значения времени.

См. разд. «CompareDate и другие функции сравнения дат и времени».

CompareValue и другие функции сравнения числовых значений

Сравнивают два числовые значения.

Заголовочный файл *Math.hpp.*

Объявления

```
typedef int TValueRelationship;

extern PACKAGE TValueRelationship __fastcall
                        CompareValue(const int A, const int B);
extern PACKAGE TValueRelationship __fastcall
                  CompareValue(const __int64 A, const __int64 B);
```

```
extern PACKAGE TValueRelationship__fastcall
                  CompareValue(const float A, const float B,
                               float Epsilon = 0);
extern PACKAGE TValueRelationship__fastcall
                 CompareValue(const double A, const double B,
                              double Epsilon = 0);
extern PACKAGE TValueRelationship__fastcall
                CompareValue(const Extended A, const Extended B,
                             Extended Epsilon =0);

extern PACKAGE bool__fastcall SameValue(const float A,
                               const float B, float Epsilon = 0);
extern PACKAGE bool__fastcall SameValue(const double A,
                              const double B, double Epsilon = 0);
extern PACKAGE bool__fastcall SameValue(const Extended A,
                             const Extended B, Extended Epsilon = 0);
```

Описание

Различные перегруженные формы функции **CompareValue** сравнивают значения двух своих числовых аргументов A и B различного типа. Функции возвращают:

Значение	Именованная константа	Условие
-1	LessThanValue	A < B
0	EqualsValue	A = B
+1	GreaterThanValue	A > B

При сравнении действительных значений параметр **Epsilon** позволяет задать различие значений, при котором они еще считаются равными. Это дает возможность устранить влияние ошибок округления.

Перегруженные формы функций **SameValue** осуществляют сравнение A и B только на эквивалентность с точностью до **Epsilon.** Функции возвращают **true,** если модуль разности значений A и B не превышает **Epsilon.** Впрочем, по умолчанию **Epsilon** = 0, так что осуществляется точное сравнение. Поскольку аргументы — действительные числа, то в этом случае несовпадение может быть связано с ошибками округления. Так что обычно лучше задавать конечное значение **Epsilon.**

Примеры

Пусть A и B — действительные числа, причем A = 10.05, а B = 10. Тогда выражение **CompareValue(A, B)** вернет +1, а выражение **CompareValue(A, B, 0.01*abs(B))** вернет 0, так как это выражение сравнивает число с точностью до 1%. Выражение **SameValue(A, B)** вернет **false,** а выражение **SameValue(A, B, 0.01*abs(B))** вернет **true.**

_control87 и другие функции доступа к управляющему слову FPU

Обеспечивают доступ к управляющему слову FPU.

Заголовочные файлы *float.h, System.hpp.*

Синтаксис

```
#include <float.h>
unsigned int _control87(unsigned int newcw, unsigned int mask);
unsigned int _controlfp(unsigned int newcw, unsigned int mask);

#include <System.hpp>
extern PACKAGE Word __fastcall Get8087CW(void);
extern PACKAGE void __fastcall Set8087CW(Word NewCW);
```

Описание

Функции обеспечивают доступ к управляющему слову FPU (см. разд. 1.9.3), определяющему точность вычислений, способы округления и генерацию исключений при выполнении операций с плавающей запятой.

Программную установку управляющего слова имеет смысл проводить, если вы решили запретить генерацию каких-то видов исключений. Это, в частности, приходится делать при использовании некоторых пакетов. Например, при использовании для трехмерной графики OpenGL надо запретить генерацию всех исключений.

В функциях **_control87** и **_controlfp** параметр **newcw** содержит устанавливаемое значение управляющего слова FPU. Параметр **mask** — маска, которая определяет, какие именно биты из **newcw** будут действительно заноситься в управляющее слово. В управляющем слове заменяются только те биты, для которых в **mask** заданы 1. Например, вызов функции

```
_control87(0x4, 0x4)
```

использует маску, двоичное представление которой равно 0100. Значит, такой вызов функции установит в единицу третий бит (если младший бит считать первым) — маску исключений деления на нуль, оставив остальные биты управляющего слова неизменными. Вызов функции

```
_control87(0x4, 0x3F)
```

(маска 00111111, новое значение слова — 00000100) установит в единицу 3-й бит и в 0 все остальные из первых 6-ти битов, отвечающих за маски исключений. Остальные биты управляющего слова останутся неизменными.

Функции возвращают новое значение управляющего слова. Если же маска равна нулю, то вернется текущее значение слова. Например, оператор

```
unsigned int m = _control87(0, 0);
```

вернет в переменную m текущее значение управляющего слова, причем само слово останется неизменным.

Функция **_controlfp** введена для совместимости с Microsoft. Она отличается от **_control87** только тем, что всегда выключает флаг DM, соответствующий ненормализованным операциям.

Функция **Get8087CW** возвращает текущее состояние управляющего слова. А функция Set8087CW позволяет установить новое значение слова **NewCW.**

При установке слова функциями **_controlfp, _control87, Set8087CW** можно использовать переменную **Default8087CW.** которая содержит значение по умолчанию управляющего слова FPU.

Рассмотренные функции — не самый удобный способ решать задачи изменения управляющего слова. Часто более удобно использовать функции **GetExceptionMask. SetExceptionMask. GetPrecisionMode. SetPrecisionMode. GetRoundMode. SetRoundMode**.

Примеры

Если вам в каком-то фрагменте кода надо запретить генерацию исключения при делении на нуль, а затем восстановить прежнее состояние управляющего слова, это можно сделать следующим кодом:

```
// запоминание слова
unsigned int m = _control87(0, 0);
_control87(0x4, 0x4); // маскирование исключения

<код, в котором может возникнуть исключение>

_control87(m, 0xFFFF); // восстановление слова
```

Первый оператор этого кода запоминает в переменной m управляющее слово с помощью функции **_control87** с нулевой маской. Затем той же функцией с соответствующей маской маскируется исключение. А последний оператор восстанавливает прежнее значение управляющего слова.

Ниже приведен аналогичный пример, использующий функции **Get8087CW** и **Set8087CW**:

```
Word Old8087CW;
Old8087CW = Get8087CW(); //запоминание слова
Set8087CW(0x133f);       // маскируются все исключения

<код, в котором могут возникать исключения>

Set8087CW(Old8087CW);    // восстановление слова
```

Еще один пример. Пусть вы хотите в некоторый момент запретить генерацию исключения, связанного с делением на нуль. Для этого вам надо установить в 1 бит **ZM** (см. разд. 1.9.3), запрещающий генерацию этого исключения. Но этого мало, так как при этом все-таки будет сгенерировано исключение, связанное с ошибочной операцией деления. Так что одновременно надо установить в 1 бит **OM** (маску исключения переполнения), или бит **IM,** запрещающий генерацию исключения при ошибочных операциях. Остальные биты слова мы хотим оставить без изменения.

Эту задачу можно решить оператором:

```
Set8087CW(Get8087CW() | 0xD);
```

Он устанавливает в слове, возвращаемом функцией **Get8087CW**, биты ZM и OM в 1, оставляя остальные биты неизменными. Аналогичный результат дает оператор

```
Set8087CW(Get8087CW | 5);
```

который устанавливает в 1 биты ZM и IM.

Многочисленные примеры использования рассмотренных функций вы найдете также в разд. 1.9.3, гл. 1 и в описаниях других функций, работающих с управляющим словом FPU.

_controlfp — доступ к управляющему слову FPU

Обеспечивает доступ к управляющему слову FPU.

См. разд. «_control87 и другие функции доступа к управляющему слову FPU».

cprintf — форматированный вывод на экран

Выводит форматированные данные в выходной поток.

См. разд. «fprintf и другие функции форматированного вывода».

cputs — вывод строки в поток

Выводит строку в стандартный поток вывода.

См. разд. «fputs и другие функции ввода/вывода строк».

CreateMessageDialog — создание диалогового окна

Создает диалоговое окно, позволяющее анализировать ответ пользователя.

См. разд. «MessageDlg и другие функции отображения диалоговых окон».

CreateProcess — порождение дочернего процесса

Функция API Windows, порождает дочерний процесс.

Объявление

```
bool___fastcall CreateProcess(
        const char * lpApplicationName,
        char * lpCommandLine,
        _SECURITY_ATTRIBUTES * lpProcessAttributes,
        _SECURITY_ATTRIBUTES * lpThreadAttributes,
        bool bInheritHandles,
        unsigned long dwCreationFlags,
        void * lpEnvironment,
        const char * lpCurrentDirectory,
        STARTUPINFO * lpStartupInfo,
        PROCESS_INFORMATION * lpProcessInformation
);
```

Описание

Функция **CreateProcess** порождает новый дочерний процесс и его первый поток (нить). В рамках этого процесса выполняется указанный файл **lpApplicationName** с командной строкой **lpCommandLine.** Впрочем, параметр **lpApplicationName** может быть равен NULL, а имя выполняемого модуля в этом случае должно быть первым элементом командной строки, задаваемой параметром **lpCommandLine.** Сам выполняемый модуль может быть любого вида: 32-разрядным приложением Windows, приложением MS-DOS, OS/2 и т.п. Однако если из приложения Windows создается процесс MS-DOS, то параметр **lpApplicationName** должен быть равен NULL, а имя файла и его командная строка включаются в **lpCommandLine.** Так что, как правило, чтобы не ошибиться, проще всегда задавать **lpApplicationName = NULL** и помещать всю информацию в **lpCommandLine.**

Если имя файла не содержит расширения, то предполагается расширение .exe. Но если имя кончается символом точки или если файл задан вместе с путем, то расширение **.exe** к имени не добавляется.

Если путь к файлу не задан, файл ищется в каталогах в следующей последовательности:

- Каталог, из которого запускается приложение
- Текущий каталог родительского процесса
- Системный каталог Windows, возвращаемый функцией **GetSystemDirectory**
- Каталог SYSTEM (Windows NT/2000/XP).
- Каталог Windows, возвращаемый функцией **GetWindowsDirectory**
- Каталоги, перечисленные в переменной окружения PATH

Если функция успешно выполнена, она возвращает ненулевое значение **(true).** Если произошла ошибка — возвращается 0 (false). Тогда информацию об ошибке можно получить, вызвав функцию **GetLastError.**

Функция **CreateProcess** пришла на смену прежним функциям **WinExec и LoadModule,** которые теперь реализуются посредством вызова **CreateProcess.**

Функция **CreateProcess** возвращается, не ожидая окончания инициализации порождаемого процесса. Но в ряде случаев родительский процесс должен взаимодействовать с порожденным. Такое взаимодействие возможно только после того, как закончена инициализация порожденного процесса. Приостановить выполнение до окончания инициализации дочернего процесса можно функцией **WaitForInputIdle.** В некоторых случаях выполнение родительского процесса должно быть приостановлено до завершения порожденного процесса. Это необходимо, например, если родительский процесс должен использовать какие-то результаты, полученные порожденным процессом. Для ожидания завершения порожденного процесса можно использовать функцию **WaitForSingleObject.**

Порожденный процесс остается в памяти системы, пока не завершатся все его потоки (нити) и пока все его дескрипторы не закроются вызовом **CloseHandle.**

Если эти дескрипторы не нужны, лучше всего закрыть их сразу после инициализации процесса.

Чтобы досрочно прекратить выполнение дочернего процесса лучше всего использовать функцию **ExitProcess**.

Множество параметров функции позволяют определить условия выполнения и управлять дочерним процессом. Ниже приведено краткое описание параметров функции.

lpApplicationName	Указатель на строку, содержащую имя выполняемого модуля: или с полным путем, или только имя (тогда файл должен находиться в текущем каталоге). Если IpApplicationName = NULL, имя модуля должно задаваться первым элементом строки IpCommandLine (подробнее см. выше в описании функции).
lpCommandLine	Указатель на строку, содержащую командную строку выполняемого файла. Если IpCommandLine = NULL, то в качестве командной строки выступает IpApplicationName (подробнее см. выше в описании функции).
lpProcessAttributes, lpThreadAttributes	Указатели на структуры типа PSequrityAttributes, определяющие наследование дескриптора в дочернем процессе. Если эти параметры равны NULL, наследование невозможно.
bInheritHandles	Определяет, наследуют ли новые процессы дескрипторы родительских. Если true — наследуют с тем же уровнем доступа, что и в родительском процессе.
dwCreationFlags	Определяет флаги dwCreationFlags, задающие характеристики создаваемого процесса.
lpEnvironment	Указывает на блок окружения нового процесса. Если параметр равен NULL, используется окружение родительского процесса. Блок окружения состоит из оканчивающихся нулевым символом строк вида: `<имя>=<значение>` Если задан блок окружения, то информация о текущем каталоге в окружение нового процесса автоматически не передается. Блок может состоять из символов UNICODE или ANSI (см. флаги dwCreationFlags). Блок ANSI должен завершаться двумя нулевыми символами (один для строки, другой для блока). Блок UNICODE должен завершаться четырьмя нулевыми символами.
lpCurrentDirectory	Указывает на строку, определяющую текущий каталог и диск дочернего процесса. Это используется в приложениях — оболочках, выполняющих различные приложения с различными рабочими каталогами. Если параметр равен NULL, текущий каталог совпадает с родительским.
lpStartupInfo	Указывает на структуру типа STARTUPINFO или тождественного ему **TStartupInfo**, определяющую основное окно дочернего процесса.
lpProcessInformation	Указывает на структуру типа PROCESS_INFORMATION или тождественного ему TProcessInformation, из которой родительское приложение может получать информацию о выполнении нового процесса.

Параметры **lpProcessAttributes, lpThreadAttributes, lpEnvironment, bInheritHandles** определяют наследование дочерним процессом свойств родительского процесса. Если не вдаваться в подробности наследования, то можно первые три из этих параметров задавать равными **NULL,** а последний — **false.** Параметр **lpCurrentDirectory** указывает на строку, определяющую текущий каталог и диск дочернего процесса. Это используется в приложениях-оболочках, выполняющих различные приложения с различными рабочими каталогами. Если параметр равен NULL, текущий каталог совпадает с родительским.

Параметр **dwCreationFlags** определяет флаги, задающие характеристики создаваемого процесса.

Указанные ниже флаги, управляющие созданием процесса, могут задаваться в любых комбинациях (кроме специально оговоренных) с помощью операции ИЛИ (|).

CREATE_DEFAULT_ ERROR_MODE	Новый процесс не наследует режим ошибок родительского процесса. В нем устанавливается режим по умолчанию вызовом **SetErrorMode.** Обычно используется в многопоточных процессах.
CREATE_NEW_CONSOLE	Создается новое консольное приложение. Этот флаг не может использоваться совместно с **DETACHED_PROCESS.**
CREATE_NEW_PROCESS_ GROUP "	Новый процесс является корневым для новой группы процессов: всех процессов, которые будут наследовать создаваемому. Идентификатор новой группы — тот, который возвращается параметром **lpProcessInformation.** Группы процессов используются функцией **GenerateConsoleCtrlEvent** для посылки сигналов Ctrl-C или Ctrl-Break группе консольных процессов.
CREATE_SEPARATE_ WOW_VDM	Используется только в Windows NT для создания 16-битных процессов. Его установка приводит к использованию для процесса отдельной VDM. В этом случае отказ в данном процессе не приведет к гибели других выполняемых процессов.
CREATE_SHARED _ WOW_VDM	Используется только в Windows NT для создания 16-битных процессов. Если ключ **DefaultSeparateVDM** в разделе Windows файла *WIN.INI* установлен в true, то этот флаг приводит к изменению ключа и все новые процессы запускаются в общей VDM.
CREATE_SUSPENDED	Основной поток (нить) нового приложения создается в состоянии ожидания и не выполняется, пока не будет вызвана функция **ResumeThread.**
CREATE_UNICODE_ ENVIRONMENT	При установке этого флага блок окружения, на который указывает **lpEnvironment,** использует символы Unicode. В отсутствие флага используются символы ANSI.
DEBUG_PROCESS	При установке этого флага родительский процесс воспринимается как отладчик дочернего процесса. Система информирует отладчик обо всех событиях в отлаживаемом процессе. В этом режиме функция **WaitForDebugEvent** может использоваться только для потока, созданного функцией **CreateProcess.**

DEBUG_ ONLY_ THIS_ PROCESS	Если этот флаг отсутствует, и родительский процесс отлаживается, новые процессы тоже отлаживаются тем же отладчиком.
DETACHED_PROCESS	Новый консольный процесс не имеет доступа к консоли родительского. Он может позднее вызвать функцию **AllocConsole** для создания новой консоли. Этот флаг не может использоваться совместно с флагом **CREATE_NEW_CONSOLE.**

Параметр **dwCreationFlags** может также контролировать класс приоритета нового процесса. Если ни один из описанных ниже флагов приоритета не установлен, по умолчанию используется **NORMAL_PRIORITY_CLASS,** если только родительский процесс не имеет класс **IDLE_PRIORITY_CLASS.** В последнем случае для дочерних процессов по умолчанию принимается класс **IDLE_PRIORITY_CLASS.**

Приоритет может задаваться одним из следующих флагов:

HIGH_PRIORITY_CLASS	Указывает на процесс как на критическую задачу, которая должна выполняться немедленно.
IDLE_PRIORITY_CLAS S	Все потоки процесса выполняются только во время простоя системы. Пример — хранители экрана. Все наследники такого процесса будут иметь тот же класс приоритета.
NORMAL_ PRIORITY_ CLASS	Нормальный приоритет процесса.
REALTIME_PRIORITY_ CLASS	Высокий приоритет, превышающий приоритеты других процессов, включая приоритеты процессов операционной системы.

Параметр **lpStartupInfo** указывает на структуру типа **TStartupInfo,** соответствующего типу структуры **STARTUPINFO** API Windows. Структура определяет свойства главного окна создаваемого процесса. Для процессов с графическим интерфейсом пользователя (GUI) эта информация относится к первому окну, создаваемому функцией **CreateWindow** и отображаемому функцией **ShowWindow.** Для консольных приложений эта информация относится к создаваемому консольному окну.

Объявление этого типа:

```
typedef STARTUPINFOA STARTUPINFO;
typedef LPSTARTUPINFOA LPSTARTUPINFO;
typedef struct _STARTUPINFOA {
    DWORD   cb;
    LPSTR   lpReserved;
    LPSTR   lpDesktop;
    LPSTR   lpTitle;
    DWORD   dwX;
    DWORD   dwY;
    DWORD   dwXSize;
    DWORD   dwYSize;
    DWORD   dwXCountChars;
    DWORD   dwYCountChars;
    DWORD   dwFillAttribute;
    DWORD   dwFlags;
    WORD    wShowWindow;
    WORD    cbReserved2;
    LPBYTE  lpReserved2;
    HANDLE  hStdInput;
```

```
    HANDLE  hStdOutput;
    HANDLE  hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
```

Поля структуры обозначают следующее:

Cb	Размер в байтах данной структуры.
lpReserved	Зарезервировано. Пока значение должно быть **NULL.**
lpDesktop	Только для Windows NT. Указывает на строку или только с именем **desktop,** или *с* именем окна и **desktop** для данного процесса.
lpTitle	Для консольных процессов — надпись в заголовке окна. Для процессов GUI и консольных, не создающих новое окно, значение должно быть **NULL.**
dwX, dwY	Игнорируются, если **dwFlags** не включает флаг **STARTF_USEPOSITION.** При включенном флаге **STARTF_USEPOSITION** определяют координаты левого верхнего угла окна.
dwXSize, dwYSize	Игнорируется, если **dwFlags** не включает флаг **STARTF_USESIZE.** При включенном флаге **STARTF_USESIZE** определяют ширину и высоту окна.
dwXCountChars, dwYCountChars	Игнорируются, если **dwFlags** не включает флаг **STARTF_USECOUNTCHARS.** При включенном флаге **STARTF_USECOUNTCHARS** и только для консольных приложений, создающих новое окно, определяют буферы ширины и высоты экрана в числе символов. В остальных случаях эти поля игнорируются.
dwFillAttribute	Игнорируются, если **dwFlags** не включает флаг **STARTF_USEFILLATTRIBUTE.** При включенном флаге **STARTF_USEFILLATTRIBUTE** и только для консольных приложений, создающих новое окно, определяют цвета текста и фона. Значение поля может быть комбинацией следующих значений: **FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, BACKGROUND_INTENSITY.** Например, следующая комбинация определяет красный цвет текста на белом фоне: `FOREGROUND_RED BACKGROUND_RED	BACKGROUND_GREEN	BACKGROUND_BLUE.`

dwFlags	Битовое поле, определяющее флаги, указывающие на использование тех или иных полей структуры при создании окна процесса. Может содержать любые комбинации следующих значений: **STARTF_USESHOWWINDOW**— игнорировать поле **wShow**Window STARTF_USEPOSITION — использовать поля **dwX** и dwY **STARTF_USESIZE** — использовать поля **dwXSize** и dwYSize **STARTF_USECOUNTCHARS** — использовать поля dwXCo-**untChars** и **dwYCountChars** STARTF_USEFILLATTRIBUTE — использовать поле **dwFill-Attribute** STARTF_FORCEONFEEDBACK — если этот флаг установлен, курсор указывает на процесс создания на протяжении 2 секунд после вызова **CreateProcess.** Если за это время процесс сделал первый вызов GUI, система дает еще 5 секунд на процесс показа окна. Если за это время окно не нарисовано, дается еще 5 секунд на завершение рисования. Форма курсора восстанавливается после первого вызова GetMessage STARTF_FORCEOFFFEEDBACK — восстанавливать форму курсора сразу после запуска процесса **STARTF_USESTDHANDLES** — использовать дескрипторы потоков ввода, вывода и ошибок, указываемые полями hStdlnput, hStdOutput, hStdError
wShowWindow	Игнорируется, если dwFlags не включает флаг **STARTF_USESHOWWINDOW.** Может содержать любые константы **SW_**, объявленные в файле *WINUSER.H* Например, **SW_SHOWNORMAL** — обычное окно, **SW_HIDE** – невидимое.
cbReserved2	Зарезервировано, должно равняться 0.
lpReserved2	Зарезервировано, должно равняться NULL.
hStdInput	Игнорируется, если dwFlags не включает флаг STARTF_USESTDHANDLES. Определяет дескриптор стандартного потока ввода.
hStdOutput	Игнорируется, если dwFlags не включает флаг **STARTF_USESTDHANDLES.** Определяет дескриптор стандартного потока вывода.
hStdError	Игнорируется, если dwFlags не включает флаг **STARTF_USESTDHANDLES.** Определяет дескриптор стандартного потока ошибок.

Из всех полей этой структуры обязательным для заполнения является только cb — размер в байтах данной структуры. Остальные можно не заполнять, что обеспечит вид окна по умолчанию.

Параметр **lpProcessInformation** указывает на структуру **TProcessInformation**, или тождественного ему типа структуры **PROCESS_INFORMATION** API Windows. Из этой структуры приложение может получать информацию о выполнении нового процесса.

Объявление этого типа:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION,
  *LPPROCESS_INFORMATION;
```

Поля обозначают следующее:

hProcess	Возвращает дескриптор созданного процесса. Используется во всех функциях, осуществляющих операции с объектом процесса.
hThread	Возвращает дескриптор первого потока (нити) созданного процесса. Используется во всех функциях, осуществляющих операции с объектом потока.
dwProcessId	Возвращает глобальный идентификатор процесса. Значение доступно с момента создания процесса и до момента его завершения.
dwThreadId	Возвращает глобальный идентификатор потока. Значение доступно с момента создания потока и до момента его завершения.

Пример

В качестве примера порождения дочернего процесса функцией **CreateProcess** приведем код, который при щелчке на кнопке запускает консольный процесс архивации всех файлов текущего каталога:

```
STARTUPINFO StartInfo = { sizeof (TStartupInfo) };
PROCESS_INFORMATION ProcInfo;
LPCTSTR s;
StartInfo.cb = sizeof (StartInfo);
StartInfo.dwFlags = STARTF_USESHOWWINDOW;
StartInfo.wShowWindow = SW_SHOWNORMAL;
if (! CreateProcess(NULL, "arj a all *.*",
                    NULL, NULL, false,
                    CREATE_NEW_CONSOLE |
                    HIGH_PRIORITY_CLASS,
                    NULL, NULL, &StartInfo, &ProcInfo))
  ShowMessage("Ошибка " + IntToStr(GetLastError()));
else
 {
  if (WaitForSingleObject(ProcInfo.hProcess, 10000)
                                      == WAIT_TIMEOUT)
   ShowMessage("За 10 сек. архивация не завершена");
  CloseHandle(ProcInfo.hProcess);
 }
```

_crotl — циклический сдвиг кода символа влево

Осуществляет циклический сдвиг влево кода символа.

См. разд. «_rotl и другие функции циклического сдвига».

_crotr — циклический сдвиг кода символа вправо

Осуществляет циклический сдвиг вправо кода символа.

См. разд. «_rotl и другие функции циклического сдвига».

cscanf — форматированный ввод с клавиатуры

Вводит форматированные данные из входного потока (с клавиатуры).
См. разд. «scanf и другие функции форматированного ввода».

cwait и другие функции ожидания завершения порожденного процесса

Ожидают завершения порожденного процесса.

Заголовочный файл *process.h*.

Синтаксис

```
#include <process.h>
int cwait(int *statloc, int pid, int action);
int wait(int *statloc);
```

Описание

Функции **cwait и wait** позволяют организовать в программе ожидание завершения процесса, порожденного функциями семейств **exec... и spawn...**.

Функция **wait** обеспечивает ожидание завершения порожденного процесса или нескольких процессов. Окончания процессов, запущенных из этих порожденных процессов с вытеснением родителей, функция не ждет.

Если параметр **statloc** функции **wait** не **NULL,** то он указывает на целое, представляющее собой статус завершения порожденного процесса. При нормальном его завершении биты этого целого означают следующее:

биты 0-7	Нули
биты 8-15	Старшие разряды кода возврата порожденного процесса. Это то значение, которое передает программа в функцию exit или в оператор return функции main. Если порожденный процесс просто покинул main без оператора return, то значение этих битов не определено

При аварийном завершении порожденного процесса биты его статуса означают:

биты 0-7	1	неисправимая ошибка
	2	генерация исключения
	3	прерывание внешним сигналом
биты 8-15	Нули	

При нормальном завершении функция **wait** возвращает идентификатор порожденного процесса. При неудаче возвращается -1, а переменная **errno** равна EINTR — ненормальное завершение процесса, или **ECHILD** — порожденного процесса нет.

Функция **cwait** подобна **wait,** но дает большую гибкость. Помимо параметра **statloc,** рассмотренного выше, она имеет еще два параметра: **pid** и **action.** Если параметр **pid** задан равным 0, это означает, что происходит ожидание окончания любого порожденного процесса. Но в качестве значения pid может быть задан идентификатор конкретного порожденного процесса. Тогда происходит ожидание завершения именно указанного процесса.

Параметр **action** может принимать одно из двух значений: **WAIT_CHILD** — ожидание окончания указанного дочернего процесса, или **WAIT_GRANDCHILD** — ожидание окончания не только самого порожденного процесса, но и всех дочерних процессов, порожденных им.

См. пример организации ожидания порожденного процесса в описании функций семейства **spawn...**

Date и другие функции определения даты и времени

Определяют текущую дату и время.

Заголовочные файлы *SysUtils.hpp, DateUtils.hpp.*

Синтаксис

```
#include <SysUtils.hpp>
extern PACKAGE System::TDateTime __fastcall Date(void);
extern PACKAGE System::TDateTime __fastoall Now(void);
extern PACKAGE System::TDateTime __fastcall Time(void);
extern PACKAGE bool __fastcall
              IsToday(const System::TDateTime AValue);

#include <DateUtils.hpp>
extern PACKAGE System::TDateTime __fastcall Today(void);
extern PACKAGE System::TDateTime __fastcall Tomorrow(void);
extern PACKAGE System::TDateTime __fastcall Yesterday(void);
```

Описание

Функции **Date и Today** возвращают текущую дату в виде значения типа **TDateTime.** Часть возвращаемого значения, определяющая время, равна 0. Функция Time возвращает текущее время, а часть возвращаемого значения, определяющая дату, равна 0. Функция Now возвращает дату и время, объединяя возможности **Date и Time.**

Функции **Tomorrow и Yesterday** возвращают соответственно завтрашнюю и вчерашнюю дату, не указывая времени. Функция **IsToday** позволяет проверить, соответствует ли **AValue** сегодняшней дате.

Значения, возвращаемые описанными функциями, могут быть преобразованы в строку функциями **DateToStr. TimeToStr. DateTimeToStr. DateTimeToString** и др. Функции **DecodeDate** и др. позволяют выделить из даты отдельно день, месяц, год, час и т.п.

Примеры

Оператор

```
Editl->Text = "Сегодня " + DateToStr(Date());
```

помещает в окне **Edit1** текст вида "Сегодня 25.05.02".

Код

```
AnsiString Days[7] = {"Воскресенье", "Понедельник", "Вторник",
                      "Среда", "Четверг", "Пятница", "Суббота"};
Editl->Text = "Сегодня " + DateToStr(Date()) + " (" +
              Days[DayOfWeek(Date())-1] + ")";
```

помещает в окне **Editl** текст вида "Сегодня 25.05.2002 (Суббота)".

Если вместо функции **DayOfWeek,** использованной в последнем примере, применить функцию **DayOfTheWeek,** надо одновременно изменить последовательность дней в массиве **Days:**

```
AnsiString Days[7] = {"Понедельник", "Вторник", "Среда",
                      "Четверг", "Пятница", "Суббота", "Воскресенье"};
Editl->Text = "Сегодня " + DateToStr(Date()) + " (" +
              Days[DayOfTheWeek (Date())-1] + ")";
```

Оператор

```
Editl->Text = DateTimeToStr(Now());
```

записывает в окно Editl строку вида "25.05.2002 14:35:49". А оператор

```
Edit1->Text = DateTimeToStr(Date());
```

записывает в окно **Edit1** строку вида "25.05.2002".

Оператор

```
Edit1->Text = "Сейчас " + TimeToStr(Time());
```

помещает в окне **Editl** текст вида "Сейчас 14:38:26".

Если вам надо зафиксировать интервал времени, на протяжении которого выполняются какие-то длинные вычисления, вы можете сделать это следующим кодом:

```
TDateTime T1 = Time O;
<операторы, соответствующие длительному процессу>
Edit1->Text = "Прошло " + TimeToStr(Time() - T1);
```

В результате в окно **Editl** будет помещен текст вида "Прошло 0:01:25".

DateTimeToStr — преобразование даты в строку

Преобразует дату в строку.

См. разд. «DateToStr и другие функции преобразования даты и времени в строку».

DateTimeToString и другие функции форматированного преобразования даты и времени в строку

Преобразуют дату и время в строку по заданному формату.

Заголовочный файл *SysUtils.hpp.*

Синтаксис

```
extern PACKAGE void__fastcall
            DateTimeToString(AnsiString &Result,
                             const AnsiString Format,
                             System::TDateTime DateTime);
extern PACKAGE AnsiString__fastcall
            FormatDateTime(const AnsiString Format,
                           System::TDateTime DateTime);
```

Описание

Функция **DateTimeToString** заносит в параметр **Result,** а функция **FormatDateTime** возвращает строку, отображающую по формату, заданному строкой форматирования **Format,** дату и время, заданные параметром **DateTime** типа **TDateTime.** Строка форматирования поддерживает следующие спецификаторы:

Спецификатор	Действие спецификатора
c	Отображает дату в формате, соответствующем глобальной переменной ShortDateFormat ("день.месяц.год", год отображается двузначным числом) и время в формате, соответствующем глобальной переменной **LongTimeFormat** ("час:минута:секунда"). Если DateTime содержит только дату, то время не отображается.
d	Отображает день числом без предшествующего нуля: 1-31.
dd	Отображает день, причем всегда двузначным числом: 01-31.
ddd	Отображает день недели аббревиатурой, задаваемой глобальной переменной ShortDayNames. Для русифицированных Windows это обычно аббревиатуры: "Пн", "Вт", "Ср", "Чт", "Пт", " Сб", "Вс".

Спецификатор	Действие спецификатора
dddd	Отображает день недели полными наименованиями, задаваемыми глобальной переменной **LongDayNames**: "понедельник" "воскресенье".
ddddd	Отображает дату в формате, соответствующем глобальной переменной **ShortDateFormat**: "день.месяц.год" (год отображается двузначным числом).
dddddd	Отображает дату в формате, соответствующем глобальной переменной **LongDateFormat**: день, название месяца, год (четырехзначное число с последующими символами "г.").
ш	Отображает месяц числом без предшествующего нуля: 1-12. Если спецификатор m следует сразу за спецификатором h или hh, то он отображает не месяц, а минуты: 0-59.
mm	Отображает месяц двузначным числом: 01-12. Если спецификатор m следует сразу за спецификатором h или hh, то он отображает не месяц, а минуты: 00-59.
mmm	Отображает месяц его аббревиатурой, задаваемой глобальной переменной **ShortMonthNames**: "янв" "дек".
mmmm	Отображает месяц его полным именем, задаваемым глобальной переменной **LongMonthNames**: "Январь" "Декабрь".
yy	Отображает год двузначным числом: 00-99.
yyyy	Отображает год четырехзначным числом: 0000-9999.
h	Отображает час числом без предшествующего нуля: 0-23.
hh	Отображает час всегда двузначным числом: 00-23.
n	Отображает минуты числом без предшествующего нуля: 0-59.
nn	Отображает минуты всегда двузначным: 00-59.
s	Отображает секунды числом без предшествующего нуля: 0-59.
ss	Отображает секунды всегда двузначным числом: 00-59.
z	Отображает миллисекунды числом без предшествующего нуля: 0-999.
zzz	Отображает миллисекунды всегда трехзначным числом: 000-999.
t	Отображает время в формате, соответствующем глобальной переменной **ShortTimeFormat** ("час:минута").
tt	Отображает время в формате, соответствующем глобальной переменной **LongTimeFormat** ("час:минута:секунда").
am/pm	используется при 12-часовом отображении времени для записи символов "am" или "pm". регистр символов соответствует регистру, использованному в записи спецификатора.
a/p	Используется при 12-часовом отображении времени для записи символов "a" или "p". Регистр символов соответствует регистру, использованному в записи спецификатора.

Спецификатор	Действие спецификатора
ampm	Используется при 12-часовом отображении времени для записи символов, задаваемых глобальными переменными TimeAMString и TimePMString (в русифицированных Windows обычно пустые).
/	Отображает разделитель дат, заданный глобальной переменной DateSeparator (обычно "/").
:	Отображает разделитель времени, заданный глобальной переменной TimeSeparator (обычно":").
'xx'/"xx"	Символы, заключенные в одинарные или двойные кавычки, как и символы, отличные от других спецификаторов, переносятся в результирующую строку без форматирования.

Все спецификаторы могут записываться в строке форматирования в любом регистре. Если строка форматирования Format пуста, то отображение производится, как при спецификаторе "c".

Примеры

Ниже приведена таблица, содержащая строки форматирования и соответствующие им результирующие строки.

Формат	Строка результата
пустой	27.09.00 21:00:11
c	27.09.00 21:00:11
Сегодня c	Сегодня 27.09.00 21:00:11
Сегодня d mmmm yyy года	Сегодня 27 Сентябрь 2000 года
Сегодня dddddd	Сегодня 27 Сентябрь 2000 г.
Московское время h час. m мин.	Московское время 21 час. 0 мин.

Оператор

```
Editl->Text = FormatDateTime("Московское время h час. m мин.", Now());
```

заносит в окно редактирования Edit1 текст, который приведен в последней строчке таблицы.

Код

```
AnsiString S;
...
DateTimeToString(S, "Сегодня d mmmm yyy года", Now());
```

формирует строку вида "Сегодня 25 Май 2002 года".

DateToStr и другие функции преобразования даты и времени в строку

Преобразуют дату и время в строку.

Заголовочный файл *SysUtils.hpp.*

Синтаксис

```
extern PACKAGE AnsiString__fastcall
              DateTimeToStr(const System::TDateTime DateTime);
extern PACKAGE AnsiString__fastcall
              DateToStr(const System::TDateTime DateTime);
```

```
extern PACKAGE AnsiString__fastcall
            TimeToStr(const System::TDateTime DateTime);
```

Описание

Функция **DateTimeToStr** возвращает строку, отображающую дату и время, заданные параметром DateTime типа **TDateTime.**

Если параметр **DateTime** содержит только дату, то часть строки, связанная со временем, отсутствует. Аналогично, функция **DateToStr** возвращает строку, отображающую дату, а функция **TimeToStr** — строку, отображающую время.

Для отображения дат все функции используются формат преобразования, определяемый глобальной переменной ShortDateFormat, а для отображения времени — формат, определяемый глобальной переменной **LongTimeFormat.** Для русифицированных Windows эти форматы обычно имеют вид "день.месяц.год" и "час:минута:секунда". Более широкие возможности форматирования дат и времени предоставляют функции **Da'teTimeToString и FormatDateTime.**

См. примеры применения функций в разд. «Date и другие функции определения даты и времени».

DayOf и другие функции дешифрации дат и времени

Извлекают отдельные составляющие даты и времени.

Заголовочный файл *DateUtils.hpp.*

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE Word__fastcall
              DayOf (const System::TDateTime AValue);
extern PACKAGE Word__fastcall
              DayOfTheMonth(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
              HourOf(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
              HourOfTheDay(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
              MilliSecondOf(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
              MilliSecondOfTheSecond(constTDateTime AValue);
extern PACKAGE Word__fastcall
              MinuteOf(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
            MinuteOfTheHour(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
              MonthOf(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
            MonthOfTheYear(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
              SecondOf(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
           SecondOfTheMinute(const System::TDateTime AValue);
extern PACKAGE Word__fastcall
              YearOf (const System::TDateTime AValue);
```

Описание

Функции извлекают из значения **AValue** типа **TDateTime** отдельные составляющие даты и времени: год **(YearOf),** месяц **(MonthOf, MonthOfTheYear),** день месяца **(DayOf, DayOfTheMonth), час (HourOf, HourOfTheDay),** минуту **(MinuteOf, MinuteOfTheHour),** секунду **(SecondOf, SecondOfTheMinute),** миллисекунды **(MilliSecondOf, MilliSecondOfTheSecond).** Наличие пар одинаковых функций с разными именами объясняется стремлением к однозначности и точности имен. Например, день может обозначать день месяца, день недели, день года. В данном

случае речь идет о дне месяца. Поэтому соответствующая функция названа **DayOfTheMonth,** чтобы отличить ее, например, от функции **DayOfTheWeek,** возвращающей день недели. Аналогичные соображения относятся и к остальным функциям.

См. также функции **DecodeDate и DecodeTime.** которые позволяют получить те же значения, но не по отдельности, а группами, относящимися к дате и времени.

Примеры

Операторы

```
ShowMessage("Сейчас " + IntToStr(YearOf(Now())) + " год");
ShowMessage("Сегодня " + IntToStr(DayOf(Now())) + "-й день " +
            IntToStr(MonthOf(Now())) + "-го месяца " +
            IntToStr(YearOf(Now())) + " года");
```

отображают сообщения вида: "Сейчас 2002 год" и "Сегодня 8-й день 5-го месяца 2002 года".

DayOfTheMonth — дешифрация дня

Определяет день месяца.

См. разд. «DayOf и другие функции дешифрации дат и времени».

DayOfTheWeek и другие функции определения дня недели

Определяют день недели.

Заголовочные файлы *DateUtils.hpp, SysUtils.hpp.*

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE Word__fastcall
               DayOfTheWeek(const System::TDateTime AValue);
#include <SysUtils.hpp>
extern PACKAGE Word__fastcall
               DayOfWeek(const System::TDateTime DateTime);
```

Описание

Функции **DayOfTheWeek и DayOfWeek** возвращают день недели, соответствующий дате, заданной параметром **Date** типа **TDateTime.** День возвращается в виде целого числа от 1 до 7. В функции **DayOfTheWeek** 1 соответствует понедельнику, 7 - - воскресенью. Это согласуется со стандартом ISO 8601. В функции **DayOfWeek** 1 соответствует воскресенью, 7 — субботе. Различаются функции также модулями, в которых они описаны.

См. примеры применения функций в разд. «Date и другие функции определения даты и времени».

DayOfWeek — день недели

Определяет день недели.

См. разд. «DayOfTheWeek и другие функции определения дня недели».

DaysBetween и другие функции определения разности дней двух дат

Возвращают число дней между двумя значениями даты и времени.

Заголовочный файл *DateUtils.hpp.*

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE int__fastcall
                  DaysBetween (const System::TDateTime ANow,
                               const System::TDateTime AThen);
```

```
extern PACKAGE double__fastcall
                  DaySpan(const System::TDateTime ANow,
                          const System::TDateTime AThen);
```

Описание

Функции **DaysBetween и DaySpan** возвращают число суток между двумя значениями даты и времени **ANow и AThen** типа **TDateTime.** Функция **DaysBetween** возвращает число полных суток между двумя датами с учетом времени. А функция **DaySpan** возвращает действительное число, содержащее дробную часть, отображающую неполные сутки с учетом времени.

Примеры

Операторы

```
TDateTime T1 = EncodeDateTime(2002, 10, 5, 11, 25, 45, 300);
TDateTime T2 = EncodeDateTime(2002, 10, 6, 11, 24, 45, 300);
int i = DaysBetween(T2, T1);
double r = DaySpan(T2, T1) ;
```

зададут переменной i значение О, а переменной г значение 0,99930555555. В этом примере значения дат и времени T1 и T2 задаются с помощью функции **EncodeDateTime.** Различие между двумя значениями составляет 23 часа 59 минут. Поэтому функция **DaysBetween** возвращает 0, так как разность значений менее суток. А функция **DaySpan** возвращает число, близкое к единице.

DaySpan — разность дней двух дат

Возвращает число дней между двумя значениями даты и времени.

См. разд. «DaysBetween и другие функции определения разности дней двух дат».

DecodeDate и другие функции декодирования дат и времени типа TDateTime

Выделяют отдельные составляющие дат и времени.

Заголовочные файлы *SysUtils.hpp*, *DateUtlls.hpp*.

Синтаксис

```
#include <SysUtils.hpp>
extern PACKAGE void__fastcall
                 DecodeDate(const System::TDateTime DateTime,
                            Word SYear, Word &Month, Word &Day);
extern PACKAGE void__fastcall
                 DecodeTime(const System::TDateTime DateTime,
                            Word SHour, Word &Min, Word &Sec,
                            Word &MSec);

#include <DateUtils.hpp>
extern PACKAGE void__fastcall
                DecodeDateTime(const System::TDateTime DateTime,
                               Word &Year, Word &Month,
                               Word &Day, Word &Hour,
                               Word SMin, Word sSec,
                               Word &MSec);
```

Описание

Функции выделяют из значения параметра **DateTime** типа **TDateTime** отдельные составляющие даты и времени: **Year — год, Month** — месяц, Day — день, **Hour** — часы, Min — минуты, Sec — секунды, MSec — миллисекунды. Функции **DecodeDate и DecodeTime** декодируют соответственно только дату и время. Функция **DecodeDateTime** осуществляет декодирование и даты, и времени.

См. также функции, описанные в разд. «DayOf и другие функции дешифрации дат и времени» и позволяющие извлекать отдельные составляющие времени.

Примеры

Ниже приведен код, определяющий по значению переменной **Year_b,** в которой хранится год рождения какого-то сотрудника, его возраст Age в текущем году:

```
Word Year, Month, Day, Age, Year_b;
...
DecodeDate(Now(), Year, Month, Day);
Age = Year - Year_b;
```

После вызова процедуры **DecodeDate** в переменных **Year, Month** и **Day** хранятся соответственно текущий год, месяц и день. Значение Year используется для вычисления возраста.

Если вам надо зафиксировать с точность до секунды интервал времени, на протяжении которого выполняются какие-то длинные вычисления, вы можете сделать это следующим кодом:

```
Word Hour1, Hour2, Min1, Min2, Sec1, Sec2, MSec;
DecodeTime(Time(), Hourl, Min1, Sec1, MSec);
  <операторы, соответствующие длительному процессу>
DecodeTime(Time(), Hour2, Min2, Sec2, MSec);
Edit1->Text = "Прошло " + IntToStr(Hour2 - Hourl) + " часов, "
                        + IntToStr(Min2- Min1) + " минут, "
                        + IntToStr(Sec2- Sec1) + " секунд";
```

В результате в окно **Edit1** будет помещен текст вида: "Прошло ... часов, ... минут, ... секунд".

DecodeDateTime — декодирование дат и времени типа TDateTime

Выделяет отдельные составляющие даты и времени.

См. разд. «DecodeDate и другие функции декодирования дат и времени типа TDateTime».

DecodeTime — декодирование значения времени типа TDateTime

Выделяет отдельные составляющие времени.

См. разд. «DecodeDate и другие функции декодирования дат и времени типа TDateTime».

div и другие функции целочисленного деления

Целочисленное деление, возвращающее целое значение частного и остаток.

Заголовочные файлы *stdlib.h, Math.hpp.*

Синтаксис

```
#include <stdlib.h>
typedef struct {
   int  quot;        // частное
   int  rem;         // остаток
} div_t;

div_t div(int numer, int);

typedef struct {
   long int quot;    // частное
   long int rem;     // остаток
   } ldiv_t;

ldiv_t ldiv(long int numer, long int denom);
```

```
#include <Math.hpp>
extern PACKAGE void __fastcall
                        DivMod(int Dividend, Word Divisor,
                               Word &Result, Word &Remainder) ;
```

Описание

Функции осуществляют целочисленное деление двух целых чисел с подсчетом целого результата и остатка. В функциях **div** и **ldiv** значение **numer** делится на denom. Результат деления возвращается в структуру типа **div_t** или **Idiv_t** соответственно. Поле **quot** этой структуры содержит целое значение частного, а поле rem — целое значение остатка.

Функция **DivMod** осуществляет целочисленное деление **Dividend на Divisor,** возвращая результат **Result** (целую часть) и остаток **Remainder** (разность между **Dividend и Result * Divisor).**

Примеры

```
#include <stdlib.h>
div_t x;
x = div(StrToInt(Edit1->Text), StrToInt(Edit2->Text));
```

В этом примере осуществляется целочисленное деление чисел, введенных пользователем в окна **Edit1 и Edit2.** Пример результатов приведен в следующей таблице:

текст **Edit1**	текст Edit2	**x.quot**	**x.rem**
10	3	3	1
10	4	2	2
10	5	2	0

Еще пример. Оператор

```
DivMod(11, 4, Result, Remainder);
```

возвращает **Result = 2 и Remainder = 3.**

DivMod — целочисленное деление

Осуществляет целочисленное деление, возвращая результат и остаток. См. разд. «div и другие функции целочисленного деления».

EncodeDate и другие функции формирования типа TDateTime

Формируют дату и время из отдельных составляющих времени.

Заголовочные файлы *SysUtils.hpp, DateUtlls.hpp.*

Синтаксис

```
#include <SysUtils.hpp>
extern PACKAGE System::TDateTime__fastcall
            EncodeDate(Word Year, Word Month, Word Day);
extern PACKAGE bool__fastcall
            TryEncodeDate(Word Year, Word Month, Word Day,
                          System::TDateTime &Date);
extern PACKAGE System::TDateTime__fastcall
            EncodeTime(Word Hour, Word Min, Word Sec,
                       Word MSec);
extern PACKAGE bool__fastcall
            TryEncodeTime(Word Hour, Word Min, Word Sec,
                          Word MSec, System::TDateTime &Time);
```

```
#include <DateUtils.hpp>
extern PACKAGE System::TDateTime__fastcall
            EncodeDateTime(const Word Year,
                           const Word Month, const Word Day,
                           const Word Hour, const Word Kin,
                           const Word Sec, const Word MSec);
extern PACKAGE bool__fastcall
            TryEncodeDateTime(const Word Year,
                              const Word Month, const Word Day,
                              const Word Hour, const Word Min,
                              const Word Sec, const Word MSec,
                              System::TDateTime &Value);
```

Описание

Функции **EncodeDate, EncodeTime, EncodeDateTime** возвращают значение даты и времени типа **TDateTime.** сформированное из года **Year,** месяца **Month,** дня **Day,** часов **Hour,** минут **Min,** секунд Sec, миллисекунд MSec. Допустимые значения **Year** лежат в пределах от 1 до 9999, **Month** — от 1 до 12, **Day** — от 1 до 28-31 в зависимости от месяца, а для февраля — от того, високосный год, или нет. Допустимые значения **Hour** лежат в пределах от 0 до 24, Min и Sec — от 0 до 59, MSec — от 0 до 999. Если **Hour** = 24, то **Min, Sec, MSec** должны быть = 0 — это начало следующего дня.

Значение, возвращаемое функцией **EncodeDate,** имеет нулевую часть, описывающую время, а значение, возвращаемое функцией **EncodeTime,** имеет нулевую часть, описывающую дату. Функция **EncodeDateTime** возвращает и дату, и время.

При неверных значениях параметров перечисленные функции генерируют исключение **EConvertError.**

Функции, имя которых начинается с "Try", формируют дату и время аналогичным образом, но заносят результат по адресу своего последнего параметра **(Date, Time или Value).** Но основное их отличие в другом: при неверных значениях параметров они не генерируются исключение, а возвращают **false.**

Примеры

Приведенный ниже код переводит в тип **TDateTime** дату 25 октября 1917 года:

```
TDateTime T = EncodeDate(1917, 10, 25);
```

Аналогичный результат дадут операторы:

```
TDateTime T;
TryEncodeDate(1917, 10, 25, T);
```

Но если вы определите следующие переменные:

```
Word Year = 1917, Month = 10, Day = 32;
```

то оператор

```
TDateTime T = EncodeDate(Year, Month, Day);
```

вызовет генерацию исключения **(Day** не может равняться 32), а операторы

```
TDateTime T;
TryEncodeDate(Year, Month, Day, T);
```

исключения не сгенерируют. Так что при возможной ошибке в исходных данных вызов функции **EncodeDate** надо оформлять так:

```
try
{
  TDateTime T = EncodeDate(Year, Month, Day);
)
catch(EConvertError &)
{
  ShowMessage("Неверная дата");
}
```

А для функции **TryEncodeDate** аналогичный код имеет вид:

```
if (! TryEncodeDate(Year, Month, Day, T))
  ShowMessage("Неверная дата");
```

Приведенный ниже код переводит в тип **TDateTime** время 17 часов 45 минут: ,

```
TDateTime T = EncodeTime(17, 45, O, O);
```

Аналогичный результат дадут операторы:

```
TDateTime T;
TryEncodeTime(17, 45, O, O, T);
```

EncodeDateTime — формирование даты и времени типа TDateTime

Формирует дату и время из отдельных составляющих.
См. разд. «EncodeDate и другие функции формирования типа TDateTime».

EncodeTime — формирование времени типа TDateTime

Формирует время из отдельных его составляющих.
См. разд. «EncodeDate и другие функции формирования типа TDateTime».

EnsureRange — число, ближайшее к указанному

Возвращает число, ближайшее к указанному в заданном диапазоне.

Заголовочный файл *Math.hpp.*

Синтаксис

```
extern PACKAGE int___fastcall EnsureRange(const int AValue,
                          const int AMin, const int AMax);
extern PACKAGE___int64___fastcall EnsureRange(
                  const___int64 AValue, const___int64 AMin,
                  const___int64 AMax);
extern PACKAGE double___fastcall EnsureRange(
                    const double AValue, const double AMin,
                    const double AMax);
```

Описание

Перегруженные варианты функции **EnsureRange** возвращают для разных типов параметров число, ближайшее к **AValue** в диапазоне **AMin — AMax.** Если **AValue** находится внутри диапазона, возвращается значение **AValue.** Если **AValue < AMin,** возвращается **AMin.** Если **AValue > AMax,** возвращается **AMax.**

Например,

```
EnsureRange(5, 1, 10);  // возвращается 5
EnsureRange(11, 1, 10); // возвращается 10
EnsureRange(0, 1, 10);  // возвращается 1
```

exec... — функции выполнения порождаемых процессов

Порождают новый процесс.

Заголовочный файл *process.h.*

Синтаксис

```
#include <process.h>
int execl(char *path, char *argO *argl, ..., *argn, NULL);
int _wexecl(wchar_t *path, wchar_t *argO *argl, ..., *argn, NULL);

int execle(char *path, char *argO, *argl, ...,
           *argn, NULL, char **env);
```

```
int _wexecle(wchar_t *path, wchar_t *argO, *argl, ...,
             *argn, NULL, wchar_t **env);

int execlp(char *path, char *argO,*argl, ...,
           *argn, NULL);
int _wexeclp(wchar_t *path, wchar_t *argO,*argl, ...,
             *argn, NULL);

int execlpe(char *path, char *argO, *argl, ...,
            *argn, NULL, char **env);

int _wexeclpe(wchar_t *path, wchar_t *arg0, *argl,
              ..., *argn, NULL, wchar_t **env);

int execv(char *path, char *argv[]);
int _wexecv(wchar_t *path, wchar_t *argv[]);

int execve(char *path, char *argv[], char **env);
int _wexecve(wchar_t *path, wchar_t *argv[],
             wchar_t **env);

int execvp(char *path, char *argv[]);
int _wexecvp(wchar_t *path, wchar_t *argv[]);

int execvpe(char *path, char *argv[], char **env);
int _wexecvpe(wchar_t *path, wchar_t *argv[],
              wchar_t **env);
```

Описание

Функции **exec...** загружают в память и выполняют некоторую внешнюю программу **path,** называемую порожденным процессом. Вызванная программа замещает в памяти вызвавший ее процесс. Таким образом, родительский процесс завершается и начинается новый.

Различия между функциями семейства **exec...** определяются их суффиксами, которые обозначают следующее:

L	В процесс передается список указателей на аргументы **arg0, arg1, ..., argn.** Обычно используется, если число аргументов заранее известно.
v	В процесс передается указатель **argv[]** на массив указателей на аргументы **argO, arg1, ..., argn.** Обычно используется, если число передаваемых аргументов может изменяться.
p	Файл загружаемой программы ищется в каталогах, указанных в переменной окружения PATH. Если параметр **path** не содержит явного указания каталога, поиск ведется сначала в текущем каталоге, а затем в каталогах, указанных в PATH. Если функция не содержит суффикса "p", то файл ищется только в рабочем каталоге.
e	В порождаемый процесс может быть передан аргумент **env,** указывающий на окружение порождаемого процесса. Если функция не содержит суффикса "e", то порождаемый процесс наследует окружение родительского процесса.

Каждая из функций **exec...** должна передать в порождаемый процесс хотя бы один аргумент **(argO),** и по соглашению этот аргумент — копия **path.** Впрочем, передача другого значения не является ошибкой. Суммарная длина всех аргументов (не учитывая нулевых символов, но учитывая пробелы) не должна превышать 128 символов.

В функциях с суффиксом "l" аргументы перечисляются непосредственно в операторе вызова функции как указатели на строки с нулевым символом в конце. Количество аргументов не ограничено. Последним аргументом передается **NULL,** что является признаком окончания списка.

В функции с суффиксом "v" в качестве параметра передается указатель на массив произвольной длины, содержащий указатели на строки, являющиеся аргументами порождаемого процесса. Последним из указателей в массиве должен быть **NULL,** показывающий, что список аргументов завершился.

В функции с суффиксом "e" передается массив указателей **env** на строки, определяющие переменные окружения порождаемого процесса. Эти строки обычно имеют вид

```
<имя_переменной> = <значение>
```

Если env = **NULL,** то для функций с суффиксом "e" так же, как и для всех остальных функций, порождаемый процесс наследует окружение родительского процесса.

Файлы, открытые на момент вызова порождаемого процесса, остаются открытыми и для этого процесса. Однако в порожденный процесс не передается режим, в котором отрыты файла (текстовый или двоичный). Если режим отличается от принятого по умолчанию, то в порожденном процессе надо произвести его установку соответствующими функциями.

Поиск файла **path,** загружаемого функциями **exec...,** осуществляется следующим образом. Если в параметре **path** явно указано расширение файла или стоит точка, ищется файл такой, который задан. Если же расширение не задано, то сначала ищется файл такой, который задан. Если он не находится, к имени добавляется расширение .exe и поиск повторяется. Если файл опять не находится, к имени добавляется расширение **.com** и поиск повторяется. Функции без суффикса "p" ведут поиск файла только в текущем каталоге (если только каталог не задан явно в **path**). А функции с суффиксом "p" сначала ведут поиск в текущем каталоге, а затем — в каталогах, указанных в переменной окружения PATH.

Все функции возвращают 0 при успешной загрузке порожденного процесса, а при ошибке возвращают **–1.** В этом случае глобальная переменная **errno** может принимать значения **EACCES** — нарушение права доступа, **EMFILE** — слишком много открытых файлов, **ENOENT** — не найден путь или файл, **ENOEXEC** — ошибка формата, **ENOMEM** — не хватает памяти.

Если в программе требуется организовать ожидание завершения порожденного процесса, используются функции **cwait и wait.**

Рассмотренные функции могут найти достаточно ограниченное применение, поскольку они обеспечивают безвозвратную передачу управления из вызвавшего приложения в новое. И для возврата в исходное приложение надо принимать специальные меры: например, вызванное приложение в конце своей работы должно аналогичной функцией **exec...** вызвать первоначальное приложение. Зато у этих функций есть и большое преимущество — оверлэйная загрузка приложений. Новое приложение загружается в оперативную память на место вызвавшего его приложения. Соответственно сокращаются затраты памяти, так как не требуется держать в ней оба приложения.

Таким образом, сфера применения функции **exec...:**

- построение входного интерфейса к какому-то приложению, работающего только перед запуском этого приложения
- создание оверлэйных приложений, загружаемых в память по частям

Имеется родственное рассмотренному семейству функций семейство функций **spawn...** (см. разд. «spawn... — функции выполнения порождаемых процессов»), также решающее задачи порождения процессов, но обладающее более широкими возможностями.

Примеры

Оператор

```
if(execl("F1.exe","F1.exe", NULL))
   ShowMessage("Программа Fl.exe не выполнена");
```

завершает текущий процесс и передает управление программе с выполняемым файлом *Fl.exe.* Этот файл должен быть расположен в рабочем каталоге. Иначе функция **execl** вернет -1 и будет выдано сообщение функцией **ShowMessage.** Аналогичное сообщение будет выдано, если, например, для загрузки *Fl.exe* не хватает оперативной памяти.

Оператор

```
execlp("nc","nc", NULL);
```

передает управление программе Norton Commander (файл *nc.exe),* если только путь к этой программе указан в переменной окружения PATH.

Операторы

```
char * prog = "command.com";
execlp(prog,prog, NULL);
```

передают управление DOS, если только путь к файлу *command.com* указан в переменной окружения PATH.

Оператор

```
execlp("Winword","Winword", "F.doc",NULL))
```

запускает редактор Word и передает в него файл *F.doc.*

Вызов редактора Word можно оформить иначе:

```
char *arg[5] = {"Winword"}; // может принять до трех аргументов
arg[1] = "F1.doc";
arg[2] = "F2.doc";
execvp(arg[0],arg);
```

В массив **arg** при его объявлении заносится в качестве нулевого аргумента имя программы "Winword", а остальные четыре элемента массива по умолчанию получают значения **NULL.** После этого в элементы с индексами 1 и 2 заносятся имена передаваемых в Word файлов. Следующий элемент остается прежним — NULL. В результате функция **execvp** передаст управление программе Winword и загрузит в редактор два указанных файла.

fabs, fabsl — вычисление модуля

Функции вычисляют модуль действительного числа.

См. разд. «abs и другие функции вычисления модуля».

fgetc и другие функции ввода/вывода символа

Вводят символ из потока и выводят символ в поток.

Заголовочные файлы *stdio.h, conio.h.*

Синтаксис

```
ttinclude <stdio.h>
int fgetc (FILE *stream);
wint_t fgetwc(FILE * stream);

int fputc(int c, FILE *stream);
wint_t fputwc(wint_t c, FILE * stream);

int getc(FILE *stream);
wint_t getwc(FILE *stream);
```

```
int putc(int c, FILE *stream);
wint_t putwc(wint_t c, FILE *stream);

int getchar(void);
wint_t getwchar(void);
int _fgetchar(void);
wint_t _fgetwchar(void);

int putchar(int c);
wint_t putwchar(wint_t c);
int _fputchar(int c);
wint_t _fputwchar(wint_t c);

int ungetc(int c, FILE *stream);
wint_t ungetwc(wint_t c, FILE *stream);

#include <conio.h>
int getch(void);
int getche(void);
int ungetch(int ch);
```

Описание

Функции **fgetc и fgetwc** возвращают очередной символ из потока **stream.** Символ возвращается преобразованным в целое без знака. Если при чтении произошла ошибка или достигнут конец потока, возвращается EOF. В качестве потока **stream** может фигурировать стандартный входной поток **stdin,** который по умолчанию связан с клавиатурой. Функции используются в основном в консольных приложениях. **Например:**

```
#include <stdio.h>
#include <string.h>

int main(void)
{
 char s[80], ch;
 memset(s, '\0', sizeof(s) - 1);
 do s[strlen(s)] = fgetc(stdin);
 while (s[strlen(s)-1] != '.');
 puts(s);
 fflush(stdin);
 fgetc(stdin);
}
```

Функция **memset** очищает строку s. Затем *в* цикле **do ... while** функцией fgetc читаются вводимые пользователем символы и добавляются в строку s. Для определения позиции, в которую следует заносить символ, используется функция **strlen.** Когда занесенный символ оказывается точкой, чтение завершается и прочитанная строка выдается на экран функцией **puts.** Два последних оператора введены, чтобы предотвратить закрытие окна DOS до того, как пользователь посмотрит результат. Вызов функции **fflush** очищает входной поток, а последний вызов **fgetc** обеспечивает ожидание момента, когда пользователь нажмет *Enter.*

В приведенном примере функция **fgetc** работает так. При первом вызове выполнение приложения останавливается и программа ждет, пока пользователь вводит текст. После того как пользователь что-то написал и нажал *Enter,* функция **fgetc** возвращает первый введенный символ, выполнение программы возобновляется и поочередно читаются (уже без ожидания) все остальные введенные символы (если не встречается указанный в операторе **while** символ точки), включая символ перехода к новой строке. Все эти символы заносятся в s. Затем при очередном вызове **fgetc** выполнение опять приостанавливается и повторяется ожидание ввода и чтение новой строки. Последний оператор вызова **fgetc** обеспечивает завершение

приложения и, значит, закрытие окна DOS только после того, как пользователь нажмет клавишу *Enter*.

Функцию **fgetc** можно использовать в приложениях Windows или в консольных приложениях для чтения из любого потока, в частности, из текстового файла. Пусть, например, требуется найти в текстовом файле *input.txt* фрагмент, расположенный после символа "$" и кончающийся символом точки ".". Это можно сделать следующим кодом:

```
char s[256], ch;
memset(s, '\0', sizeof(s) - 1);
FILE *F;
F = fopen("input.txt", "rt");
while(fgetc(F) != '$');
do s[strlen(s)] = fgetc(F);
  while (s[strlen(s)-1] != '.');
ShowMessage(s);
fclose(F);
```

Первый оператор **while** обеспечивает просмотр в файле всех символов до тех пор, пока не встретится символ "$". Далее цикл **do ... while** обеспечивает занесение в s всех последующих символов файла вплоть до символа точки.

Функции **getc и getwc** идентичны рассмотренным функциям **fgetc** и **fgetwc.** Отличие заключается в том, что **getc и getwc** реализованы макросами, а не функциями.

Функции **getchar и getwchar** — макросы, используемые только для стандартного потока **stdin** и реализованные через вызовы **getc(stdin) и getwc(stdin).** Так что эти функции предназначены только для консольных приложений и их нельзя применять в приложениях Win32 с графическим интерфейсом. Аналогично **_fgetchar и _fgetwchar** — макросы, реализованные как **fgetc(stdin) и fgetwc(stdin)** и также предназначенные только для консольных приложений.

Функция **getch** читает символ непосредственно с клавиатуры, не отображая его на экране. При вызове функции выполнение приложения останавливается до тех пор, пока пользователь не нажмет какую-то (любую) клавишу. Символ, соответствующей нажатой клавише на экране не отобразится. Функция **getch** вернет нажатый символ, после чего выполнение приложения продолжится.

Эта функция может использоваться только в консольных приложениях Win32 без графического интерфейса. Она полезна, например, для организации ожидания действий пользователя при завершении консольного приложения. Ниже приведен пример подобной функции (она названа **MyClose),** которая предлагает пользователю нажать любую клавишу, чтобы закрыть окно DOS. Подобное завершение требуется почти всегда, чтобы пользователь мог спокойно просмотреть результаты работы программы и только после этого завершить ее. Для вывода русского текста используется функция **CharToOem.** Для ожидания нажатия пользователем клавиши используется функция **getch,** не отображающая нажатый символ на экране.

```
#include <stdio.h>
#include<system.hpp>
void MyClose(void)
{
 char S[] = "\nНажмите любую клавишу";
 CharToOem(S, S);
 puts(S);
 getch();
}
```

Подобную функцию можно вызывать в конце выполнения приложения оператором

```
MyClose();
```

Функция **getche** работает так же, как **getch,** но отображает на экране введенный символ. Эта функция полезна, например, при организации в консольном приложении простейшего диалога. Пусть вы хотите, чтобы пользователь в ответ на вопрос программы выбрал одну из трех возможностей: 1, 2 или 3. Тогда после задания вопроса вы можете поместить код:

```
switch (getche())
{
 case 'Ч': ...
           break;
 case '2': ...
           break;
 case '3': ...
           break;
 default : ...
}
```

В этом коде точками обозначены действия, которые надо осуществить при том или ином ответе пользователя.

Функции **fputc и fputwc** помещают в выходной поток **stream** символ с. Это может быть стандартный выходной поток **stdout,** связанный по умолчанию с экраном, или текстовый файл. Например, следующий код, вставленный в консольное приложение:

```
char *S = "Hello world";
for(int i=0; i <strlen(S); i++)
 { fputc(S[i], stdout); Sleep(500); }
```

обеспечит посимвольный постепенный (с задержками в 500 миллисекунд — см. описание функции **Sleep)** вывод строки S. Впрочем, для консольных приложений разумнее использовать функции **putchar и putwchar,** которые выводят символ в выходной поток **stdout** и только в консольных приложениях могут использоваться. Так что приведенный в примере вызов **fputc** целесообразнее заменить на:

```
putchar(S[i]);
```

Функции **putc и putwc** являются полными аналогами **fputc и fputwc,** но реализованы в виде макросов. Аналогично **_fputchar и _fputwchar** — макросы, реализованные как **fputc(c, stdout) и fputwc(c, stdout).** Все эти макросы предназначены только для консольных приложений без графического интерфейса Windows.

Функции **ungetc и ungetwc** заносят символ с в поток stream, открытый для чтения. В этом их уникальность, так как иными способами занести символ во входной (а не выходной) поток невозможно. Функция ungetch заносит символ в буфер клавиатуры. После вызова этих функций последующий вызов любой из рассмотренных ранее функций чтения прочтет именно этот символ, искусственно занесенный во входной поток. Например, ранее в описании функции fgetc приводился пример выделения фрагмента текстового файла, следующего за символом "$". Но если надо, чтобы сам символ "$" входил в выделенный фрагмент, достаточно в этом примере перед циклом **do** вставить оператор:

```
ungetc('$', F);
```

Тогда очередной вызов функции **fgetc** введет повторно этот символ.

Основное применение **ungetc и ungetch** — возврат символа, прочитанного из стандартного входного потока **stdin** (с клавиатуры) в некоторую переменную ch, обратно во входной поток: **ungetc(ch, stdin)** или **ungetch(ch),** чтобы его можно было повторно прочитать функцией **getc** или **fread.** Можно использовать также функции **ungetc и ungetch** для имитации нажатия пользователем определенной клавиши.

Необходимо учитывать, что функции **ungetc, ungetwc, ungetch** могут занести во входной поток только один символ. Повторный их вызов без промежуточного вызова **getc** просто сотрет символ, занесенный предыдущим вызовом. Учтите также, что вызовы функций **fflush, fsetpos, rewind** удаляют символ, занесенный в память функциями **ungetc, ungetwc, ungetch.**

_fgetchar — ввод символа из потока

Вводит символ из входного потока.
См. разд. «fgetc и другие функции ввода/вывода символа».

fgets — ввод строки из потока

Вводит строку из указанного потока.
См. разд. «fputs и другие функции ввода/вывода строк».

fgetwc — ввод симвбла из потока

Вводит символ из указанного потока.
См. разд. «fgetc и другие функции ввода/вывода символа».

_fgetwchar — ввод символа из потока

Вводит символ из входного потока.
См. разд. «fgetc и другие функции ввода/вывода символа».

fgetws — ввод строки из потока

Вводит строку из указанного потока.
См. разд. «fputs и другие функции ввода/вывода строк».

FindClose — завершение поиска файлов

Завершает поиск файлов, начатый функцией FindFirst.
См. разд. «FindFirst и другие функции поиска файлов из библиотеки C++Builder».

FindExecutable — функция API Windows

Возвращает имя и путь приложения, связанного с указанным файлом.

Модуль *ShellAPI*

Определение

```
HINSTANCE FindExecutable(
 LPCTSTR lpFile,         // строка с именем файла документа
 LPCTSTR lpDirectory,    // строка каталога по умолчанию
 LPTSTR lpResult         // строка с именем выполняемого файла
);
```

Описание

Функция **FindExecutable** позволяет получить имя выполняемого файла .exe, связанного с файлом, указанным параметром **IpFile.** Параметр **IpDirectory** определяет каталог по умолчанию. Оба параметра являются указателями на строки с нулевым символом в конце. Параметр **IpResult** является указателем на буфер в виде строки с нулевым символом в конце, в который функция заносит имя и путь приложения, связанного с файлом **IpFile.**

При успешном завершении функция **FindExecutable** возвращает значение, большее 32. Если возвращено меньшее значение, это свидетельствует об ошибке.

Пример

Операторы

```
char APchar[254];
FindExecutable("Doc.doc",NULL,APchar);
```

приведут к тому, что в массив **APchar** будет занесено имя приложения, связанного с файлом типа **Doc.doc,** например:

```
C:\\PROGRAM FILES\\MICROSOFT OFFICE\\OFFICE\\WINWORD.EXE
```

Правда, при этом файл **Doc.doc** должен существовать в доступном каталоге.

Успешность завершения функции **FindExecutable** можно проверить с помощью функции **GetLastError**. Если она возвращает значение не большее 32, значит произошла ошибка. Эту проверку могут, например, осуществить следующие операторы:

```
int i = GetLastError();
if (i <= 32)
 ShowMessage("Программа не найдена. Код ошибки "+IntToStr(i));
```

findfirst и другие стандартные функции поиска файлов

Обеспечивают поиск файлов, удовлетворяющих заданному шаблону и имеющих указанные атрибуты.

Заголовочный файл *dir.h.*

Синтаксис

```
#include <dir.h>
struct ffblk {                    // Для Win32
   long           ff_reserved;    // Зарезервировано
   long           ff_fsize;       // Размер файла
   unsigned long  ff_attrib;      // Атрибуты файла
   unsigned short ff_ftime;       // Время создания
   unsigned short ff_fdate;       // Дата создания
   char           ff_name[256];   // Имя файла
   };
struct _wffblk {                    // Для Unicode
    long           ff_reserved;    // Зарезервировано
    long           ff_fsize;       // Размер файла
    unsigned long  ff_attrib;      // Атрибуты файла
    unsigned short ff_ftime;       // Время создания
    unsigned short ff_fdate;       // Дата создания
    wchar_t        ff_name[256];   // Имя файла
};

int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);
int  _wfindfirst(const wchar_t *pathname,
                 struct _wffblk *ffblk, int attrib);

int findnext(struct ffblk *ffblk);
int _wfindnext(struct _wffblk *ffblk);
```

Описание

Функции **findfirst и findnext,** а также их аналоги для Unicode **_wfindfirst и _wfindnext,** обеспечивают поиск файлов, удовлетворяющих шаблону и имеющих указанные атрибуты.

Начинается поиск вызовом функции **findfirst.** Параметр pathname определяет путь и шаблон искомых файлов. Например, если **pathname** = "c:\\test*.*", то будут искаться все файлы в каталоге c:\test\. Если **pathname** = "c:\\test*.tmp", то в каталоге c:\test\ будут искаться файлы с расширением .tmp. А если **pathname** = "*.tmp", то файлы с расширением .tmp будут искаться в текущем каталоге.

Параметр **Attr** определяет флаги атрибутов, которые должны иметь искомые файлы:

FA_RDONLY	файл только для чтения
FA_HIDDEN	невидимый файл
FA_SYSTEM	системный файл
FA_LABEL	метка диска
FA_DIREC	каталог
FA_ARCH	архивный файл

Флаги могут объединяться. Например, если **Attr = FA_RDONLY | FA_HIDDEN,** то будут искаться невидимые файлы только для чтения.

Функция возвращает 0, если файл, удовлетворяющий условиям поиска, найден. Если файл не найден или если задан ошибочный путь к файлу, возвращается -1, переменная **errno** принимает значение **ENOENT,** а переменная **_doserrno** принимает значение **ENMFILE,** если файл не найден, или **ENOENT,** если имеется ошибка в пути или в имени файла.

Если файл найден, то сведения о нем заносятся в поля структуры типа **ffblk,** определяемой параметром **ffblk.** В поле **ff_name** этой структуры можно найти имя файла вместе с его расширением. Например, "Test.txt". Поля **ff_ftime** и **ff_fdate** содержат информацию о дате и времени создания файла. Оба поля представляют собой 16-битовые структуры, биты которых содержат:

биты	ff_ftime	ff_fdate
0-4	секунды, деленные на 2 (т.е. 1 — это 2 сек.)	день
5-10	минуты	месяц
11-15	час	год, отсчитанный от 1980 (например, 22 — это 2002 г.

Аналогично использует биты структура **ftime,** объявленная в *io.h.*

Поле **ff_attrib** структуры **ffblk** содержит атрибуты файла. Определить тип найденного файла можно комбинированием соответствующего флага с полем **ffblk** по операции И (&). Если файл имеет данный атрибут, то результат этой операции будет ненулевой. Например, чтобы узнать, является ли найденный файл системным, надо записать выражение

```
(ffblk.ff_attrib & FA_SYSTEM)
```

Это выражение вернет не 0, если файл системный.

Таким образом, вызов **findfirst** может найти первый файл, удовлетворяющий условиям поиска, или убедиться, что ни одного такого файла нет. Продолжение поиска осуществляется вызовом функции **findnext** и передачей в нее в качестве параметра **ffblk** той же записи, которая передавалась в **findfirst.** Если **findfirst** вернет 0, значит, нашелся еще один файл, удовлетворяющий условиям поиска. Информация об этом файле занесется в ту же запись **ffblk,** после чего можно снова вызывать **findfirst** для поиска следующего файла. Если **findfirst** вернет ненулевое значение, значит, больше нет файлов, удовлетворяющих условиям поиска.

Альтернативный способ поиска файлов обеспечивают функции библиотеки C++Builder, рассмотренные в разд. «FindFirst и другие функции поиска файлов из библиотеки C++Builder».

Примеры

Приведенный ниже код обеспечивает отображение в окне **Memo1** всех подкаталогов и файлов, содержащихся в каталоге *c:\My*.

```
#include <dir.h>
struct ffblk F;
int ires = findfirst("c:\\My\\*.*", &F, FA_DIREC | FA_ARCH);
Memo1->Clear();

while (! ires)
{
  if (F.ff_attrib & FA_DIREC)
   Memo1->Lines->Add("Каталог \t" + AnsiString(F.ff_name));
  else Memo1->Lines->Add("Файл \t" + AnsiString(F.ff_name));
  ires = findnext(&F);
}
```

Ниже приведен пример функции **myexitl** (см. разд. 1.12.2), удаляющей в рабочем каталоге все временные файлы с расширением .tmp:

```
#include <dir.h>
#include <stdio.h>

void myexit1 (void)
{
 struct ffblk ffblk;
 int D;
 D = findfirst("*.tmp", sffblk, 0);
 while (!D)
 {
  remove(ffblk.ff_name);
  D = findnext(&ffblk);
 }
}
```

FindFirst и другие функции поиска файлов из библиотеки C++Builder

Обеспечивают поиск файлов, удовлетворяющих заданному шаблону и имеющих указанные атрибуты.

Заголовочный файл *SysUtils.hpp*.

Синтаксис

```
#include <SysUtils.hpp>
struct TSearchRec
{
    int Time;          // Время создания файла
    int Size;          // Размер файла в байтах
    int Attr;          // Атрибуты файла
    AnsiString Name;   // Имя файла
    int ExcludeAttr;
    int FindHandle;
    _WIN32_FIND_DATAA FindData;
};

extern PACKAGE int _fastcall
    FindFirst(const AnsiString Path, int Attr, TSearchRec &F);
extern PACKAGE int _fastcall FindNext(TSearchRec SF);
extern PACKAGE void _fastcall FindClose(TSearchRec &F);
```

Описание

Функции **FindFirst, FindNext и FindClose** обеспечивают поиск файлов, удовлетворяющих заданному шаблону и имеющих указанные атрибуты.

Начинается поиск вызовом функции **FindFirst**. Параметр **Path** определяет путь и шаблон искомых файлов. Например, если **Path** = "c:\\test*.*", то будут искаться все файлы в каталоге c:\test\. Если **Path** = "c:\\test*.tmp", то в каталоге c:\test\ будут искаться файлы с расширением .tmp. А если **Path** = "*.tmp", то файлы с расширением .tmp будут искаться в текущем каталоге.

Параметр **Attr** определяет флаги атрибутов, которые должны иметь искомые файлы:

Константа	Значение	Пояснение
faReadOnly	$00000001	файл только для чтения
i284faHidden	$00000002	невидимый файл
faSysFile	$00000004	системный файл
faVolumeID	$00000008	идентификатор диска
faDirectory	$00000010	каталог
faArchive	$00000020	архивный файл
faAnyFile	$0000003F	любой файл

Флаги могут объединяться. Например, если **Attr = faReadOnly + faHidden,** то будут искаться невидимые файлы только для чтения.

Функция возвращает 0, если файл, удовлетворяющий условиям поиска, найден. В противном случае возвращается код ошибки.

Если файл найден, то сведения о нем заносятся в поля записи типа **TSearchRec,** определяемой параметром F. В поле Name этой записи можно найти имя файла вместе с его расширением. Например, "Test.txt". В поле **Time** заносится дата и время создания файла. Это время в формате DOS. Его можно перевести в значение типа **TDateTime** функцией **FileDateToDateTime,** а если требуется перевести его в строку, то к полученному значению можно затем применить функцию **DateTimeToStr.** Таким образом, выражение вида

```
DateTimeToStr(FileDateToDateTime(F.Time))
```

вернет дату и время создания файла в виде строки.

Поле **Attr** записи F содержит атрибуты файла. Определить тип найденного файла можно комбинированием соответствующего флага с полем **Attr** по операции ИЛИ (|). Если файл имеет данный атрибут, то результат этой операции будет больше 0. Например, чтобы узнать, является ли найденный файл системным, надо записать выражение

```
(F.Attr & faSysFile)
```

Это выражение вернет **true,** если файл системный.

Таким образом, вызов **FindFirst** может найти первый файл, удовлетворяющий условиям поиска, или убедиться, что ни одного такого файла нет. Продолжение поиска осуществляется вызовом функции **FindNext** и передачей в нее в качестве параметра F той же записи, которая передавалась в **FindFirst.** Если **FindNext** вернет 0, значит нашелся еще один файл, удовлетворяющий условиям поиска. Информация об этом файле занесется в ту же запись F, после чего можно снова вызывать **FindNext** для поиска следующего файла. Если **FindNext** вернет ненулевое значение, значит больше нет файлов, удовлетворяющих условиям поиска. В этом случае надо вызвать процедуру **FindClose** с тем же параметром F. Эта процедура завершает поиск и освобождает ресурсы, выделенные для него.

Альтернативный способ поиска файлов обеспечивают функции стандартной библиотеки C, рассмотренные в разд. «findfirst и другие стандартные функции поиска файлов».

Примеры

Следующие операторы осуществляют поиск всех файлов и подкаталогов текущего каталога и выводят результаты в окно редактирования **Memo1**:

```
TSearchRec sr;
Memo1->Clear();
if (! FindFirst("*.*", faAnyFile I faDirectory, sr))
{
 Memo1->Lines->Add(sr.Name+", размер: " + IntToStr(sr.Size));
 while (! FindNext(sr))
  Memo1->Lines->Add(sr.Name+", размер: " + IntTcStr(sr.Size));
}
FindClose(sr);
```

Следующий пример удаляет из каталога *c:\MyTemp* все файлы с расширением .tmp. Подобные функции полезно применять при зачистке мусора (см. разд. 1.12).

```
TSearchRec sr;
int D;
D = FindFirst("c:\\MyTemp\\*.tmp", faAnyFile, sr) ;
while (!D)
{
  DeleteFile(sr.Name);
  D = FindNext(sr);
 }
 FindClose(sr);
```

Приведем более сложный пример. Следующий обработчик щелчка на кнопке **Button1** обеспечивает в текущем каталоге и во всех его подкаталогах поиск и удаление файлов с расширением .tds:

```
void__fastcall TForm1::Button1Click(TObject *Sender)
{
 AnsiString CurDir = GetCurrentDir();
 TSearchRec sr;
 if (! FindFirst("*.*", faAnyFile I faDirectory, sr))
 {
  if(! CompareText(ExtractFileExt(sr.Name),".tds"))
     DeleteFile(sr.Name);
  else if((sr.Attr == faDirectory)&&(sr.Name !=".")
          &&(sr.Name !=".."))
  {
   SetCurrentDir(ExtractFileDir(sr.Name));
   Button1Click(Sender);
   SetCurrentDir(CurDir);
  }
  while (FindNext(sr) == 0)
  {
   if(! CompareText(ExtractFileExt(sr.Name) , ".tds"))
      DeleteFile(sr.Name);
   else if((sr.Attr == faDirectory)&&(sr.Name !=".")
           &&(sr.Name !=".."))
   {
     SetCurrentDir(sr.Name);
     Button1Click(Sender);
     SetCurrentDir(CurDir);
   }
  }
 }
 FindClose(sr);
}
```

Приведенная функция использует рекурсивный вызов самой себя при переходе в подкаталоги.

FindNext — продолжение поиска файлов

Продолжает поиск файлов, начатый функцией FindFirst.

См. разд. «FindFirst и другие функции поиска файлов из библиотеки C++Builder».

findnext — стандартная функция продолжения поиска файлов

Обеспечивает продолжение поиска файлов, начатого функцией findfirst.

См. разд. «findfirst и другие стандартные функции поиска файлов».

FindWindow — функция API Windows

Функция API Windows, возвращает дескриптор окна, заданного класса и с заданным текстом.

Модуль *winuser.*

Объявление

```
HWND FindWindow(const char *lpClassName, const char *lpWindowName);
```

Описание

Функция **FindWindow** возвращает дескриптор окна, заданного класса **lpClassName** и с заданным текстом заголовка окна **lpWindowName.** Параметр **lpClassName** указывает на строку с нулевым конечным символом, содержащую имя класса. Параметр **IpWindowName** указывает на строку с нулевым конечным символом, содержащую имя окна (это свойство **Caption** формы, отображаемое в полосе заголовка окна). Если этот параметр равен **NULL,** то считается, что под критерий поиска подходит любое окно указанного класса.

Если поиск прошел успешно, то функция возвращает дескриптор окна, имеющего указанное имя класса и имя окна. В противном случае возвращается **NULL.**

Эту функцию легко использовать, если вы знаете имя класса искомого окна. Например, если ваше приложение вызвало другое приложение, созданное вами самими, то вы знаете имя класса формы этого другого приложения. Тогда вы можете, например, с помощью кода

```
HWND H = FindWindow("TForm1","Приложение 2");
```

определить дескриптор окна приложения, класс формы которого **TForml,** а значение свойства **Caption** формы - "Приложение 2".

Если же приложение, которым вы хотите управлять, создано не вами, то текст полосы заголовка вы легко можете увидеть, выполнив его, а вот имя класса вам неизвестно. Одна из возможностей узнать имя класса какого-то приложения — воспользоваться поставляемой вместе с C++Builder программой WinSight 32 (файл ...\Program Files\Borland\CBuilder6\Bin\ws32.exe). Запустите интересующее вас приложение, затем запустите WinSight 32, выполните команду Spy | Find Window и вы увидите список всех окон, зарегистрированных в данный момент в Windows. Лучше, чтобы в этот момент у вас было бы открыто не очень много окон, чтобы проще было найти среди них нужное.

В списке, который вы увидите, для каждого окна будут указаны среди прочей информации имя класса в фигурных скобках "{ }" и заголовок окна — последний элемент данных в строке каждого окна. Например, запустив «Калькулятор», вы можете с помощью WinSight 32 найти, что имя класса окна этого приложения — «SciCalc». Следовательно, определить в своем приложении дескриптор открытого приложения «Калькулятор» вы можете оператором:

```
HWND H = FindWindow("SciCalc","Калькулятор");
```

(см. пример в разд. «PostMessage»).

Другой способ найти дескриптор окна — воспользоваться функцией **GetNextWindow** API Windows.

FloatToStr — преобразование действительного числа в строку

Преобразует действительное число в строку.

Заголовочный файл *SysUtils.hpp*

Синтаксис

```
extern PACKAGE AnsiString __fastcall FloatToStr(Extended Value);
```

Описание

Функция **FloatToStr** преобразует действительное значение **Value** в строку. Параметр **Value** — действительная константа или выражение. При преобразовании используется основной числовой формат с 15 значащими цифрами. Это обычно слишком много, если преобразуются числа с большим числом значащих цифр или иррациональные числа.

Если преобразовываемое выражение окажется не числом, функция вернет значение "NAN". Если преобразовываемое значение превышает по модулю величину, допустимую для объявленного типа данных, функция вернет значение "INF" (бесконечность) или "–INF" (минус бесконечность).

Если вам требуется управление форматом представления чисел, лучше использовать функции **FloatToStrF или FloatToText.**

Примеры

Если S — строка, а R — действительное число или выражение, то оператор

```
S = FloatToStr(R);
```

даст следующие результаты при разных значениях R:

R	S
-3	**-3**
5.1E20	5,1E20
sqrt(2)	**1,4142135623731**
1E5000	**INF**

FloatToStrF — преобразование действительного числа в строку

Преобразовывает действительное число в строку, используя заданный формат, точность и число цифр.

Заголовочный файл *SysUtils.hpp.*

Синтаксис

```
enum TFloatFormat ( ffGeneral, ffExponent, ffFixed,
                    ffNumber, ffCurrency };
extern PACKAGE AnsiString __fastcall FloatToStrF(
                          Extended Value, TFloatFormat Format,
                          int Precision, int Digits);
```

Описание

Функция **FloatToStrF** преобразовывает действительное значение **Value** в строку, используя заданный формат **Format,** точность **Precision** и число цифр **Digits.**

Параметр **Value** — действительная константа или выражение. Возможные значения параметра **Format** означают следующее:

ffGeneral	Основной числовой формат
ffExponent	Научный формат
ffFixed	Формат с фиксированной запятой
ffN umber	Числовой формат
ffCurrency	Монетарный формат

Подробное описание всех этих форматов см. в разд. 3.1.3.4, посвященном типу **TFloatFormat**.

Параметр **Precision** определяет точность преобразовываемого значения. Значение **Precision** должно быть не более 7 при преобразовании типа **Single,** не более 15 для **Double,** не более 18 для **Extended.**

Параметр **Digits** совместно с **Format** определяет форматирование строки. Подробнее см. в описании **TFloatFormat** в разд. 3.1.3.4.

Если преобразовываемое выражение окажется не числом, функция вернет значение "NAN". Если преобразовываемое значение превышает по модулю величину, допустимую для объявленного типа данных, функция вернет значение "INF" (бесконечность) или "–INF" (минус бесконечность).

Примеры

Если S — строка, а R — действительное число или выражение, то оператор

```
S = FloatToStrF(R,ffGeneral,7,0);
```

даст следующие результаты при разных значениях R:

R	S
-3	–3
5.1E20	5,1E20
sqrt(2)	1,414214
1E5000	INF

Эти результаты аналогичны получаемым при использовании функции **FloatToStr.** но для корня из 2 число цифр результата не превышает заданного значения 7: 1,414214. Как правило, формат **ffGeneral** с ограниченным числом цифр наиболее удачен для большинства применений.

floor, Floor, floorl — округление действительного числа

Округляют действительное число до целого значения.

См. разд. «Ceil и другие функции округления действительных чисел».

fmod, fmodl — функции вычисления остатка

Вычисляют остаток целочисленного деления.

Заголовочный файл *math.h.*

Синтаксис

```
#include <math.h>
double fmod(double x, double y) ;
long double fmodl(long double x, long double y);
```

Описание

Функции осуществляют деление двух действительных чисел — x на y и возвращают остаток от деления.

Пример

```
#include <math.h>
Edit3->Text = fmod(StrToFloat(Edit1->Text),
                   StrToFloat(Edit2->Text));
```

В этом примере осуществляется деление чисел, введенных пользователем в окна **Editl и Edit2.** Остаток от деления заносится в окно **EditS.** Пример результатов приведен в следующей таблице:

текст Editl	текст Edit2	текст Edit3
10	3	1
10	3,5	3
11	3,5	0,5

Format — форматирование строки аргументов

Возвращает строку, содержащую аргументы, отформатированные по заданному формату.

Заголовочный файл *SysUtils.hpp.*

Синтаксис

```
#include <SysUtils.hpp>
extern PACKAGE AnsiString __fastcall
    Format(const AnsiString Format,
           const System::TVarRec* Args, const int Args_Size);
```

Описание

Функция форматирует набор аргументов, заданный массивом **Args** типа **TVarRec.** Параметр **Args_Size** указывает последний индекс этого массива, т.е. он на 1 меньше числа аргументов. Параметр **Format** задает строку форматирования, синтаксис которой приведен в разд. 3.1.2.3.

Для задания параметров Args и **Args_Size** удобно использовать макросы **EXISTINGARRAY и OPENARRAY** (см. разд. 2.11.3) как в приведенных ниже примерах.

Функция возвращает отформатированную строку. При ошибках преобразования генерируется исключение **EConvertError.**

Примеры

```
AnsiString s;
TVarRec Args[3] = {11, -1.1e+08, 0.00011};
s = Format("Результат: %d %g %f", Args, 2);
```

В этом примере объявлена переменная s, в которую производится запись результатов форматирования, и массив **Args,** в который занесены форматируемые числа. Обратите внимание на то, что размер массива равен 3, а в функцию **Format** передается в качестве размера число 2 — максимальное значение индекса, на 1 меньшее размера. Если бы нужно было форматировать не все три, а, например, только два первых числа массива, то в качестве размера можно было бы передать 1.

Строка форматирования в этом примере сначала заносит в строку s текст "Результат: ", а затем записывает значения элементов массива по форматам "%d", "%g", "%f", оставляя между ними пробелы (пробелы между спецификациями в строке форматирования переносятся в строку результата).

Более сложное форматирование:

```
s = Format("%d %g %f %0:10d %10g %10.5f", Args, 2);
```

Здесь сначала в строку s заносятся значения элементов массива по форматам "%d", "%g", "%f". Затем спецификация "%0:10d" сбрасывает индекс массива на О, приводя к повторному форматированию элементов массива. Повторно они форматируются с заданной шириной поля 10, а последнее число еще и с заданной точностью 5.

Для ссылки на массив аргументов можно было бы воспользоваться макросом **EXISTINGARRAY:**

```
s = Format("%d %g %f %0:10d %10д %10.5f",EXISTINGARRAY(Args));
```

Можно было бы и не создавать заранее массива аргументов, а сформировать его непосредственно в вызове функции **Format** с помощью макроса **OPENARRAY:**

```
s = Format("%d %g %f %0:10d %10g %10.5f",
                OPENARRAY(TVarRec,( 11, -1.1e+08, 0.00011)));
```

FormatDateTime — преобразование даты и времени в строку

Преобразует дату и время в строку по заданному формату.

См. разд. «DateTimeToString и другие функции форматированного преобразования даты и времени в строку».

fprintf и другие функции форматированного вывода

Выводят форматированные данные в выходной поток, в файл, в буферный массив.

Заголовочные файлы *stdio.h, conio.h.*

Синтаксис

```
#include <stdio.h>
int printf(const char *format[, argument, ...]);
int wprintf(const wchar_t *format[, argument, ...]);
int vprintf(const char *format, va_list arglist);
int vwprintf(const wchar_t * format, va_list arglist);

int fprintf(FILE *stream,
            const char *format[, argument, ...]);
int fwprintf(FILE *stream,
             const wchar_t *format[, argument, ...]);
int vfprintf(FILE *stream, const char *format,
             va_list arglist);
int vfwprintf(FILE *stream, const wchar_t *format,
              va_list arglist);

int sprintf(char *buffer,
            const char *format[, argument, ...]);
int swprintf(wchar_t *buffer,
             const wchar_t *format[, argument, ...]);
int vsprintf(char *buffer, const char *format,
             va_list arglist);
int vswprintf(wchar_t *buffer, const wchar_t *format,
              va_list arglist);

#include <conio.h>
int cprintf(const char *format[, argument, ...]);
```

Описание

Все описанные ниже функции осуществляют форматированный вывод в виде строки или последовательности строк на экран, в файл или в буферный массив

в памяти. При этом производится вывод произвольного числа числовых, символьных и иных аргументов **argument,** причем числовые значения преобразуются в текстовое представление с заданными характеристиками отображения. Параметр **format** указывает строку форматирования. Она определяет текст формируемой строки и содержит спецификаторы, записываемые после символа "%", которые указывают формат включения в строку аргументов. Подробнее вы можете посмотреть полное описание строки форматирования в разд. 3.1.3.1. А для приведенных ниже примеров достаточно знать, что спецификатор "%i" отображает целые числа, спецификатор "%g" — действительные, спецификатор "%s" - строки.

Функции **printf и wprintf** обеспечивают форматированный вывод в стандартный поток вывода **stdout** (см. разд. 2.10.3). Параметр **format** указывает строку форматирования, которая применяется к множеству аргументов **argument,** расположенных в вызовах функций после строки форматирования.

Функции возвращают число выведенных байтов. Если при выводе происходят ошибки, возвращается значение **EOF.**

Функции находят применение в основном в консольных приложениях, в которых поток **stdout** соответствует выводу на экран. Например:

```
#include <stdio.h>
void main(void)
{
 int I = 10;
 double A = 5.1;
 char S[] = "string";
 printf("Output with printf\n");
 printf("I = %i S = %s A = %g \n", I, S, A);
 getchar();
}
```

Приведенный код обеспечивает вывод на экран двух строк:

```
Output with printf
I = 10 S = string A = 5.1
```

Для упрощения в этом коде использованы английские тексты. О выводе в консольных приложениях русских текстов см. в разд. «CharToOem, CharToOemBuff — перевод строки в текст DOS».

Функция **cprintf** аналогична по синтаксису **printf** и тоже обеспечивает вывод на экран. Строка пишется или прямо в память экрана, или выводится посредством вызова BIOS в зависимости от значения глобальной переменной **_directvideo.** В отличие от других функций вывода, **cprintf** не транслирует символ "\n" в пару символов "\r\n" и не разворачивает символ табуляции "\t" в пробелы.

Если желательно применять функции **printf и wprintf** для записи в файл, то предварительно надо перенаправить поток **stdout.** Однако для форматированного вывода в файл естественнее использовать функции **fprintf и fwprintf.** Они работают так же, как **printf,** а параметр **stream** указывает поток или файл, в который осуществляется вывод. Например, операторы:

```
FILE *F;
int I = 10;
double A = 5.1;
char S[] = "строка";
F = fopen("output.txt", "wt");
fprintf(F, "Вывод функцией fprintf\n");
if (fprintf(F, "I = %i S = %s A = %g \n", I, S, A) == EOF)
 ShowMessage("Ошибка записи в файл");
fclose(F);
```

запишут в файл *output.txt* две строки:

```
Вывод функцией fprintf
I = 10 S = string A = 5.1
```

Функции **sprintf и swprintf** аналогичны рассмотренным ранее, но формируют строку вывода не на экране, не в файле, а в массиве символов **buffer.** В конец строки заносится нулевой символ. В дальнейшем эта строка может быть выведена на экран или в файл. Может она также использоваться в различных диалоговых окнах, в метках и т.п.

Пример формирования строки:

```
char buffer[100];
int I = 10;
double A = 5.1;
char S[] = "строка";
sprintf(buffer, "Вывод функцией sprintf\n");
sprintf(buffer+strlen(buffer),
        "I = %i S = %s A = %g \n", I, S, A);
```

Первый вызов **sprintf** заносит соответствующую строку в **buffer.** А в последующих вызовах приемником строки может указываться адрес **buffer+strlen(buffer).** Это обеспечит добавление новой строки после прежнего текста.

В дальнейшем сформированный в буфере текст может быть выведен в файл функцией **fputs**. Например:

```
FILE *F;
F = fopen("output.txt", "wt");
fputs(buffer, F);
fclose(F);
```

Имеются определенные преимущества подобного предварительного формирования текста в памяти. Прежде всего, функция неформатированного вывода **fputs** работает быстрее, чем функция форматированного вывода **fprintf**. К тому же, вывод осуществляется один раз и может заменить целую последовательность вызовов **fprintf.** Эта последовательность заменяется последовательностью вызовов **sprintf** и однократным вызовом **fputs,** а запись в память, естественно, осуществляется намного быстрее, чем запись в файл. Еще одно преимущество предварительного формирования текста в памяти проявляется в том случае, если с файлом поочередно работает несколько приложений. В этом случае требуется, чтобы каждое из них захватывало файл на минимальное время. Формирование текста без открытия файла и последующее открытие его только на время выполнения быстрой функции **fputs** решает эту задачу.

Варианты рассмотренных функций с именами, начинающимися с символа "v", работают так же, как описанные выше, но в них передается не список аргументов, а указатель на список типа **va list** (см. разд. 1.7.5). Это позволяет вам создавать собственную функцию вывода, принимающую произвольное число аргументов. Ниже приведен пример такой функции pr, в точности воспроизводящей функцию **fprintf.**

```
#include <stdio.h>
#include <stdarg.h>

int pr(FILE *F,char *fmt, ...)
{
   va_list arg;
   int cnt;
   va_start(arg, fmt);
   cnt = vfprintf(F,fmt, arg);
   va_end(arg);
   return(cnt);
}
```

Вызов такой функции не отличается от вызова **fprintf**(кроме имени) и работает она точно так же. Но, конечно, реально имеет смысл создавать собственную функцию вывода только в том случае, если она должна чем-то отличаться от **fprintf:** заносить какие-то разделительные линии между строками, какие-то надписи, осуществлять табулированный вывод и т.п.

fputc — вывод символа в поток

Выводит символ в указанный поток.

См. разд. «fgetc и другие функции ввода/вывода символа».

_fputchar — вывод символа в поток

Выводит символ в выходной поток.

См. разд. «fgetc и другие функции ввода/вывода символа».

fputs и другие функции ввода/вывода строк

Вводят строки из потоков и выводят строки в потоки.

Заголовочные файлы *stdio.h, conio.h.*

Синтаксис

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
int fputws (const wchar_t *s, FILE *stream);

int puts(const char *s);
int _putws(const wchar_t *s) ;

char *fgets(char *s, int n, FILE *stream);
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);

char *gets(char *s);
wchar_t *_getws(wchar_t *s);

#include <conio.h>
int cputs(const char *str);
char *cgets(char *str);
```

Описание

Функции **fputs и fputws** выводят в выходной поток **stream** строку символов *s* с нулевым символом в конце. В выходной поток не добавляется символ перехода на новую строку. Концевой нулевой символ исходной строки в поток не копируется.

В случае успешной записи в поток функции возвращают неотрицательное значение. При ошибке возвращается EOF.

В качестве потока **stream** может фигурировать стандартный выходной поток **stdout,** который по умолчанию связан с экраном. Функции используются в основном в консольных приложениях. Например:

```
#include <stdio.h>
void main(void)
{
  fputs("Hello world\n", stdout);
}
```

Для вывода русских текстов в консольных приложениях надо использовать функцию **CharToOem.**

Можно в качестве потока указать текстовый файл. Например:

```
#include <stdio.h>
...
```

```
FILE *F;
F = fopen("output2.txt", "wt");
if (fputs("Привет !!!\n", F) < 0)
 ShowMessage("Ошибка записи в файл");
fclose(F);
```

Функции **puts** и **_putws** также выводят строку, но только в выходной поток **stdout.** При этом в выходной поток заносится после вывода символ перехода на новую строку. Функции предназначены для применения в консольных приложениях. Если их почему-то требуется использовать в приложениях Windows, стандартный поток **stdout** должен быть перенаправлен.

Функция **cputs** тоже обеспечивает вывод на экран. Строка пишется или прямо в память экрана, или выводится посредством вызова BIOS в зависимости от значения глобальной переменной **_directvideo.** В отличие от других функций вывода, **cputs** не транслирует символ "\n" в пару символов "\r\n" и не разворачивает символ табуляции "\t" в пробелы. Функция возвращает последний выведенный символ. В приложениях Windows функцию применять не следует.

Функции **fgets** и **fgetws** читают строку символов из потока **stream** в строку s. Чтение завершается, если прочитано n - 1 символов или если раньше встретился символ перехода на новую строку. Так что значение n должно задаваться исходя из допустимого размера s. Прочитанный из потока символ перехода на новую строку заносится в s. В конец s заносится нулевой символ. Функции возвращают указатель на прочитанную строку s или **NULL,** если достигнут конец файла или произошла ошибка чтения. Следующий пример демонстрирует чтение по строкам текстового файла *input.txt*:

```
#include <stdio.h>
...
char S[256];
FILE *F;
F = fopen("input.txt", "rt");
while(fgets(S, 256, F))
 <операторы обработки строки S>
fclose(F);
```

Функции **gets** и **_getws** читают в s строку из стандартного входного потока **stdin,** который по умолчанию связан с клавиатурой. Чтение происходит до появления во входном потоке символа новой строки. Этот символ в s заменяется нулевым символом. Функция возвращает указатель на прочитанную строку s или возвращает **NULL** в случае ошибки. При чтении не осуществляется проверка длины строки s. Так что при вводе недопустимо длинной строки могут быть непредсказуемые последствия. С этой точки зрения надежнее описанная выше функция **fgets,** которая может контролировать длину читаемой строки. При использовании **fgets** для чтения из стандартного входного потока в нее в качестве **stream** надо передать **stdin.** Функции **gets** и **_getws** предназначены в основном для использования в консольных приложениях. При использовании в приложениях Windows поток **stdin** надо перенаправить.

Функция **cgets** тоже обеспечивает чтение с клавиатуры в строку str. Чтение происходит до появления во входном потоке символов новой строки и возврата каретки — комбинации "CR/LF", или до достижения максимально допустимого для чтения числа символов (см. ниже). Если прочитаны символы конца строки и возврата каретки, их комбинация заменяется в **str** нулевым символом.

Максимальное число читаемых символов должно быть занесено до вызова **cgets** в **str[0].** После окончания чтения функция заносит в **str[1]** число реально прочитанных символов. Сами прочитанные символы начинаются с позиции **str[2].** В случае успешного завершения чтения функция возвращает указатель на первый из прочитанных символов, т.е. на **str[2].**

Функция cgets неприменима для приложений Windows.

Помимо рассмотренных функций для ввода и вывода строк могут использоваться функции форматированного ввода и вывода, описанные в разд. «scanf и другие функции форматированного ввода» и «fprintf и другие функции форматированного вывода».

fputwc — вывод символа в поток

Выводит символ в указанный поток.
См. разд. «fgetc и другие функции ввода/вывода символа».

_fputwchar — вывод символа в поток

Выводит символ в выходной поток.
См. разд. «fgetc и другие функции ввода/вывода символа».

fputws — вывод строки в поток

Выводит строку в указанный поток.
См. разд. «fputs и другие функции ввода/вывода строк».

free — освобождение памяти

Функция освобождает блок памяти.
См. разд. «malloc и другие функции динамического распределения памяти».

frexp, frexpl, Frexp — выделение мантиссы

Разбивают число на мантиссу и показатель степени 2.

Заголовочные файлы *math.h* и *math.hpp*.

Синтаксис

```
#include <math.h>
double frexp(double x, int *exponent);
long double frexpl(long double x, int *exponent);

tinclude <math.hpp>
extern PACKAGE void__fastcall
        Frexp(Extended X, Extended &Mantissa, int &exponent);
```

Описание

Функции разделяют значение X на мантиссу и показатель степени 2. Т.е. число представляется в виде: **X = Mantissa • $2^{exponent}$**. Показатель степени заносится в параметр **exponent,** а мантисса в функциях **frexp и frexpl** возвращается как значение функции, а в функции **Frexp** заносится в параметр **Mantissa.**

Пример

```
#include<math.h>
double number;
int exponent;

number = StrToFloat(Edit1->Text);
Edit3->Text = frexp(number, &exponent);
Edit2->Text = exponent;
```

В этом примере число, вводимое пользователем в окне **Edit1**, разбивается на показатель степени **exponent,** отображаемый в окне **Edit2,** и мантиссу, отображаемую в окне **Edit3**.

Пример результатов приведен в следующей таблице:

x	exponent	Возвращаемый результат
1	1	0,5
2	2	0,5
10	4	0,625
1.5	1	0.75

fscanf — форматированный ввод из файла

Вводит форматированные данные из файла.
См. разд. «scanf и другие функции форматированного ввода».

fwprintf — форматированный вывод в файл

Выводит форматированные данные в файл.
См. разд. «fprintf и другие функции форматированного вывода».

fwscanf — форматированный ввод из файла

Вводит форматированные данные из файла.
См. разд. «scanf и другие функции форматированного ввода».

get — функция-элемент ifstream

Вводит символы из входного потока.

Класс *ifstream*

Объявления

```
char get();
bool get(char);
void get(char *, int n, char delim);
```

Описание

Метод get представляет собой функции-элементы класса входного потока **ifstream.** Он вводит символы из файла, связанного с потоком. Метод имеет три приведенные выше модификации.

Первая модификация

Функция get без аргументов вводит одиночный символ из указанного потока (даже, если это символ разделитель) и возвращает этот символ в качестве значения вызова функции. Этот вариант функции get возвращает **EOF,** когда в потоке встречается признак конца файла.

Следующий код использует функцию get без аргумента, чтобы построчно читать и обрабатывать весь текст файла:

```
char s[80], c;
ifstream infile("Test.dat");
if(!infile)
{
 ShowMessage("Файл не удается открыть");
 return;
}
int i = 0;
while((c = infile.get()) != EOF)
{
 if(c == '\n')
```

```
 {
  // занесение нулевого символа в конец строки
  s[i] = 0;
  // обработка строки

  i = 0;
 }
 // формирование строки
 else s[i++] = c;
}
// закрытие файла
 infile.close();
```

Здесь символы файла поочередно читаются в символьную переменную c. Если прочитанный символ не является символом перевода строки "\n", то символ добавляется в строку s. Если же символ равен "\n", то в конец строки заносится нулевой символ, строка подвергается какой-то обработке, после чего начинает формироваться следующая строка. Отметим, что этот код имеет один недостаток: если символу конца файла не предшествует символ перевода строки, то последняя строка оказывается без завершающего нулевого символа и остается необработанной. Нетрудно придумать дополнение кода, которое ликвидировало бы этот недостаток.

Функцию **get()** удобно использовать для поиска в файле какого-то ключевого символа. Например, цикл поиска в файле символа "$" можно организовать следующим образом:

```
while((c = infile.get()) != EOF)
  {
  if(c == '$') break;
  if(c == '$') ...
  }
```

Вторая модификация

Второй вариант функции-элемента get с символьным аргументом вводит очередной символ из входного потока (даже, если этот символ разделитель) и сохраняет его в символьном аргументе. Этот вариант функции get возвращает ложь, когда встречается признак конца файла; в остальных случаях этот вариант функции get возвращает ссылку на тот объект потока, для которого вызывалась функция-элемент get.

При использовании этого варианта функции **get** приведенные ранее примеры можно оставить практически без изменений, переписав только заголовки структур **while:**

```
while(infile.get(c))
```

Третья модификация

Третий вариант функции-элемента get принимает три параметра: символьный массив s, максимальное число символов n и ограничитель **delim** (по умолчанию символ перевода строки "\n"). Этот вариант читает символы из входного потока до тех пор, пока не достигается число символов, на 1 меньше указанного максимального числа n, или пока не считывается ограничитель. Затем для завершения введенной строки в символьный массив, используемый в качестве буфера программы, помещается нулевой символ. Ограничитель в символьный массив не помещается, а остается во входном потоке (он будет следующим считываемым символом). Таким образом, результатом второго подряд использования функции get явится пустая строка, если только ограничитель не удалить из входного потока.

Приведенный ранее пример чтения всего файла по строкам в данном случае реализуется проще:

```
char s[80];
 ifstream infile("Test.dat");
 if(!infile)
```

```
  {
   ShowMessage("Файл не удается открыть");
   return;
  }
 while(!infile.eof())
  {
   infile.get(s,80);
   infile.get();
   // обработка строки
   ...
  }
 // закрытие файла
  infile.close();
```

В данном случае третий аргумент в вызове get не указан. Значит подразумевается по умолчанию ограничитель "\n" и каждый вызов get читает одну строку (подразумевается, что ее длина не более 80 символов). Обратите внимание на то, что после оператора

```
infile.get(s,80);
```

добавлен оператор

```
infile.get();
```

Этот оператор удаляет из потока ограничитель. Если этого не сделать, программа зациклится.

Функция get с тремя параметрами не всегда удобна, поскольку оставляет ограничитель в потоке, и для повторного вызова функции его приходится убирать отдельным оператором. Часто более удобна другая функция — **getline.**

Get8087CW — доступ к управляющему слову FPU

Обеспечивает доступ к управляющему слову FPU.

См. разд. «_control87 и другие функции доступа к управляющему слову FPU».

getc — ввод символа из потока

Вводит символ из указанного потока.

См. разд. «fgetc и другие функции ввода/вывода символа».

getch — ввод символа из потока

Вводит символ из входного потока без эхо на экране.

См. разд. «fgetc и другие функции ввода/вывода символа».

getchar — ввод символа из потока

Вводит символ из входного потока.

См. разд. «fgetc и другие функции ввода/вывода символа».

getche — ввод символа из потока

Вводит символ из входного потока с отображением на экране.

См. разд. «fgetc и другие функции ввода/вывода символа».

GetExceptionMask и другие функции доступа к маскам исключений

Обеспечивают доступ к маскам исключений управляющего слова FPU.

Заголовочный файл *math.hpp.*

Объявления

```
#include <math.hpp>
enum TFPUException {exInvalidOp, exDenormalized, exZeroDivide,
                    exOverflow, exUnderflow, exPrecision};
typedefSet<TFPUException, exInvalidOp, exPrecision>
                                                TFPUExceptionMask;

extern PACKAGE TFPUExceptionMask____fastoall
                                        GetExceptionMask(void);
extern PACKAGE TFPUExceptionMask____fastcall
                SetExceptionMask(void TFPUExceptionMask Mask);
```

Описание

Функции **GetExceptionMask и SetExceptionMask** обеспечивают доступ к маскам исключений управляющего слова FPU (см. разд. 1.9.3). Обе функции возвращают текущую маску исключений. Но функция **SetExceptionMask,** помимо этого, устанавливает заданную маску **Mask.**

Маска представляет множество значений, соответствующих определенным битам управляющего слова FPU и маскирующих генерацию различных видов исключений при выполнении операций с плавающей запятой:

exInvalidOp	Маска исключений при ошибочных операциях
exDenormalized	Маска исключений ненормализованных операций
exZeroDivide	Маска исключений деления на нуль
exOverflow	Маска исключений переполнения
exUnderflow	Маска исключений потери порядка
exPrecision	Маска исключений точности

Если какое-то из перечисленных значений **TFPUExceptionMask** включено в маску, генерация соответствующего исключения блокируется. В этих случаях ошибка выполнения операции приведет к тому, что в качестве результата будет выдано значение одной из предопределенных в C++Builder 6 констант: **Infinity** (положительная бесконечность), **NegInfinity** (отрицательная бесконечность), **NaN** (нецифровое значение).

Обычно по умолчанию маска имеет вид: **[exDenormalized, exUnderflow, exPrecision],** т.е. блокируется генерация исключений, связанных с ненормализованными операциями, потерей порядка и точностью.

Программную установку масок управляющего слова имеет смысл проводить, если вы решили запретить генерацию каких-то видов исключений. Это, в частности, приходится делать, при использовании некоторых пакетов. Например, при использовании для трехмерной графики OpenGL надо запретить генерацию всех исключений.

Примеры

Пусть вы объявили глобальную переменную **Mask:**

```
#include <math.hpp>
TFPUExceptionMask Mask;
```

Тогда следующий оператор очищает все маски исключений, сохраняя в **Mask** прежние значения масок:

```
Mask = SetExceptionMask(Mask.Clear());
```

В результате при всех ошибках выполнения операций с плавающей запятой будут генерироваться соответствующие исключения.

Следующий оператор устанавливает в 1 все маски исключений:

```
Mask = SetExceptionMask(Mask << exInvalidOp <<
                        exDenormalized << exZeroDivide <<
                        exOverflow << exUnderflow <<
                        exPrecision);
```

В результате никакие исключения, связанные с операциями с плавающей запятой, генерироваться не будут.

Если в дальнейшем после того фрагмента кода, в котором вы маскировали исключения, требуется удалить маски, достаточно выполнить оператор:

```
Mask = SetExceptionMask(Mask);
```

Он восстановит маски, запомненные предыдущим оператором, и запомнит текущие маски.

getline — функция-элемент ifstream

Вводит строку символов из потока.

Класс *ifstream*

Объявления

```
void getline(char *s, int n);
void getline(char *s, int n, char delim);
```

Описание

Метод **getline** представляет собой функцию-элемент класса входного потока **ifstream.** Он вводит строку символов из файла, связанного с потоком.

Функции **getline** принимает три параметра: символьный массив s, максимальное число символов n и ограничитель **delim** (по умолчанию символ перевода строки "\n"). Функция читает символы из входного потока до тех пор, пока не достигается число символов, на 1 меньше указанного максимального числа n, или пока не считывается ограничитель. Затем для завершения введенной строки в символьный массив, используемый в качестве буфера программы, помещается нулевой символ. Ограничитель в символьный массив не помещается и удаляется из входного потока. В этом основное отличие функции **getline** от варианта функции **get**, читающего строки символов. Функция get оставляет разделитель во входном потоке и его приходится удалять из него, чтобы прочитать следующую строку. Так что в этом отношении функция **getline** удобнее.

Пример

Ниже приведен пример чтения файла по строкам с помощью функции **getline.** Сравнив его с аналогичным примером, приведенным в описании функции get, вы можете увидеть преимущества функции **getline.**

```
char s[80], c;
ifstream infile("Test.dat");
if (!infile)
(
 ShowMessage("Файл не удается открыть");
 return;
}
while(!infile.eof())
  (
   infile.getline(s,80);
   // обработка строки
   if (s[0] != 0) Label1->Caption = s;
  }
// закрытие файла
 infile.close();
```

GetLastError — функция API Windows

Функция API Windows, возвращает значение кода последней ошибки данного потока.

Объявление

```
DWORD GetLastError(VOID);
```

Описание

Функция **GetLastError** возвращает значение кода последней ошибки данного потока. Эти коды индивидуальны для каждого потока и другие потоки их не изменяют. Различные функции задают эти коды функцией **SetLastError.**

Функция **GetLastError** должна вызываться сразу после возврата функции, ошибку которой вы хотите проверить. Это связано с тем, что некоторые функции вызывают при своем успешном завершении **SetLastError(0),** что уничтожает код ошибки.

Большинство функций API Win32 устанавливают код последней ошибки при аварийном завершении, хотя некоторые устанавливают этот код в случае успешного выполнения. Обычно функции задают такие коды, как **FALSE, NULL,** OxFFFFFFFF или **-1.**

Код ошибки — 32-битовое значение. Наиболее значимый бит 31. Бит 29 зарезервирован для кода, определяемого приложением. В системных кодах этот бит не используется. Таким образом, этот бит показывает, что код был определен приложением. Это гарантирует отсутствие пересечения между вашими и системными кодами.

Чтобы получить строку сообщения об ошибке по коду, используйте функцию **FormatMessage.** Полный список кодов ошибок имеется в заголовочном файле WINNT.H в Win32 SDK.

Пример

Проверку ошибки при выполнении какой-то функции можно осуществить, например, так:

```
int i = GetLastError();
if (i <= 32)
  ShowMessage("Код ошибки "+IntToStr(i));
```

GetNextWindow — функция API Windows

Функция API Windows, определяет дескриптор следующего или предыдущего окна в Z-последовательности.

Объявление

```
HWND GetNextWindow(
    HWND hWnd,
    UINT wCmd    );
```

Описание

Функция **GetNextWindow** определяет дескриптор следующего или предыдущего окна в Z-последовательности. Если указано дочернее окно, то поиск ведется среди дочерних окон.

Параметр **hWnd** — дескриптор окна, от которого начинается отсчет. Параметр **wCmd** определяет направление поиска. Если **wCmd = GW_HWNDNEXT,** то ищется следующее окно, находящееся ниже. Если **wCmd = GW_HWNDPREV,** то ищется предыдущее окно, находящееся выше.

Если окно найдено, то возвращается его дескриптор. Если следующего или предыдущего окна нет (в зависимости от значения **wCmd),** то возвращается 0. Развернутую информацию об ошибке можно получить вызовом функции **GetLastError.**

Использование функции GetNextWindow дает тот же самый результат, что и вызов функции **GetWindow** со значениями параметра **GW_HWNDNEXT** или GW_HWNDPREV.

См. пример в разд. «GetWindowText — функция API Windows».

GetPrecisionMode и другие функции управления точностью

Обеспечивают доступ к битам управления точностью управляющего слова FPU.

Заголовочный файл *Math.hpp.*

Синтаксис

```
enum TFPUPrecisionMode {pmSingle, pmReserved,
                        pmDouble, pmExtended}

extern PACKAGE TFPUPrecisionMode __fastcall
                                    GetPrecisionMode(void);
extern PACKAGE TFPUPrecisionMode __fastcall
          SetPrecisionMode(const TFPUPrecisionMode Precision);
```

Описание

Функции **GetPrecisionMode и SetPrecisionMode** обеспечивают доступ к битам управления точностью управляющего слова FPU (см. разд. 1.9.3). Обе функции возвращают значение, соответствующее текущим значениям этих битов, в виде перечислимого тип **TFPUPrecisionMode.** Но функция **SetPrecisionMode,** помимо этого, устанавливает заданное значение битов управления точностью **Precision.**

Биты управления точностью могут принимать следующие значения:

pmSingle	точность, соответствующая типу Single
pmReserved	не используется
pmDouble	точность, соответствующая типу Double
pmExtended	точность, соответствующая типу Extended

К битам управления точностью, как и ко всем остальным битам управляющего слова FPU, можно также получить доступ с помощью функций Get8087CW. **Set8087CW. _control87. _controlfp.**

GetRoundMode и другие функции управления округлением

Обеспечивают доступ к битам управления округлением управляющего слова FPU.

Заголовочный файл *Math.hpp.*

Синтаксис

```
enum TFPURoundingMode {rmNearest, rmDown, rmUp, rmTruncate)

extern PACKAGE TFPURoundingMode __fastcall GetRoundMode(void);
extern PACKAGE TFPURoundingMode __fastcall
                SetRoundMode(const TFPURoundingMode RoundMode);
```

Описание

Функции **GetRoundMode и SetRoundMode** обеспечивают доступ к битам управления округлением управляющего слова FPU (см. разд. 1.9.3). Обе функции возвращают значение, соответствующее текущим значениям этих битов, в виде перечислимого тип **TFPURoundingMode.** Но функция **SetRoundMode,** помимо этого, устанавливает заданное значение битов управления округлением **RoundMode.**

Биты управления округлением могут принимать следующие значения:

rmNearest	округление к ближайшему значению
rmDown	округление к меньшему значению, т.е. округление в сторону отрицательной бесконечности
rmUp	округление к большему значению, т.е. округление в сторону положительной бесконечности
rmTruncate	усечение младших разрядов, т.е. округление в сторону нуля

К битам управления округлением, как и ко всем остальным битам управляющего слова FPU, можно также получить доступ с помощью функций **Get8087CW. Set8087CW. _control87, controlfp.**

gets — ввод строки из потока

Вводит строку из стандартного потока.
См. разд. «fputs и другие функции ввода/вывода строк».

getwc — ввод символа из потока

Вводит символ из указанного потока.
См. разд. «fgetc и другие функции ввода/вывода символа».

getwchar — ввод символа из потока

Вводит символ из входного потока.
См. разд. «fgetc и другие функции ввода/вывода символа».

GetWindow — функция API Windows

Определяет дескриптор окна, находящегося с указанным в указанном соотношении (по Z-последовательности или по последовательности владельцев).

Объявление

```
HWND GetWindow(
    HWND hWnd,
    UINT uCmd
  );
```

Описание

Функция **GetWindow** определяет дескриптор окна **hWnd,** находящегося с указанным в указанном родственном отношении **uCmd.** Эта функция позволяет найти при использовании различных значений **uCmd** дескрипторы любых открытых окон Windows.

Параметр **hWnd** — дескриптор окна, по отношению к которому определяются соотношения родства. Параметр **uCmd** определяет соотношения родства и может принимать значения:

GW_CHILD	Определяется дескриптор дочернего окна на вершине Z-последовательности, если указано родительское окно. В противном случае возвращается дескриптор NULL. Проверяются только дочерние окна указанного окна, но не его потомков.
GW_HWNDFIRST	Определяется дескриптор окна того же типа, находящегося в верху Z-последовательности.

GW_HWNDLAST	Определяется дескриптор окна того же типа, находящегося в внизу Z-последовательности.
GW_HWNDNEXT	Определяется дескриптор следующего окна в Z-последовательности.
GW_HWNDPREV	Определяется дескриптор предыдущего окна в Z-последовательности.
GW_OWNER	Определяется дескриптор окна, являющегося владельцем указанного.

Если окно найдено, то возвращается его дескриптор. Если требуемого окна не существует, то возвращается 0. Развернутую информацию об ошибке можно получить вызовом функции **GetLastError.**

GetWindowText — функция API Windows _ _

Функция API Windows, копирует текст, связанный с указанным окном, в указанный буфер.

Объявление

```
int  GetWindowText(
     HWND hWnd,          // дескриптор окна, содержащего текст
     LPTSTR lpString,    // буфер
     int  nMaxCount      // максимальное число, символов
    );
```

Описание

Функция API Windows **GetWindowText** копирует текст, связанный с указанным окном (отображаемый в его полосе заголовка) или оконным элементом в указанный буфер указанного размера. Она не может воспринять текст окна редактирования из другого приложения.

Параметр **hWnd** — дескриптор окна. Параметр **lpString** указывает на буфер, в который копируется текст. Параметр **nMaxCount** определяет максимальное число копируемых символов. Если число символов в тексте превышает эту величину, текст усекается.

Если функция выполнилась успешно, она возвращает число скопированных символов, исключая завершающий нулевой символ. Если окно не имеет полосы заголовка или текст заголовка отсутствует, или при неверном дескрипторе возвращается нуль. Развернутую информацию об ошибке можно получить вызовом функции **GetLastError.**

Функция посылает указанному окну или элементу, указанному в ее вызове, сообщение Windows **WM_GETTEXT.**

Пример

В приведенном ниже коде предполагается, что неизвестен класс приложения Windows «Калькулятор». Код проверяет, имеется ли выполняемое приложение «Калькулятор», и если имеется, то его выполнение завершается посылкой функцией **SendMessage** сообщения **WM_CLOSE.**

```
HWND H =Handle;
char Pch[128];
do
{
 H = GetNextWindow(H,GW_HWNDNEXT);
 GetWindowText(H,Pch,128);
 if(CompareText(Pch, "Калькулятор") == 0)
   break;
) while (H != NULL) ;
```

```
if (H != NULL)
  SendMessage(H, WM_CLOSE,0,0);
```

Первый выполняемый оператор присваивает переменной H значение свойства **Handle** формы вашего приложения. Далее в цикле просматриваются окна, лежащие ниже в Z-последовательности, и их текст функцией **GetWindowText** заносится в переменную Pch. При этом ищется окно с текстом «Калькулятор». Для этого используется функция **CompareText.** сравнивающая без учета регистра строку, на которую указывает Pch, со строкой «Калькулятор». Если строки совпадают, функция **CompareText** возвращает нуль. Пользуясь тем, что C++ позволяет оперировать с целыми значениями как с булевыми, строку оператора if можно было бы записать и в таком виде:

```
if( ! CompareText(Pch, "Калькулятор"))
```

Окно калькулятора будет найдено, если оно получало фокус после запуска вашего приложения. Таким образом, если пользователь запустил «Калькулятор» из вашего приложения или даже если «Калькулятор» был запущен ранее или независимо от вашего приложения, дескриптор его окна будет найден.

Если цикл завершается со значением H = NULL, значит приложение «Калькулятор» в данный момент не открыто.

_getws — ввод строки из потока

Вводит строку из стандартного потока.

См. разд. «fputs и другие функции ввода/вывода строк».

HourOf — дешифрация часа

Определяет час.

См. разд. «DayOf и другие функции дешифрации дат и времени».

HourOfTheDay — дешифрация часа дня

Определяет час дня.

См. разд. «DayOf и другие функции дешифрации дат и времени».

HoursBetween и другие функции определения разности часов двух дат

Возвращают число часов между двумя значениями даты и времени.

Заголовочный файл *DateUtils.hpp.*

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE int___fastcall  HoursBetween(
                                const System::TDateTime ANow,
                                const System::TDateTime AThen);
extern PACKAGE double___fastcall HourSpan(
                                const System::TDateTime ANow,
                                const System::TDateTime AThen);
```

Описание

Функции **HoursBetween и HourSpan** возвращают число часов между двумя значениями даты и времени **ANow** и **AThen** типа **TDateTime.** Функция **DaysBetween** возвращает число полных часов между двумя датами. А функция **DaySpan** возвращает действительное число, содержащее дробную часть, отображающую неполные часы.

Примеры

Операторы

```
TDateTime T1 = EncodeDateTime(2002, 10, 5, 11, 00, 45, 300);
```

```
TDateTime T2 = EncodeDateTime(2002, 10, 5, 11, 59, 45, 300);
int i = HoursBetween(T2, T1);
double r = HourSpan(T2, T1);
```

зададут переменной i значение 0, а переменной r значение 0,983333333220799. В этом примере значения дат и времени T1 и T2 задаются с помощью функции **EncodeDateTime.** Различие между двумя значениями составляет 59 минут. Поэтому функция **HoursBetween** возвращает 0, так как разность значений менее часа. А функция **HourSpan** возвращает число, близкое к единице.

HourSpan — разность часов двух дат

Возвращает число часов между двумя значениями даты и времени.

См. разд. «HoursBetween и другие функции определения разности часов двух дат».

InputBox — диалог запроса пользователю

Возвращает текст, введенный пользователем в диалоговом окне с указанным заголовком и сообщением.

Заголовочный файл *Dialogs.hpp.*

Синтаксис

```
#include <Dialogs.hpp>
extern PACKAGE AnsiString __fastcall InputBox(
            constAnsiString ACaption, const AnsiString APrompt,
            const AnsiString ADefault);
```

Описание

Функция **InputBox** предлагает пользователю диалоговое окно (см. рис. 4.1) с заголовком ACaption, с предложением **APrompt** пользователю что-то написать и с окошком редактирования, в котором предварительно загружено значение текста по умолчанию **ADefault.** Если пользователь нажмет в окне OK, то функция вернет введенную им строку текста. Если же пользователь в диалоге нажал Cancel, или нажал Esc, или закрыл окно системной кнопкой, то функция вернет строку **ADefault,** даже если перед этим пользователь что-то написал в окошке редактирования.

Понять по возвращенному результату, написал ли пользователь какой-то текст, или отказался от ввода, можно, сравнив возвращенный результат со значением **ADefault.** Впрочем, результат останется неизменным и в случае, если пользователь ничего не написал в диалоге, но нажал кнопку OK. Если надо достоверно знать, отказался ли пользователь от диалога, или нажал OK, следует использовать похожую на **InputBox** функцию **InputQuery.**

Пример

Оператор

```
AnsiString Goal = InputBox("Регистрация",
                           "Укажите цель Вашего приезда",
                           "Неизвестна");
```

отобразит окно, представленное на рис. 4.1, и вернет текст, введенный пользователем, или строку "Неизвестна".

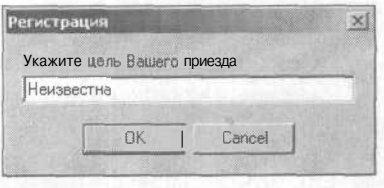


Рис. 4.1
Пример окна, выводимого функцией InputBox

Еще один пример использования InputBox приведен в описании функции SelectDirectory.

InputQuery — диалог запроса пользователю

Возвращает текст, введенный пользователем в диалоговом окне с указанным заголовком и сообщением, и позволяет определить действия пользователя в диалоге.

Заголовочный файл *Dialogs.hpp.*

Синтаксис

```
#include <Dialogs.hpp>
extern PACKAGE bool __fastcall InputQuery(
            constAnsiString ACaption, const AnsiString APrompt,
            AnsiString &Value);
```

Описание

Функция **InputQuery** предлагает пользователю диалоговое окно (см. рис. 4.2) с заголовком **ACaption,** с предложением **APrompt** пользователю что-то написать и с окошком редактирования, в котором пользователь может написать ответ. Параметр **Value** — это строка текста в окошке редактирования. Вы можете присвоить ей начальное значение, а после вызова **InputQuery** в параметре **Value** будет находиться ответ пользователя.

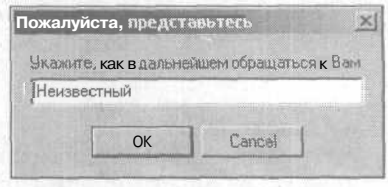


Рис. 4.2
Пример окна, выводимого функцией InputQuery

Функция **InputQuery** возвращает **true** только в том случае, если пользователь вышел из диалога, нажав OK. В остальных случаях (при нажатии Esc, при щелчке на системной кнопке окна или на кнопке Cancel) возвращается **false,** а значение параметра **Value** сохраняется тем, какое было до обращения к **InputQuery.** Этой возможностью достоверно определить, нажал ли пользователь OK или вышел из окна иным способом, функция **InputQuery** отличается от похожей на нее функции **InputBox.**

Пример

Операторы

```
AnsiString Name="Неизвестный";
if (I InputQuery("Пожалуйста, представьтесь",
                 "Укажите, как в дальнейшем обращаться к Вам", Name))
  ShowMessage("Вы не представились, господин Неизвестный");
else ShowMessage("Здравствуйте, господин " + Name + " !");
```

отобразят окно, представленное на рис. 4.2, и после окончания диалога выдадут сообщение, зависящее от того, нажал ли пользователь в диалоге OK, или нет.

InRange — функция

Определяет, лежит ли указанное число в заданном диапазоне.

Заголовочный файл *Math.hpp.*

Синтаксис

```
extern PACKAGE bool __fastcall InRange(const int AValue,
                                       const int AMin, const int AMax);
```

```
extern PACKAGE bool __fastcall InRange(const __int64 AValue,
                    const __int64 AMin, const __int64 AMax);
extern PACKAGE bool __fastcall InRange(const double AValue,
                    const double AMin, const double AMax);
```

Описание

Перегруженные варианты функции **InRange** возвращают **true,** если число **AValue** лежит в диапазоне **AMin - AMax,** включая границы. Возвращается **false,** если **AValue** строго меньше **AMin** или строго больше **AMax.**

Например, операторы

```
L = InRange(5, 1, 10);
L = InRange(1, 1, 10);
L = InRange(10, 1, 10);
```

присвоят булевой переменной L значение **true,** а операторы

```
L = InRange(11, 1, 10);
L = InRange(0, 1, 10);
```

присвоят **L = false.**

IntPower — возведение в целую степень

Возводит заданное число в заданную целую степень.

См. разд. «pow, powl и другие функции возведения в степень».

IntToStr — преобразование целого числа в строку

Преобразует целое число в строку.

Заголовочный файл *SysUtils.hpp.*

Синтаксис

```
extern PACKAGE AnsiString __fastcall IntToStr(int Value);
extern PACKAGE AnsiString __fastcall IntToStr(__int64 Value);
```

Описание

Функция **IntToStr** преобразует целое значение **Value** в строку. Параметр **Value** — целая константа или выражение. Если преобразовываемое значение превышает величину, допустимую для целого типа данных, результат преобразования может быть неверным.

Примеры

Операторы

```
int A = 40000;
int B = 40000;
Edit1->Text = IntToStr(A * B );
```

обеспечат отображение в окне **Edit1** строки "1600000000". Но если в этих операторах заменить число 40000 на 50000, то результат будет "-1794967296", т.е. неверным, так как выражение, переданное в функцию **IntToStr,** превысит допустимое значение.

IsInfinite — проверка на бесконечность

Определяет, равен ли аргумент бесконечности.

Заголовочный файл *Math.hpp.*

Синтаксис

```
#include <Math.hpp>
extern PACKAGE bool __fastcall IsInfinite(const double AValue);
```

Описание

Функция **IsInfinite** определяет, не равен ли аргумент значениям **Infinity** или **NegInfinity** — константам, определяющим положительную и отрицательную бесконечности (см. в гл. 1 разд. 1.9.3). В случае бесконечного значения аргумента возвращается **true.** В этом случае знак бесконечности можно определить функцией Sign.

Например:

```
if (IsInfinite(X))
   if (Sign(X) < 0)
    ShowMessage("Отрицательная бесконечность");
   else ShowMessage ("Положительная бесконечность");
```

Учтите, что константы **Infinity и NegInfinity** нельзя непосредственно использовать в операциях сравнения. Именно для таких сравнений и предусмотрена функция **IsInfinite.** Учтите также, что бесконечные значения появляются при вычислениях с плавающей запятой только в случае, если вы замаскировали генерацию соответствующих исключений

IsNan — функция

Определяет, не равен ли аргумент нечисловому значению.

Заголовочный файл *Math.hpp.*

Синтаксис

```
- extern PACKAGE bool __fastcall IsNan(const double AValue);
```

Описание

Функция **IsNan** определяет, равен ли аргумент значению NaN — константе, определяющей нечисловой результат выполнения арифметической операции (см. в гл. 1 разд. 1.9.3). Если аргумент равен NaN, то возвращается **true.** Учтите, что константу NaN нельзя непосредственно использовать в операциях сравнения. Именно для такого сравнения и введена функция IsNan. Учтите также, что значение NaN появляется при вычислениях с плавающей запятой только в случае, если вы замаскировали генерацию соответствующих исключений.

IsToday — определяет, является ли дата сегодняшней

Позволяет определить, является ли дата сегодняшней.

См. разд. «Date и другие функции определения даты и времени».

labs — функция вычисления модуля

Функция вычисляет модуль целого числа.

См. разд. «abs, labs, fabs, fabsl — функции вычисления модуля.

ldexp, ldexpl, Ldexp — умножение на 2 в степени

Умножают число на 2 в заданной степени.

Заголовочные файлы *math.h* и *math.hpp.*

Синтаксис

```
tfinclude <math.h>
double ldexp(double x, int exp);
long double ldexpl(long double x, int exp);

#include <math.hpp>
extern PACKAGE Extended __fastcall Ldexp(Extended X, int exp);
```

Описание

Функции вычисляют x • 2^{exp}, т.е. умножают число x на 2 в заданной степени **exp.**

Если при вычислении возникают ошибки, их обработчик можно изменить с помощью функций **_matherr** (для **ldexp**) **и _matherrl** (для **ldexpl**) (см. разд. 3.1.4.4).

ldiv — целочисленное деление

Целочисленное деление, возвращающее целое значение частного и остаток.

См. разд. «div и другие функции целочисленного деления».

LnXP1 — вычисление натурального логарифма

Вычисляют логарифмы по разным основаниям.

См. разд. «log и другие логарифмические функции».

log и другие логарифмические функции

Вычисляют логарифмы по разным основаниям.

Заголовочные файлы *math.h* и *math.hpp.*

Синтаксис

```
#include <math.h>
double log(double x);
long double logl(long double x);
double log10(double x);
long double loglOl(long double x);

#include<math.hpp>
extern PACKAGE Extended__fastcall Log10(Extended X);
extern PACKAGE Extended__fastcall Log2(Extended X);
extern PACKAGE Extended__fastcall LogN(Extended N, Extended X);
extern PACKAGE Extended__fastcall LnXP1(Extended X);
```

Описание

Функции вычисляют логарифмы по разным основаниям и для разных типов данных. Функции **log** и **logl** вычисляют натуральный логарифм по основанию e. Функции loglO и **loglOl, LoglO** вычисляют десятичный логарифм по основанию 10. Функция **Log2** вычисляет логарифм по основанию 2. Функция LogN вычисляет логарифм по произвольному основанию, передаваемому в нее через параметр **N.** Функция **LnXPl** вычисляет натуральный логарифм, но не от аргумента X, а от выражения **(X+1).** Эту функцию удобно использовать при значениях X << 1.

Если в функции (кроме **LnXPl)** передается отрицательный аргумент, глобальной переменной **errno** задается значение **EDOM.** Если в те же функции передается аргумент, равный 0, то возвращается отрицательное значение **HUGE_VAL** (функции **log** и **loglO), или _LHUGE_VAL** (функции **logl и loglOl).** Переменной errno задается при этом значение **ERANGE.**

Стандартный обработчик ошибок этих функций можно изменить, задав собственные обработчики — функции **_matherr и _matherrl** (см. разд. 3.1.4.4).

loglO, loglOl, logl — вычисление логарифмов

Вычисляют десятичные и натуральные логарифмы.

См. разд. «log и другие логарифмические функции».

LoglO, Log2, LogN — вычисление логарифмов

Вычисляют логарифмы по разным основаниям.

См. разд. «log и другие логарифмические функции».

LowerCase — преобразование строки к нижнему регистру

Преобразует строку к нижнему регистру.

См. разд. «AnsiLowerCase и другие функции преобразования строки к нижнему регистру».

_lrand — генерация псевдослучайных чисел

Генерирует целые псевдослучайные числа.

См. разд. «random и другие функции генерации псевдослучайных чисел».

_lrotl — циклический сдвиг целого числа влево

Осуществляет циклический сдвиг влево целого числа.

См. разд. «_rotl и другие функции циклического сдвига».

_lrotr — циклический сдвиг целого числа вправо

Осуществляет циклический сдвиг вправо целого числа.

См. разд. «_rotl и другие функции циклического сдвига».

main — функция

Главная функция консольных приложений C и C++.

Определения

```
#include <dos.h>
extern int _argc;
extern char **_argv;
extern wchar_t ** _wargv;
extern char ** _environ;
extern wchar_t ** _wenviron;

int main ()
int main(int argc)
int main(int argc, char * argv[])
int main(int argc, char * argv[], char * env[])
int wmain(int argc, wchar_t *argv[])
int _tmain(int argc, _TCHAR *argv[])
```

Описание

Функция **main** размещается в головном файле консольного приложения C и C++ и ей передается управление в начале выполнения приложения.

В настоящее время стандарт C++ признает только две формы функции — первую и третью. Впрочем, реально компиляторы признают и остальные формы, и даже иные — например, с возвращаемым типом **void.**

Вариант **wmain** является версией Unicode, в которой третий параметр — строка Unicode. Вариант **_tmain** — это макрос, форма развертывания которого автоматически изменяется в зависимости от типа приложения.

Параметр **argc** содержит число параметров, передаваемых в приложение через командную строку. В число этих параметров входит и нулевой параметр, представляющий собой имя выполняемого файла с полным путем к нему. Таким образом, если никакие параметры через командную строку не переданы, **argc** = 1.

Параметр **argv** — это массив указателей на строки, содержащие значения параметров, переданных через командную строку. Параметр env — это аналогичный параметру **argv** массив указателей на строки, содержащие информацию о переменных окружения в форме:

```
имя_переменной=значение
```

Имена переменных — это такие имена, как PATH, COMSPEC и т.п.

В заголовке функции **main** можно не указывать никаких параметров. В этом случае доступ к параметрам командной строки и к переменным окружения можно получить с помощью глобальных переменных **_argc, _argv** и **_environ.**

Согласно стандарту C++, функцию не обязательно завершать оператором **return.** В отличие от C, C++ при отсутствии этого оператора неявно вставляет в конце функции оператор

```
return 0;
```

свидетельствующий об успешном выполнении.

Примеры

Следующий код отображает всю информацию о параметрах командной строки и переменных окружения, полученную из параметров функции **main.** Для простоты тексты сообщений даны английские. Для использования в консольных приложениях русских текстов надо перекодировать их в формат DOS с помощью функций **CharToOem или CharToOemBuff** (см. разд. «CharToOem, CharToOemBuff — перевод строки в текст DOS»).

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[])
{
  int i;
  printf("The value of argc is %d \n\n", argc);
  printf("These are the %d command-line arguments"
         " passed to main:\n\n", argc);
  for (i = 0; i < argc; i++)
    printf("  argv[%d]: %s\n", i, argv[i]);
  printf("\nThe environment string(s) "
         "on this system are:\n\n");
  for (i = 0; env[i] != NULL; i++)
     printf("  env[%d]: %s\n", i, env[i]);
 getchar();
 return 0;
}
```

Выдаваемый это программой текст может иметь вид (считается, что через командную строку передан один параметр "-v"):

```
The value of argc is 2

These are the 2 command-line arguments passed to main:

argv[0]: C:\CBUILDER\TEST.EXE
argv[1]: -v

The environment string(s) on this system are

   env[0]: COMSPEC=C:\COMMAND.COM
   env[1]: PROMPT=$p $g
   env[2]: PATH=C:\SPRINT;C:\DOS;C:\ CBUILDER
```

Ниже приведен аналогичный пример, в котором вместо параметров функции **main** используются глобальные переменные:

```
#include <stdio.h>
#include <stdlib.h>

int main(/*int argc, char *argv[], char *env[]*/)
{
 int i;
 printfC'The value of argc is %d \n\n", _argc);
```

```
    printf("These are the %d command-line arguments"
           " passed to _argc\n\n", _argc);
    for (i = 0; i < _argc; i++)
       printf("   argv[%d] : %s\n", i, _argv[i]);

    printf("\nThe environment string(s) "
           "on this system are:\n\n");
    for (i = 0; _environ[i] != NULL; i++)
      printf("   env[%d]: %s\n", i, _environ[i]);
    getchar();
    return 0;
}
```

Результат выполнения этой программы аналогичен приведенному выше.

В данном примере весь вывод помещен в тело функции **main. Но** с таким же успехом он мог бы быть помещен в любую другую функцию, определенную в приложении.

malloc и другие функции динамического распределения памяти

Функции динамически выделяют и освобождают память.

Заголовочные файлы *stdlib.h* или *alloc.h*.

Синтаксис

```
void *malloc(size_t size);
void *calloc(size_t nitems, size_t size);
void *realloc(void *block, size_t size);
void free(void *block);
```

Описание

Функции **malloc и calloc** динамически выделяют блок памяти под объекты (см. гл. 1, разд. 1.11). Функция **realloc** позволяет изменить размер ранее выделенного блока. А функция **free** освобождает выделенную этими функциями память. Имеется также альтернативный подход к динамическому распределению памяти — операции **new** и **delete,** описанные в разд. 1.11.

Функция **malloc** выделяет в динамически распределяемой области памяти (heap) блок размером в **size** байтов. В случае успешного выделения памяти функция возвращает указатель на выделенный блок. Если не хватило места для блока требуемого размера или если **size = 0,** возвращается **NULL.**

Другая функция — **calloc** выделяет память под **nitems** объектов, размер каждого из которых равен size. Таким образом, общий объем выделяемой памяти составляет **nitems * size.** Выделенная память инициализируется нулями. В случае успешного выделения памяти функция возвращает указатель на выделенный блок. Если не хватило места для блока требуемого размера, или если **size = 0, или nitems** = 0, возвращается **NULL.**

Еще одна функция — **realloc** позволяет изменить размер ранее выделенного блока памяти. Она изменяет размер блока в heap, на который указывает **block,** до размера size. При этом предполагается, что **block** указывает блок памяти, выделенной ранее функциями **malloc, calloc** или **realloc.** Если же аргумент **block** задан равным **NULL,** то функция **realloc** работает так же, как описанная выше функция **malloc.**

Если размер size задан равным нулю, то выделенный ранее блок, на который указывает **block,** освобождается, а функция возвращает NULL. Таким образом, функция с **size** равным 0 может использоваться не для выделения памяти, а для освобождения памяти, выделенной ранее.

Если блок нового размера не может быть выделен, то функция **realloc** возвращает **NULL.** Если же память выделилась успешно, то возвращается адрес выделенного блока. При этом он может отличаться от начального значения **block,** поскольку функция при необходимости осуществляет копирование содержимого блока в новое место.

Функция **free** освобождает блок памяти, выделенный ранее функциями **malloc**, calloc или realloc, на который указывает block.

Примеры

Следующий код динамически выделяет функцией malloc память под строку, а затем, после выполнения с ней каких-то операций, освобождает выделенную память.

```
tinclude <stdio.h>
#include <alloc.h>
char *str;

// str - указатель на строку, под которую выделена память
str = (char *) malloc(100);
...
// освобождение памяти
free(str);
```

В этом примере можно было бы использовать для выделения памяти функцию calloc:

```
str = (char *) calloc(100, sizeof(char));
```

Размер выделенной функциями malloc или calloc памяти можно было бы изменить, например, следующим оператором:

```
str = (char *) realloc(str, 20);
```

Впрочем, к тому же результату привел бы и более простой оператор:

```
realloc(str, 20);
```

Необходимо помнить, что рассмотренные функции возвращают NULL (0), если память не удалось выделить. Поэтому прежде, чем использовать возвращенные ими указатели, надо обязательно проверять, не равны ли они NULL. Иначе возможны очень тяжелые ошибки при работе программы.

MaxIntValue, MaxValue — вычисление максимального значения

Возвращают максимальное значение со знаком элементов массива.

Заголовочный файл *Math.hpp*.

Синтаксис

```
#include <Math.hpp>
extern PACKAGE int___fastcall
            MaxIntValue(const int * Data, const int Data_Size);
extern PACKAGE double___fastcall
            MaxValue(const double * Data, const int Data_Size);
```

Описание

Функции MaxIntValue и MaxValue возвращают максимальное значение со знаком элементов массива Data соответственно целых или действительных чисел. Параметр Data_Size — индекс последнего элемента массива, учитываемого при подсчете среднего значения.

Например, оператор

```
B = MaxValue(A, 99);
```

присваивает действительной переменной B максимальное из значений первых 100 элементов, хранящихся в массиве действительных чисел A.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

_mbscpy — копирование строк

Копирует одну строку в другую.

См. разд. «StrCopy и другие функции копирования строк».

_mbslwr — преобразование строки к нижнему регистру

Преобразует строку к нижнему регистру.

См. разд. «AnsiLowerCase и другие функции преобразования строки к нижнему регистру».

_mbsncpy — копирование строк

Копирует одну строку в другую.

См. разд. «StrCopy и другие функции копирования строк».

_mbsupr — преобразование строки к верхнему регистру

Преобразует строку к верхнему регистру.

См. разд. «AnsiUpperCase и другие функции преобразования строки к верхнему регистру».

Mean — вычисление среднего значения

Возвращает среднее арифметическое значение элементов массива.

Заголовочный файл *Math.hpp*.

Синтаксис

```
#include <Math.hpp>
extern PACKAGE Extended __fastcall
               Mean(const double * Data, const int Data_Size);
```

Описание

Функция **Mean** возвращает среднее арифметическое значение (математическое ожидание) элементов массива действительных чисел **Data.** Параметр **Data_Size** – индекс последнего элемента массива, учитываемого при подсчете среднего значения. Если имеется массив действительных чисел A, содержащий n элементов, то среднее значение рассчитывается по формуле

$$\sum_{j=1}^{n} A[i]/n.$$

Например, оператор

```
B = Mean(A, 99);
```

присваивает действительной переменной B значение математического ожидания первых 100 элементов, хранящихся в массиве действительных чисел A.

Если необходимо одновременно рассчитывать математическое ожидание и среднее квадратическое отклонение, то лучше воспользоваться более быстрой процедурой **MeanAndStdDev.**

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

MeanAndStdDev — вычисление среднего значения и среднего квадратического отклонения

Вычисляет математическое ожидание и среднее квадратическое отклонение элементов массива.

Заголовочный файл *Math.hpp*.

Синтаксис

```
#include <Math.hpp>
extern PACKAGE void__fastcall
        MeanAndStdDev(const double * Data, const int Data_Size,
                      Extended &Mean, Extended &StdDev);
```

Описание

Функция **MeanAndStdDev** рассчитывает для массива **Data (Data_Size** — индекс последнего элемента массива, учитываемого при подсчете) одновременно математическое ожидание **Mean** и несмещенную оценку среднего квадратического отклонения **StdDev.** Время вычислений по процедуре **MeanAndStdDev** вдвое меньше, чем при последовательном вызове функций **Mean и StdDev.** Поэтому, если требуется знать и математическое ожидание, и среднее квадратическое отклонение, то лучше использовать именно эту процедуру.

Формулы вычислений те же, которые приведены в описаниях функций **Mean и StdDev.**

При очень больших значениях математического ожидания (> 107) и сравнительно малых значениях среднего квадратического отклонения при расчетах возможны погрешности.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

memccpy — копирование блоков памяти

Копирует блок памяти.

См. разд. «тетеру и другие функции копирования и заполнения блоков памяти».

тетеру и другие функции копирования и заполнения блоков памяти

Копируют блоки памяти или заполняют блок заданными значениями.

Заголовочные файлы *mem.h*, *string.h*.

Синтаксис

```
void *memcpy(void *dest, const void *src, size_t n); <
void *_wmemcpy(void *dest, const void *src, size_t n) ;
void *memmove(void *dest, const void *src, size_t n);
void *memccpy(void *dest, const void *src,
                                    int c, size_t n);
void *memset(void *s, int c, size_t n);
void *_wmemset(void *s, int c, size_t n);
```

Описание

Функции **memcpy и _wmemcpy** копируют в блок **dest n** байтов из блока **src.** Например, операторы

```
char S1[] = "12345", S2[] = "6789";
memcpy(S1, S2, strlen(S2));
```

приведут к тому, что в массиве S1 будет храниться текст "67895". Как видите, при копировании в конце копии не записывается нулевой символ. Так что после скопированных символов идут те символы, которые были записаны в приемник ранее. Если это не устраивает, надо перед копированием очистить приемник описанной далее функцией **memset.**

Функции возвращают указатель на приемник копирования. Но это указатель типа **(void** *), так что перед использованием его надо явным образом приводить

к требуемому типу. Например, если в приведенном выше коде заменить второй оператор на

```
char *P = (char *)memcpy(S1, S2, strle'n(S2));
```

то P будет указывать на строку **S1**, содержащую результат копирования.

При применении функций **memcpy и _wmemcpy** блоки источника и приемника не должны перекрываться в памяти. Иначе результат непредсказуем. Функция **memmove** отличается тем, что работает даже при перекрытии блоков в памяти. Т.е. она работает так, как будто блок-источник сначала копируется во временный массив, а затем содержание этого массива переносится в блок-приемник.

Функция **memccpy** копирует **src** в **dest,** пока или не скопирует первый символ, совпадающий с заданным значением c, или не скопирует n символов. Если символ c скопирован, функция возвращает указатель на байт в **dest,** следующий непосредственно за ним. Если символ c не скопирован, возвращается **NULL.**

Ниже приведен пример применения функции memccpy:

```
char S[21];
char *Source = (Edit1->Text).c_str();
char *P = (char *)memccpy(S, Source, ' ', 20);
if (P == NULL)
 ShowMessage("Пробел в первых 20 символах не найден");
else
{
 *(P-1) = '\0';
 ShowMessage("Слово '"+AnsiString(S)+"', символов "+
             IntToStr(P - S - 1));
}
```

Создается буфер S на 20 значащих символов (плюс нулевой символ). В строку **Source** заносится текст из окна **Edit1**. Затем из **Source** в S копируется текст до символа пробела (одно слово), но не более 20-ти символов. Если указатель P, который вернула функция **memccpy,** равен **NULL,** выводится соответствующее сообщение. В противном случае в S заносится нулевой символ в позицию, на 1 меньшую, чем та, на которую указывает P. Выводится сообщение о найденном слове и числе символов в нем.

Функции **memset и _wmemset** заполняют первые n байтов блока s символами c и возвращают указатель на s. Например, оператор

```
memset(S, '\0', sizeof(S) - 1);
```

заполняет всю строку S нулевыми символами. Это можно использовать для предварительной очистки строки перед копированием в нее рассмотренными ранее функциями. Операторы

```
memset(S, '*', sizeof(S) - 1);
*(S+sizeof(S)-1) = '\0';
```

заполняют строку S символами "*" и записывают в конец нулевой символ. Подобная строка может использоваться далее для записи разделительных строк в каких-то операциях вывода.

Если блоки памяти являются строками (массивами символов), то, помимо рассмотренных выше функций, для копирования можно использовать функции копирования строк, описанные в разд. «StrCopy и другие функции копирования строк».

memmove — копирование блоков памяти

Копирует блок памяти.

См. разд. «memcpy и другие функции копирования и заполнения блоков памяти».

memset — заполнение блока памяти

Заполняет блок памяти заданным символом.

См. разд. «memcpy и другие функции копирования и заполнения блоков памяти».

MessageBox — метод TApplication

Метод, отображающий полностью русифицированное диалоговое окно сообщения.

Модуль *Forms*

Объявление

```
function MessageBox(Text, Caption: PChar; Flags: Longint): Integer;
```

Описание

Функция **MessageBox** является методом переменной **Application** типа **TApplication,** доступной в любом проекте C++Builder. Он позволяет устранить основной недостаток других функций и процедур отображения диалоговых окон, таких, как **ShowMessage**, **ShowMessageFmt**, **MessageDlg**, **MessageDlgPos. CreateMessageDialog.** Этим недостатком является отсутствие русификации диалоговых окон: английские надписи на кнопках и невозможность указать русский текст заголовка окна (кроме функции **CreateMessageDialog).**

Функция **MessageBox** отображает диалоговое окно с заданными кнопками, сообщением и заголовком и позволяет проанализировать ответ пользователя. Функция инкапсулирует функцию **MessageBox** API Windows. Параметр **Text** представляет собой текст сообщения, которое может превышать 255 символов. Для длинных сообщений осуществляется автоматический перенос текста. Параметр **Caption** представляет собой текст заголовка окна. Он тоже может превышать 255 символов, но не переносится. Так что длинный заголовок приводит к появлению длинного и не очень красивого диалогового окна.

Параметр **Flags** представляет собой множество флагов, определяющих вид и поведение диалогового окна. Этот параметр может комбинироваться операцией сложения по одному флагу из следующих групп.

Флаги кнопок, отображаемых в диалоговом окне

Флаг	Значение (в скобках даны надписи в русифицированных версиях Windows)
MB_ABORTRETRYIGNORE	Кнопки Abort (Стоп), Retry (Повтор) и Ignore (Пропустить).
MB_OK	Кнопка OK. Этот флаг принят по умолчанию.
MB_OKCANCEL	Кнопки OK и Cancel (Отмена).
MB_RETRYCANCEL	Кнопки Retry (Повтор) и Cancel (Отмена),
MB_YESNO	Кнопки Yes (Да) и No (Нет).
MB_YESNOCANCEL	Кнопки Yes (Да), No (Нет) и Cancel (Отмена).

Флаги пиктограмм в диалоговом окне

Флаг	Пиктограмма
MB_ICONEXCLAMATION, MB_ICONWARNING	Восклицательный знак (замечание, предупреждение).

Флаг	Пиктограмма
MB_ICONINFORMATION, MB_ICONASTERISK	Буква "i" в круге (подтверждение).
MB_ICONQUESTION	Знак вопроса (ожидание ответа).
MB_ ICONSTOP, MB_ICONERROR, MB_ICONHAND	Знак креста на красном круге (запрет, ошибка).

Флаги, указывающие кнопку по умолчанию (которая в первый момент находится в фокусе)

Флаг	Кнопка
MB_DEFBUTTON1	Первая кнопка. Это принято по умолчанию.
MB_DEFBUTTON2	Вторая кнопка.
MB_DEFBUTTON3	Третья кнопка.
MB_DEFBUTTON4	Четвертая кнопка.

Флаги модальности

Флаг	**Пояснение**
MB_APPLMODAL	Пользователь должен ответить на запрос, прежде чем сможет продолжить работу с приложением. Но он может перейти в окна другого приложения. Он может также работать со всплывающими окнами данного приложения. Этот флаг принят по умолчанию.
MB_SYSTEMMODAL	То же самое, что **MB_APPLMODAL,** но окно диалога отображается в стиле WS_EX_TOPMOST, то есть всегда остается поверх других окон, даже если пользователь перешел к другим приложениям. Используется для предупреждения о серьезных ошибках, требующих немедленного вмешательства.

Некоторые **дополнительные флаги (могут задаваться оба флага)**

Флаг	Пояснение
MB_HELP	Добавляет в окно кнопку Help (Справка), щелчок на которой или нажатие клавиши F1 генерирует событие Help.
MB_TOPMOST	Помещает окно всегда сверху (в стиле WS EX TOPMOST).

Возможны еще некоторые флаги, определяющие характер поведения окна при работе в сети нескольких пользователей, позволяющие отображать тексты справа налево (для восточных языков) и т.п.

Функция возвращает нуль, если не хватает памяти для создания диалогового окна. Если же функция выполнена успешно, то возвращаемая величина свидетельствует о следующем:

Значение	Численное значение	Пояснение
IDABORT	3	Выбрана кнопка Abort (Стоп).
IDCANCEL	2	Выбрана кнопка Cancel (Отмена) или нажата клавиша Esc.
IDIGNORE	5	Выбрана кнопка Ignore (Пропустить).
IDNO	7	Выбрана кнопка No (Нет).
IDOK	1	Выбрана кнопка OK.
IDRETRY	4	Выбрана кнопка Retry (Повтор).
IDYES	6	Выбрана кнопка Yes (Да).

Пример

Ниже приведен текст, предусматривающий проверку правильности ввода данных перед пересылкой записи в базу данных.

```
if (проверка введенных данных)
{
  if (Application->MessageBox(
              "Хотите занести текущую запись в базу данных?",
              "Подтвердите занесение в базу данных",
             MB_YESNOCANCEL + MB_ICONQUESTION)  != IDYES)
    {
      DataSet->Cancel();
      Abort();
    }
}
else
{
   Application->MessageBox ("Ошибочные данные", "Ошибка",
                            MB_ICONSTOP);
   Abort();
}
```

Отображаемые этим кодом окна приведены на рис. 4.3. Безусловно, они более удачны за счет русификации, чем аналогичные окна, приведенные в гл. 4, в описаниях функций **MessageDlg и MessageDlgPos.**

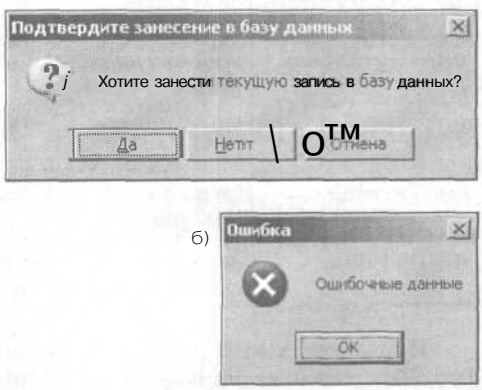


Рис. 4.3
Диалоговые окна, отображаемые функцией Application->MessageBox

MessageDlg и другие функции отображения диалоговых окон

Отображают диалоговые окна и анализируют ответ пользователя.

Заголовочный файл *Dialogs.hpp.*

Синтаксис

```
enum TMsgDlgType {mtWarning, mtError, mtInformation,
                  mtConfirmation, mtCustom } ;
enum TMsgDlgBtn {mbYes, mbNo, mbOK, mbCancel, mbAbort,
                 mbRetry, mblgnore, mbAll, mbNoToAll,
                 mbYesToAll, mbHelp };
tdefine mbYesNoCancel (System::Set<TMsgDlgBtn, mbYes, mbHelp>
                       () << mbYes << mbNo << mbCancel)
#define mbYesNoAllCancel (System::Set<TMsgDlgBtn, mbYes,
                        mbHelp> () << mbYes «mbYesToAll <<
                        mbNo << mbNoToAll << mbCancel)
#define mbOKCancel (System::Set<TMsgDlgBtn, mbYes, mbHelp> ()
                    << mbOK << mbCancel )
#define mbAbortRetryIgnore (System::Set<TMsgDlgBtn, mbYes,
                 mbHelp> () << mbAbort << mbRetry << mblgnore )
#define mbAbortIgnore (System::Set<TMsgDlgBtn, mbYes, mbHelp>
                       () << mbAbort << mblgnore )
typedef Set<TMsgDlgBtn, mbYes, mbHelp> TMsgDlgButtons;

extern PACKAGE int___fastcall
          MessageDlg(constAnsiString Msg, TMsgDlgType DlgType,
                     TMsgDlgButtons Buttons, int HelpCtx);
extern PACKAGE int___fastcall
        MessageDlgPos(constAnsiString Msg, TMsgDlgType DlgType,
                      TMsgDlgButtons Buttons, int HelpCtx,
                      int X, int Y);
extern PACKAGE Forms::TForm*____fastcall
        CreateMessageDialog(const AnsiString Msg,
                            TMsgDlgType DlgType,
                            TMsgDlgButtons Buttons);
```

Описание

Вызов **MessageDlg** отображает диалоговое окно и ожидает ответа пользователя. Сообщение в окне задается параметром функции Msg.

Вид отображаемого окна задается параметром **DlgType.** Возможные значения этого параметра:

Значение	Описание
mtConfirmation	Окно подтверждения, содержащее зеленый вопросительный знак (см. рис. 4.4 а)
mtInformation	Информационное окно, содержащее голубой символ "i"(см. рис. 4.4 б)
mtError	Окно ошибок, содержащее красный стоп-сигнал (см. рис. 4.4 в)
mtWarning	Окно замечаний, содержащее желтый восклицательный знак (см. рис. 4.4 г)
mtCustom	Заказное окно без рисунка. Заголовок соответствует имени выполняемого файла приложения (см. рис. 4.4 д)

Параметр **AButtons** определяет, какие кнопки будут присутствовать в окне. Тип **TMsgDlgBtns** параметра **AButtons** является множеством, которое включает различные кнопки. Возможные значения видов кнопок:

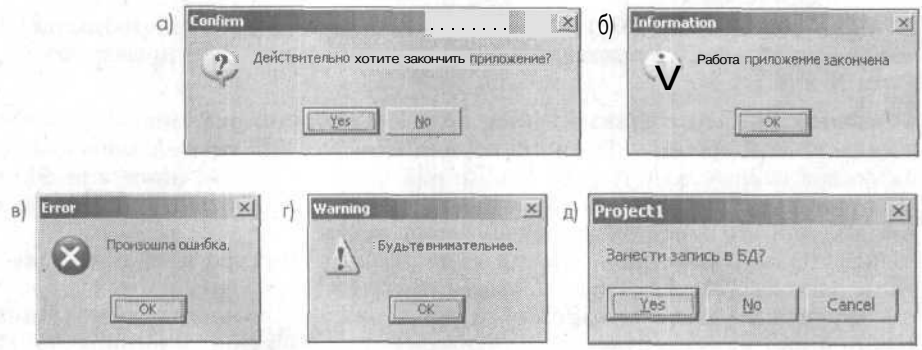


Рис. 4.4. Примеры диалоговых окон, выводимых функциями MessageDlg и MessageDlgPos

Значение	Описание
mbYes	Кнопка с надписью "Yes"
mbNo	Кнопка с надписью "No"
mbOK	Кнопка с надписью "OK"
mbCancel	Кнопка с надписью "Cancel"
mbHelp	Кнопка с надписью "Help"
mbAbort	Кнопка с надписью "Abort"
mbRetry	Кнопка с надписью "Retry"
mbIgnore	Кнопка с надписью "Ignore"
mbAll	Кнопка с надписью "All"

Необходимые кнопки заносятся в **Buttons** операцией "<<", поскольку параметр **Buttons** является множеством. Если не занести в этот параметр ничего, в окне не будет ни одной кнопки и пользователю придется закрывать окно системными кнопками Windows.

Кроме множества значений, соответствующих отдельным кнопкам, определены три константы, соответствующие множествам часто используемых сочетаний кнопок:

Множество	Описание
mbYesNoCancel	Включает в окно кнопки Yes, No и Cancel
mbOkCancel	Включает в окно кнопки OK и Cancel
mbAbortRetryIgnore	Включает в окно кнопки Abort, Retry и Ignore

Эти предопределенные множества имеют тип **TMsgDlgButtons** и могут непосредственно включаться в вызов функции вместо параметра **Buttons.**

Параметр **HelpCtx** определяет экран контекстной справки, соответствующий данному диалоговому окну. Этот экран справки будет появляться при нажатии пользователем клавиши F1. Если вы справку не планируете, при вызове **MessageDlg** надо задать нулевое значение параметра **HelpCtx.**

Функция **MessageDlg** возвращает значение, соответствующее выбранной пользователем кнопке. Возможные возвращаемые значения:

```
mrNone      mrAbort     mrYes
mrOk        mrRetry     mrNo
mrCancel    mrIgnore    mrAll
```

Функция **MessageDlgPos,** во всем подобная **MessageDlg,** отображает диалоговое окно сообщений в заданном месте экрана. Координаты определяются параметрами X и Y.

Функция **CreateMessageDialog** позволяет создать диалоговое окно сообщения в виде объекта формы. Функция только создает окно, но не отображает его. Отображение осуществляется обычными для форм методами **Show** или **ShowModal.** При использовании метода **ShowModal** можно анализировать ответ пользователя так же, как это делается для любых модальных форм.

Использованные для задания типа диалога **DlgType** и кнопок окна **Buttons** типы данных **TMsgDlgType и TMsgDlgButtons** были описаны выше.

Функцию **CreateMessageDialog** имеет смысл применять для создания диалогового окна, которое будет использоваться в приложении многократно. При этом преимуществом этого окна по сравнению с теми, которые создавались ранее рассмотренными функциями, заключается в том, что вы можете задать русскую надпись в заголовке окна, как делаете это для любой формы. В то же время существенным недостатком применения функции **CreateMessageDialog** является то, что объект диалогового окна хранится в памяти все время, пока он не будет уничтожен явно методом **Free.** Это приводит к непроизводительным затратам памяти.

См. также метод **MessageBox.** обеспечивающий, пожалуй, наиболее удачное полностью русифицируемое диалоговое окно.

В диалогах, не требующих ответа пользователя, можно использовать функции, описанные в разд. «ShowMessage и другие функции вывода простых диалоговых окон сообщений».

Примеры

Ниже приведен пример диалога при окончании работы приложения:

```
if(MessageDlg("Действительно хотите закончить приложение?",
              mtConfirmation, TMsgDlgButtons() << mbYes<< mbNo,
                0) == mrYes)
 {
  MessageDlg("Работа приложение закончена", mtInformation,
             TMsgDlgButtons() << mbOK, 0);
  Close();
 }
```

Первый вызов **MessageDlg** приводит к отображению окна типа **mtConfirmation** с вопросом о завершении приложения (см. рис. 4.4 а). Если пользователь нажимает кнопку Yes, то выводится второе окно типа mtlnformation с сообщением о завершении (см. рис. 4.4 б).

Следующий пример иллюстрирует диалог при генерации исключения (см. рис. 4.4 в и 4.4 г):

```
catch ( ... )
{
 MessageDlg("Произошла ошибка.", mtError,
            TMsgDlgButtons() << mbOK, 0);
 MessageDlg("Будьте внимательнее.", mtWarning,
            TMsgDlgButtons() << mbOK, 0);
}
```

Следующий пример иллюстрирует работу с базой данных, когда после редактирования пользователем записи ему предлагается вопрос о сохранении ее в базе данных (см. рис. 4.4 д). Если пользователь выбирает кнопку Yes, запись сохраняется методом **Post;** если пользователь выбирает кнопку No, результаты редактирования уничтожаются методом **Cancel;** если же пользователь выбирает кнопку Cancel, форма закрывается.

```
switch (MessageDlg("Занести запись в БД?", mtCustom,
                   mbYesNoCancel, 0))
```

```
{
  case mrYes: Table1->Post();
              break;
  case mrNo : Table1->Cancel();
              break;
  case mrCancel : Close();
}
```

Оператор

```
MessageDlgPos("Будьте внимательнее.", mtWarning,
              TMsgDlgButtons() << mbOK, 0, 250, 0);
```

вызовет появление окна сообщения вверху экрана (параметр Y = 0) примерно в центре. А оператор

```
MessageDlgPos("Ошибка в этом окне!", mtError,
              TMsgDlgButtons() << mbOK, 0,
              BoundsRect.Left,BoundsRect.Bottom);
```

отобразит диалоговое окно вблизи нижнего левого угла формы, в которой записан данный оператор.

Приведем пример использования **CreateMessageDialog.** Операторы

```
TForm *FMess;
...
FMess = CreateMessageDialog("Будьте внимательнее.", mtWarning,
                            TMsgDlgButtons() << mbOK);
FMess->Caption = "Предупреждение";
```

создают объект FMess диалогового окна, задают текст его сообщения и заголовок "Предупреждение". Вид этого окна (рис. 4.5) идентичен приведенному ранее на рис. 4.4 г, за исключением заголовка "Предупреждение" вместо непонятного не слишком опытному пользователю заголовка "Warning". Оператор

```
FMess->ShowModal();
```

отображает окно как модальную форму. Оператор

```
FMess->Free();
```

уничтожает объект, после чего окно уже не сможет отображаться.

Рис. 4.5
Окно, созданное функцией CreateMessageDialog

MessageDlgPos — отображение диалогового окна в указанной позиции

Отображает диалоговое окно в указанной позиции и анализирует ответ пользователя.

См. разд. «MessageDlg и другие функции отображения диалоговых окон».

MilliSecondOf — дешифрация миллисекунды

Определяет миллисекунду.

См. разд. «DayOf и другие функции дешифрации дат и времени».

MilliSecondOfTheSecond — дешифрация миллисекунды

Определяет миллисекунду.

См. разд. «DayOf и другие функции дешифрации дат и времени».

MilliSecondsBetween и другие функции определения разности миллисекунд

Возвращают число миллисекунд между двумя значениями даты и времени.

Заголовочный файл *DateUtils.hpp*.

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE int__fastcall
          MilliSecondsBetween(const System::TDateTime ANow,
                              const System::TDateTime AThen);
extern PACKAGE double__fastcall
          MilliSecondSpan(const System::TDateTime ANow,
                          const System::TDateTime AThen);
```

Описание

Функции **MilliSecondsBetween и MilliSecondSpan** возвращают число миллисекунд между двумя значениями даты и времени **ANow и AThen** типа **TDateTime.** Возвращаемые функциями значения различаются только типом данных.

MilliSecondSpan — разность миллисекунд двух дат

Возвращает число миллисекунд между двумя значениями даты и времени.

См. разд. «MilliSecondsBetween и другие функции определения разности миллисекунд».

MinIntValue, MinValue — вычисление минимального значения

Возвращают минимальное значение со знаком элементов массива.

Заголовочный файл *Math.hpp*.

Синтаксис

```
#include <Math.hpp>
extern PACKAGE int__fastcall
          MinIntValue(const int * Data, const int Data_Size);
extern PACKAGE double__fastcall
          MinValue(const double * Data, const int Data_Size);
```

Описание

Функции **MinIntValue и MinValue** возвращают минимальное значение со знаком элементов массива **Data** соответственно целых или действительных чисел. Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения.

Например, оператор

```
B = MinValue(A, 99);
```

присваивает действительной переменной B минимальное из значений первых 100 элементов, хранящихся в массиве действительных чисел A.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

MinuteOf — дешифрация минуты

Определяет минуту.

См. разд. «DayOf и другие функции дешифрации дат и времени».

MinuteOfTheHour — дешифрация минуты

Определяет минуту часа.

См. разд. «DayOf и другие функции дешифрации дат и времени».

MinutesBetween и другие функции определения разности минут

Возвращают число минут между двумя значениями даты и времени.

Заголовочный файл *DateUtils.hpp*.

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE int__fastcall
                MinutesBetween(const System::TDateTime ANow,
                               const System::TDateTime AThen);
extern PACKAGE double__fastcall
                MinuteSpan (const System::TDateTime ANow,
                            const System::TDateTime AThen);
```

Описание

Функции **MinutesBetween и MinuteSpan** возвращают число минут между двумя значениями даты и времени ANow и AThen типа **TDateTime.** Функция **MinutesBetween** возвращает число полных минут между двумя значениями. А функция **MinuteSpan** возвращает действительное число, содержащее дробную часть, отображающую неполные минуты.

Примеры

Операторы

```
TDateTime T1 = EncodeDateTime(2002, 10, 5, 11, 00, 00, 300);
TDateTime T2 = EncodeDateTime(2002, 10, 5, 11, 00, 59, 300);
int i = MinutesBetween (T2, T1);
double r = MinuteSpan(T2, T1);
```

зададут переменной i значение О, а переменной г значение 0,98333333269693. В этом примере значения дат и времени T1 и T2 задаются с помощью функции **EncodeDateTime.** Различие между двумя значениями составляет 59 секунд. Поэтому функция **MinutesBetween** возвращает 0, так как разность значений менее минуты. А функция **MinuteSpan** возвращает число, близкое к единице.

MinuteSpan — разность минут

Возвращает число минут между двумя значениями даты и времени.

См. разд. «MinutesBetween и другие функции определения разности минут».

MomentSkewKurtosis — вычисление моментов

Вычисляет первые четыре момента, коэффициент асимметрии и эксцесс элементов массива.

Заголовочный файл *Math.hpp*.

Синтаксис

```
ftinclude  <Math.hpp>
extern PACKAGE void__fastcall
   MomentSkewKurtosis(const double * Data, const int Data_Size,
                      Extended &M1, Extended &M2, Extended &M3,
                      Extended SM4, Extended &Skew,
                      Extended &Kurtosis);
```

Описание

Функция **MomentSkewKurtosis** рассчитывает первые четыре момента **Ml, M2, M3, M4,** коэффициент асимметрии Skew и эксцесс **Kurtosis**. Это набор характери-

стик, описывающий закон распределения случайной величины. Первый момент **M1** равен среднему значению (математическому ожиданию), возвращаемому функциями **Mean** и **MeanAndStdDev.** Моменты M2, M3 и M4 — это смещенные оценки центральных моментов. K-ый центральный момент рассчитывается по формуле

$$\sum_{i=1}^{n} \left(A[i] - \overline{A}\right)^{K} / n.$$

Таким образом, второй центральный момент равен смещенной оценке дисперсии, возвращаемой функцией **PopnVariance.** Третий центральный момент M3 характеризует асимметрию закона распределения. Для симметричных распределений M3 = 0. Чаще асимметрия характеризуется не самим третьим моментом, а безразмерным коэффициентом асимметрии, равным частному от деления M3 на куб среднего квадратического отклонения. Эту величину процедура **MomentSkewKurtosis** заносит в параметр **Skew.** Четвертый центральный момент M4 характеризует «островершинность» распределения. Ексцесс, возвращаемый функцией **MomentSkewKurtosis** в параметр **Kurtosis,** равен моменту M4, деленному на четвертую степень среднего квадратического отклонения. Следует сказать, что в отечественной литературе чаще в качестве характеристики эксцесса используется эта величина, уменьшенная на 3. В этом случае эксцесс нормального закона распределения равен нулю. А коэффициент **Kurtosis** для нормального закона распределения равен 3.

Параметр **Data_Size** функции — это индекс последнего элемента массива, учитываемого при подсчете среднего значения.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

MonthOf — дешифрация месяца

Определяет месяц.

См. разд. «DayOf и другие функции дешифрации дат и времени».

MonthOfTheYear — дешифрация месяца

Определяет месяц.

См. разд. «DayOf и другие функции дешифрации дат и времени».

MonthsBetween и другие функции определения разности месяцев

Возвращают число месяцев между двумя значениями даты и времени.

Заголовочный файл *DateUtils.hpp.*

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE int _fastcall
                MonthsBetween(const System::TDateTime ANow,
                              const System::TDateTime AThen);
extern PACKAGE double _fastcall
                MonthSpan(const System::TDateTime ANow,
                          const System::TDateTime AThen);
```

Описание

Функции **MonthsBetween и MonthSpan** возвращают число месяцев между двумя значениями даты и времени **ANow и AThen** типа **TDateTime.** Функция **MonthsBetween** возвращает число полных месяцев между двумя значениями. А функция **MonthSpan** возвращает действительное число, содержащее дробную часть, отображающую неполный месяц.

Хотя месяцы имеют разную продолжительность, функции **MonthsBetween и MonthSpan** это не учитывают и исходят из усредненного значения 30.4375 дней в месяце. Например, для дат 01.02 и 01.03 любого года функция **MonthsBetween** выдаст число полных месяцев 0, так как февраль содержит меньше дней, чем принято в этой функции.

Примеры

Операторы

```
TDateTime T1 = EncodeDateTime(2002, 01, 4, 11, 00, 00, 300);
TDateTime T2 = EncodeDateTime(2002, 02, 4, 00, 00, 00, 300);
int i = MonthsBetween(T2, T1);
double r = MonthSpan(T2, T1);
```

зададут переменной i значение 1, а переменной r значение 1,00342231348407. В этом примере значения дат и времени T1 и T2 задаются с помощью функции **EncodeDateTime.** Различие между двумя значениями составляет менее месяца (не хватает 11 часов). Но из-за принятых округлений функция **MonthsBetween** возвращает 1, а функция **MonthSpan** возвращает даже число, большее единицы.

MonthSpan — разность месяцев

Возвращает число месяцев между двумя значениями даты и времени.

См. разд. «MonthsBetween и другие функции определения разности месяцев».

_new_handler — указатель на обработчик ошибок выделения памяти

Функция **_new_handler** указывает на обработчик события, связанного с невозможностью динамически выделить требуемый блок памяти.

См. разд. «set_new_handler и другие функции обработки ошибок выделения памяти».

Norm — вычисление кория из суммы квадратов

Возвращает корень из суммы квадратов значений элементов массива.

Заголовочный файл *Math.hpp.*

Синтаксис

```
#include <Math.hpp>
extern PACKAGE Extended__fastcall
                Norm(const double * Data, const int Data_Size);
```

Описание

Функция **Norm** возвращает евклидову норму: корень из суммы квадратов значений элементов массива **Data.** Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

Now — текущая дата и время

Возвращает текущую дату и время.

См. разд. «Date и другие функции определения даты и времени».

OemToChar, OemToCharBuff — перевод текста DOS в строку

Функции API Windows, переводят текст MS-DOS в строку.

Заголовочный файл *winuser.h.*

Синтаксис

```
#include <system.hpp>
BOOL OemToChar(
    LPCTSTR lpszSrc,     // исходная строка
    LPSTR lpszDst        // результат перевода
   );
BOOL OemToCharBuff(
    LPCTSTR IpszSrc,     // исходная строка
    LPSTR IpszDst,       // результат перевода
    DWORD cchDstLength // число символов
   );
```

Описание

Функции применяются для перевода строки в формат MS-DOS. Это требуется, в частности, если возникает задача сохранить в файле в формате DOS текст из окна редактирования. Подобный файл далее может читаться приложениями DOS.

Параметр IpszSrc — указатель на строку, которую надо перекодировать. Параметр **IpszDst** — указатель на строку, в которую заносится перекодированный текст. Параметр **cchDstLength** в функции **OemToCharBuff** определяет число перекодированных символов, которые заносятся в результирующую строку. Если это число меньше числа символов в исходной строке, то остальные символы не заносятся в результирующую строку.

Имеются два варианта функций, работающие с кодами ANSI и с многобайтными кодами Unix. В случае, если работа идет с кодами ANSI, адреса исходной и результирующей строк могут совпадать, т.е. параметры **IpszSrc и IpszDst** могут указывать на одну строку.

Функции всегда возвращают ненулевое значение.

Имеется также функция **CharToOem**, которая осуществляет обратное преобразование.

Пример

В следующем примере, текст, загруженный из файла в формате DOS в окно редактирования **RichEditl,** переводится в формат ASCII.

```
char *S = (char *) malloc(sizeof(RichEdit1->Text));
OemToChar((RichEdit1->Text).c_str(), S);
RichEdit1->Text = S;
free(S);
```

Первый оператор отводит память под строку S, необходимую для хранения в ней текста окна **RichEditl.** Второй оператор заносит в нее перекодированный текст. Третий оператор возвращает этот текст в окно **RichEditl.** Последний оператор освобождает память.

Point и другие функции формирования точки

Формируют точку из координат.

Заголовочные файлы *Types.hpp* и *Classes.hpp*.

Синтаксис

```
#include <Types.hpp>
struct TPoint
{
 int  x;
 int  y;
};

extern PACKAGE TPoint___fastcall Point(int AX, int AY)

#include <Types.hpp>
```

```
struct TSmallPoint
{
 short x;
 short y;
}

extern PACKAGE Types::TSmallPoint __fastcall
                              SmallPoint(short AX, short AY);
```

Описание

Функции возвращают структуры, содержащие заданные координаты точки AX и AY. Такие структуры используются во многих функциях C++Builder (см., в частности, описание функции **Rect**). Координаты задаются в пикселах. В качестве системы координат принимается система координат родительского окна или экрана. За начало координат всегда принимается левый верхний угол родительского окна или экрана.

Функция **Point** формирует структуру типа **TPoint.** Функция SmallPoint формирует структуру типа **TSmallPoint,** в которой координаты представлены 16-битными целыми.

См. примеры применения **Point** в разд. «Bounds и другие функции формирования прямоугольной области».

poly, polyl, Poly — вычисление полиномов

Вычисляют значение полинома заданной степени с заданными коэффициентами.

Заголовочные файлы *math.h* и *math.hpp.*

Синтаксис

```
#include <math.h>
double poly(double x, int degree, double coeffs[]);
long double polyl(long double x, int degree, long double coeffs[]);

#include <math.hpp>
extern PACKAGE Extended __fastcall
                Poly(Extended X, const double * Coefficients,
                     const int Coefficients_Size);
```

Описание

Функции **poly и polyl** возвращают полином от переменной x степени **degree** с коэффициентами, хранящимися в массиве **coeffs:**

```
coeffs[0] + coeffs[1]*x + ... + coeffs[degree]*(x^ degree)
```

Функция **Poly** осуществляет те же самые вычисления. Только последовательность ее аргументов иная: сначала указывается массив коэффициентов **Coefficients,** а затем — степень полинома **Coefficients_Size.**

Примеры

Следующий код вычисляет с помощью функции poly значение полинома $1 + 2*x + 3*x^2 + 4*x^3$ при $x = 10$:

```
double array[4]={1,2,3,4};
double result = poly(10, 3, array);
```

Результат вычислений — 4321.

Следующий код осуществляет те же вычисления с помощью функции **Poly:**

```
double array[4]={1,2,3,4};
double result = Poly(10, array, 3);
```

PopnStdDev — вычисление среднего квадратического отклонения

Возвращает среднее квадратическое отклонение элементов массива.

Заголовочный файл *Math.hpp.*

Синтаксис

```
#include <Math.hpp>
extern PACKAGE Extended__fastcall
        PopnStdDev(const double * Data, const int Data_Size);
```

Описание

Функция **PopnStdDev** возвращает смещенную оценку среднего квадратического отклонения элементов массива действительных чисел **Data.** Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения. Если массив A содержит n элементов, то смещенная оценка среднее квадратическое отклонение рассчитывается по формуле

$$\sqrt{\sum_{i=1}^{n}\left(A[i]-\overline{A}\right)^2/n},$$

где $\overline{A}$ — среднее значение (математическое ожидание) элементов массива. Эта смещенная оценка статистически менее достоверна, чем несмещенная оценка, возвращаемая функцией **StdDev или MeanAndStdDev.** Но в некоторых расчетах используется именно такая смещенная оценка.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

PopnVariance — вычисление дисперсии

Возвращает дисперсию элементов массива.

Заголовочный файл *Math.hpp.*

Синтаксис

```
#include<Math.hpp>
extern PACKAGE Extended__fastcall
       PopnVariance(const double * Data, const int Data_Size);
```

Описание

Функция **PopnVariance** возвращает смещенную оценку дисперсии элементов массива действительных чисел **Data,** т.е. среднее значение квадрата отклонения значений элементов от их среднего значения. Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения. Если массив A содержит n элементов, то дисперсия рассчитывается по формуле

$$\sum_{i=1}^{n}\left(A[i]-\overline{A}\right)^2/n,$$

где $\overline{A}$ — среднее значение (математическое ожидание) элементов массива. Это смещенная оценка дисперсии, статистически менее точная, чем сумма квадратов отклонений, деленная на n — 1, возвращаемая функцией **Variance.**

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

PostMessage — функция API Windows

Функция API Windows, помещает указанное в ней сообщение окну или множеству окон в очередь сообщений потока, создавшего эти окна, и возвращается, не дожидаясь окончания обработки этого сообщения.

Модуль *winuser.*

Объявление

```
BOOL PostMessage(
 HWND hWnd,      // дескриптор окна - приемника
 UINT Msg,       // сообщение
 WPARAM wParam,  // первый параметр сообщения
 LPARAM lParam   // второй параметр сообщения
);
```

Описание

Функция **PostMessage** ставит указанное в ней сообщение окну или множеству окон в очередь сообщений потока, создавшего эти окна, и возвращается, не дожидаясь окончания обработки этого сообщения. Этим функция **PostMessage** отличается от функции **SendMessage.** которая ждет окончания обработки сообщения и на это время блокирует приложение, которое послало сообщение. Сообщения в дальнейшем изымаются из очереди функциями **GetMessage и PeekMessage.**

Параметр **hWnd** — дескриптор окна, которому передается сообщение. Если этот параметр равен **HWND_BROADCAST,** то сообщение передается всем окнам верхнего уровня в системе, включая недоступные, невидимые, перекрытые другими и всплывающие, за исключением дочерних окон. Если этот параметр **NULL,** то сообщение ставится в очередь сообщений (если она есть) текущего процесса.

Параметр **Msg** определяет передаваемое сообщение. Параметры **wParam** и **1Param** могут содержать дополнительную информацию.

Функция возвращает ненулевое значение при успешном завершении и нуль при аварийном завершении. В этом случае причину ошибки можно установить вызовом функции **GetLastError.**

Для определения дескриптора, передаваемого в функцию в качестве параметра **hWnd,** можно использовать функцию **FindWindow.**

Если вы посылаете сообщение в диапазоне ниже **WM_USER** асинхронной функцией **PostMessage,** надо быть уверенным, что параметры сообщения не включают указателей. В противном случае из-за немедленного возврата функции может оказаться, что к моменту, когда поток начнет обрабатывать сообщение, приложение, которое его послало, окажется уже удаленным из памяти.

Приложения, использующие **hWnd = HWND_BROADCAST** для связи между окнами разных приложений, должны предварительно зарегистрировать уникальность своих сообщений функцией **RegisterWindowMessage**.

Примеры

Оператор

```
PostMessage(Form2->Handle,WM_CLOSE,0,0);
```

посылает форме **Form2** сообщение **WM_CLOSE,** закрывающее окно этой формы. Первый параметр функции **PostMessage** содержит дескриптор окна формы **Form2,** полученный с помощью ее свойства **Handle.** Сообщение **WM_CLOSE** не имеет параметров; поэтому параметры **wParam и IParam** заданы равными нулю.

Оператор

```
PostMessage(FindWindow("SciCalc","Калькулятор"), WM_CLOSE,0,0);
```

использует функцию **FindWindow** для получения дескриптора окна приложения, которому надо послать сообщение **WM_CLOSE.** В качестве параметров в **FindWindow** передаются класс формы **SciCalc** и ее заголовок **«Калькулятор».** Это стандартное приложение Windows **«Калькулятор».**

pow, powl и другие функции возведения в степень

Возводят число в заданную степень.

Заголовочные файлы *math.h* и *math.hpp*.

Синтаксис

```
#include<math.h>
double pow(double x, double y);
long double powl(long double x, long double y);
double pow10(int p);
long double pow10l(int p);

#include<math.hpp>
extern PACKAGE Extended__fastcall Power(
                        Extended Base, Extended Exponent);
extern PACKAGE Extended__fastcall IntPower(
                        Extended Base, int Exponent);
```

Описание

Функции **pow** и **powl** возвращают значение x, возведенное в степень y. Если вычисление вызывает переполнение, функции возвращают значение **HUGE_VAL** (функция **pow) или _LHUGE_VAL** (функция **powl).** Глобальная переменная **errno** может при этом установиться в значение **ERANGE.**

Если показатель степени не целое число, а аргумент x отрицательный, глобальная переменная **errno** устанавливается в **EDOM.** То же самое происходит при возведении 0 в 0.

Стандартный обработчик ошибок этих функций можно изменить, задав собственные обработчики — функции **_matherr и _matherrl** (см. разд. 3.1.4.4).

Функция **Power** производит те же вычисления -- возводит значение Base в произвольную степень **Exponent.** Если **Exponent** не целое число или число, большее **MaxInt,** значение **Base** должно быть положительным.

Функция **IntPower** тоже возводит **Base** в степень **Exponent,** но только для целых **Exponent.** Так что в этой функции значение **Base** может быть любым.

Функции **powlO и powlOl** возвращают число 10, возведенное в заданную степень p.

powlO, powlOl — возведение в целую степень

Возводят число в заданную целую степень.

См. разд. «pow, powl и другие функции возведения в степень».

Power — возведение в заданную степень

Возводит число в заданную степень.

См. разд. «pow, powl и другие функции возведения в степень».

printf — форматированный вывод на экран

Выводит форматированные данные в выходной поток.

См. разд. «fprintfи другие функции форматированного вывода».

putc — вывод символа в поток

Выводит символ в указанный поток.

См. разд. «fgetc и другие функции ввода/вывода символа».

putchar — вывод символа в поток

Выводит символ в выходной поток.

См. разд. «fgetc и другие функции ввода/вывода символа».

puts — вывод строки в поток

Выводит строку в стандартный поток вывода.
См. разд. «fputs и другие функции ввода/вывода строк».

putwc — вывод символа в поток

Выводит символ в указанный поток.
См. разд. «fgetc и другие функции ввода/вывода символа».

putwchar — вывод символа в поток

Выводит символ в выходной поток.
См. разд. «fgetc и другие функции ввода/вывода символа».

_putws — вывод строки в поток

Выводит строку в стандартный поток вывода.
См. разд. «fputs и другие функции ввода/вывода строк».

raise — генерация сигнала

Функция генерации заданного сигнала.
См. разд. «signal и другие функции работы с сигналами».

rand, randomize, Randomize, RandG — генерации случайных чисел

Осуществляют генерацию последовательностей псевдослучайных чисел.
См. разд. «random и другие функции генерации псевдослучайных чисел».

random и другие функции генерации псевдослучайных чисел

Осуществляют генерацию последовательностей псевдослучайных чисел.

Заголовочные файлы *stdlib.h, System.hpp, Math.hpp.*

Синтаксис

```
#include <stdlib.h>
long _lrand(void);
int rand(void);
int random(int num);
void randomize(void);
void srand(unsigned seed);

#include <System.hpp>
extern PACKAGE void __fastcall Randomize(void);

#include <Math.hpp>
extern PACKAGE Extended __fastcall RandG(
                                Extended Mean, Extended StdDev);
```

Описание

Функция **rand** возвращает целые псевдослучайные числа, равномерно распределенные в диапазоне от 0 до **RAND_MAX** (Ox7FFFU - 32767). Длина отрезка апериодичности псевдослучайных чисел 2^{32} = 4 294 967 296. Число используемых случайных чисел не должно превышать эту величину. Если вам все-таки требуется больше чисел, то вы должны при приближении к границе отрезка апериодичности (а лучше задолго до нее) обновить последовательность чисел с помощью функций **randomize** или **srand.**

Если желательно генерировать случайные числа, лежащие в диапазоне от 0 до некоторого значения N, то это легко делать операцией вычисления остатка %. Например, выражение

```
rand() % 101
```

возвращает числа в диапазоне от 0 до 100, а выражение

```
(rand() % 201) - 100
```

возвращает числа в диапазоне от -100 до 100.

Функцию **rand** можно использовать и для генерации действительных случайных чисел. Например, выражение

```
10. * rand() / RAND_MAX
```

генерирует псевдослучайные действительные числа, распределенные в диапазоне от 0 до 10.

Функция **_lrand** работает аналогично функции **rand,** но имеет больший отрезок апериодичности — 2^{64} и диапазон от 0 до 2^{31} - 1.

Функция **random** отличается от предыдущих тем, что имеет параметр num, определяющий верхнюю границу диапазона генерируемых чисел. Поэтому, если надо, например, генерировать целые числа в диапазоне от 0 до 100, это можно сделать выражением

```
random(101);
```

не прибегая, как для предыдущих функций, к операции %.

Функция **RandG** генерирует квазислучайные действительные числа, распределенные по нормальному закону (закону Гаусса) с математическим ожиданием **Mean** и средним квадратичным отклонением **StdDev.**

Поскольку генерируемые рассматриваемыми функциями числа являются псевдослучайными, то при каждом новом запуске вашего приложения будет вырабатываться одна и та же последовательность чисел. Это удобно в процессе отладки. Однако в законченном приложении это во многих случаях недопустимо. Чтобы избежать этого, надо рандомизировать генератор чисел, т.е. задавать ему каждый раз новое случайное исходное число. Рандомизацию всех генераторов, кроме **RandG,** осуществляет функция **randomize.** Достаточно вставить где-то в текст программы (например, в событие **OnCreate** формы) оператор

```
randomize();
```

чтобы при каждом запуске приложения генерировалась новая последовательность чисел.

Функция **srand** отличается от **randomize** тем, что задает в качестве начального не случайное число, а значение своего параметра **seed.**

Рандомизацию генератора **RandG** осуществляет функция **Randomize,** аналогичная **randomize.** Задание конкретного начального числа для этого генератора можно осуществить, задавая значение целой переменной **RandSeed,** определенной в файле *System.hpp*.

Пример

Ниже приведен пример генерации нормально распределенных квазислучайных чисел и вычисления характеристик полученного распределения.

```
double A[1001];
long double M, Me, Std, StdD, StdD2, M1, M2, M3, M4, Skew,
            Kurtosis, MinA, MaxA, V, TV, PV, N, SOS, S, SoS, S2;
// заполнение массива нормально распределенными числами
for(int i = 0; i < 1001; i++)
  A[i] = RandG(20,4);
Me = Mean(A,1000);
V = Variance(A,1000);
```

```
TV = TotalVariance(A,1000);
MeanAndStdDev(A,1000, M, StdD) ;
PV = PopnVariance (A, 1000) ;
Std = StdDev(A,1000);
StdD2 = PopnStdDev(A,1000);
N = Norm(A,1000);
MomentSkewKurtosis(A,1000,M1,M2,M3,M4,Skew,Kurtosis);
MinA = MinValue(A,1000) ;
MaxA = MaxValue(A,1000);
SumsAndSquares(A,1000, S,. SoS) ;
SOS = SumOfSquares(A,1000) ;
S2 = Sum(A,1000);
```

realloc — функция выделения памяти

Функция изменяет размер динамически выделенного блока памяти.

См. разд. «malloc и другие функции динамического распределения памяти».

Rect — формирование прямоугольной области

Формирует прямоугольную область типа TRect.

См. разд. «Bounds и другие функции формирования прямоугольной области».

RegisterWindowMessage — функция API Windows

Функция API Windows, определяет новое окно сообщения с гарантированной уникальностью его в системе, которое может использоваться в функциях **SendMessage и PostMessage.**

Модуль *winuser.*

Объявление

```
UINT RegisterWindowMessage(
 LPCTSTR lpString          // адрес строки сообщения
);
```

Описание

Функция **RegisterWindowMessage** используется для регистрации сообщений, предназначенных для связи между различными совместно работающими приложениями. В частности, необходимо для использования функций **SendMessage и PostMessage с hWnd = HWND_BROADCAST.** Если два приложения регистрируют одну и ту же строку сообщения, то им возвращается одинаковый номер этого сообщения. Регистрация действительна до конца сеанса работы Windows.

Параметр **LpString** — указатель на строку с нулевым символом, содержащую регистрируемое сообщение.

Если регистрация прошла успешно, то возвращается идентификатор сообщения в диапазоне от OxCOOO до OxFFFF. Если регистрация завершилась аварийно, то возвращается нулевое значение.

Функцию **RegisterWindowMessage** следует использовать только в случаях, когда несколько приложений должны обрабатывать одно и то же сообщение. Для посылки собственных сообщений внутри данного класса оконных компонентов следует использовать любое целое в диапазоне от **WM USER** до **0x7FFF.**

_rotl и другие функции циклического сдвига

Осуществляют циклический сдвиг целого числа без знака или кода символа.

Заголовочный файл *stdlib.h.*

Синтаксис

```
#include <stdlib.h>
unsigned short _rotl(unsigned short val, int count);
unsigned short _rotr(unsigned short val, int count);
unsigned long _lrotl(unsigned long val, int count);
unsigned long _lrotr(unsigned long val, int count);
unsigned char _crotl(unsigned char val, int count);
unsigned char _crotr(unsigned char val, int count);
```

Описание

Функции **_rotl, _rotr, _lrotl, _lrotr** осуществляют циклический сдвиг целого числа без знака **val** на **count** разрядов. Функции **_rotl и _lrotl** сдвигают влево, заменяя освобождающиеся младшие разряды старшими. Например, **_rotl(1, 15)** вернет 32768 — единицу младшего разряда, сдвинутую влево на 15 разрядов. А **_rotl(l, 16)** вернет 1 — поскольку единица младшего разряда, сдвинувшись влево на 15 разрядов, сдвинется еще на 1 разряд и перенесется в первый младший разряд.

Функции **_rotr и _lrotr** аналогично сдвигают число вправо, заменяя освобождающиеся старшие разряды младшими.

Функции **_crotl и _crotr** осуществляют аналогичный сдвиг влево или вправо кода символа. Например, **_crotl('1',0)** вернет 49 — код символа "1", а **_crotl('l',l)** вернет 98 — вдвое больший код (это код символа "b"). Соответственно, выражение **(char)_crotl('1',1)** вернет символ "b". Циклический сдвиг символа val на 8 разрядов вернет то же значение val, что пояснялось выше для функции **_rotl**.

_rotr — циклический сдвиг целого числа вправо

Осуществляет циклический сдвиг вправо целого числа.

См. разд. «_rotl и другие функции циклического сдвига».

RoundTo и другие функции округления

Округляют действительное число до заданного десятичного порядка.

Заголовочный файл *Math.hpp*.

Синтаксис

```
typedef Word TRoundToRange;

extern PACKAGE double__fastcall RoundTo(const double AValue,
                                const TRoundToRange ADigit);
extern PACKAGE double__fastcall SimpleRoundTo(
                        const double AValue,
                        const TSimpleRoundToRange ADigit = -2) ;
```

Описание

Функции **RoundTo и SimpleRoundTo** округляют действительное число до заданного десятичного порядка. Параметр **AValue** — округляемое число. Параметр **ADigit** указывает десятичный порядок, ниже которого производится округление. Он может принимать значения от -37 до 37 включительно. Например, если **ADigit = 3,** это значит, что будут округлены все разряды, младше 1000 (10^3). А если **ADigit** = -1, то будут округлены разряды, младше одной десятой (10^{-1}).

Функции **RoundTo и SimpleRoundTo** используют несколько разные алгоритмы в случаях, когда округляемое число расположено точно посередине между двумя значениями, имеющими заданное число значащих разрядов. Функция **RoundTo** в этом случае округляет до четного числа. А функция **SimpleRoundTo** в этом случае округляет до большего числа.

См. также разд. «Ceil, ceil, ceill, Floor, floor, floorl — функции округления действительных чисел» и «GetRoundMode, SetRoundMode — управление округлением».

Примеры

AValue	ADigit	RoundTo	SimpleRoundTo
1234567	3	1234000	1234000
1234500	3	1234000	1235000
1.23456	-3	1.235	1.235
-1.23456	-3	-1.235	-1.234

SameValue — сравнение действительных значений

Проверяет совпадение двух действительных значений.

См. разд. «CompareValue и другие функции сравнения числовых значений».

scanf и другие функции форматированного ввода

Вводят форматированные данные из входного потока (с клавиатуры), из файла, из буферного массива.

Заголовочные файлы *stdio.h, conio.h.*

Синтаксис

```
#include <stdio.h>
int scanf(const char *format[, address, ...]);
int wscanf(const wchar_t *format[, address, ...]);
int vscanf(const char *format, va_list arglist);

int fscanf(FILE *stream,
           const char *format[, address, ...]);
int fwscanf(FILE *stream,
            const wchar_t *format[, address, ...]);
int vfscanf(FILE *stream,
            const char *format,va_list arglist);

int sscanf(const char *buffer,
           const char *format[, address, ...]);

int swscanf(const wchar_t *buffer,
            const wchar_t *format[, address, ...]);
int vsscanf(const char *buffer,
            const char *format, va_list arglist);

#include <conio.h>
int cscanf(char *format[, address, ...]);
```

См. также методы **get** и **getline** — функции-элементы класса ifstream.

Описание

Все описанные ниже функции осуществляют ввод строки и форматированное преобразование ее полей в числа или символьные массивы. Если поле числовое, то считается, что оно закончилось, при появлении символа, который не может присутствовать в формате числа. А если поле — строка, то оно состоит из произвольной последовательности символов до разделителя — пробела, символа табуляции, символа новой строки и т.п. Функции производят ввод произвольного числа полей, преобразуют их в соответствии с заданными форматами и затем сохраняют преобразованные значения в числовых и символьных объектах, чьи адреса указаны в списке аргументов **argument.** Параметр **format** указывает строку форматирования. Она определяет способ преобразования отдельных полей и содержит спецификаторы, записываемые после символа "%". Подробнее вы можете посмотреть

в полном описании строки форматирования (см. разд. 3.1.3.2), а для приведенных ниже примеров достаточно знать, что спецификатор "%d" преобразует поле в целое число, спецификатор "%e" — в действительные, спецификатор "%s" • — в строку. Символ "*" между символом "%" и символом типа поля обеспечивает сканирование очередного поля без его преобразования и сохранения. Иначе говоря, это поле пропускается.

Функции форматированного ввода с клавиатуры

Функции **scanf** и **wscanf** обеспечивают форматированный ввод из стандартного потока ввода **stdin.** По умолчанию входной поток связан с клавиатурой. Параметр **format** указывает строку форматирования, которая применяется к множеству аргументов **argument,** расположенных в вызовах функций после строки форматирования.

Функции возвращают число успешно введенных, преобразованных и сохраненных полей. Если ни одно поле не сохранено, возвращается 0. Если делается попытка читать после конца входной строки, возвращается значение EOF.

Функции находят применение в основном в консольных приложениях, в которых поток **stdin** соответствует вводу с клавиатуры. Например:

```
#include <stdio.h>
int main(void)
{
 int I;
 float R;
 char S[80];
 puts("Enter integer number, float number,"
      "string:\n");
 if (scanf("%d %e %s", &I, &R, &S) < 3)
  puts("Wrong input\n");
 else printf("You enter %i, %g, %s", I, R, S);
 fflush(stdin);
 getchar();
}
```

Приведенный код сначала функцией **puts** выводит предложение пользователю ввести целое число, действительное число и строку. Для упрощения в этом коде использованы английские тексты (о выводе в консольных приложениях русских текстов см. в разд. «CharToOem, CharToOemBuff — перевод строки в текст DOS»). После этого осуществляется вызов функции **scanf** и в ее строке форматирования указывается, что ожидается чтение целого числа (спецификатор "%e"), действительного (спецификатор "%g"), и строки (спецификатор "%s"). После строки форматирования в вызове функции указаны адреса, по которым должны быть занесены результаты ввода. Обратите внимание, что указываются не переменные, воспринимающие результаты, а их адреса.

При выполнении этой команды пользователь видит окно DOS, в котором должен ввести требуемые данные. Он может разделять эти данные символами пробелов, символом табуляции, нажатием клавиши Enter. Например: "5 5.6 текст". В результате переменные получат значения I = 5, R = 5.6, S = "текст". Впрочем, тот же результат будет, если пользователь забудет ввести пробел перед текстом: "5 5.6текст". При появлении во входном потоке символа "т" функция ввода "поймет", что числовое поле кончилось, и последующие символы введет в строковую переменную. А вот если пользователь вообще забудет о разделителях и введет "55.6текст", то в результате переменные получат значения I = 55, R = 0.6, S = "текст". Иначе говоря, символ точки воспримется как окончание целого поля.

Функция **cscanf** аналогична по синтаксису **scanf** и тоже обеспечивает форматированный ввод с клавиатуры. Но в отличие от **scanf** она читает символы непосредственно с клавиатуры до их отображения на экране, производит их форматирование и затем выводит непосредственно на экран. Переносимость приложений,

использующих **cscanf**, ограничена, так что ее можно рекомендовать использовать только в исключительных случаях.

Функции форматированного ввода из текстового файла

Если желательно применять функции scanf и **wscanf** для чтения из текстового файла, то предварительно надо перенаправить поток stdin. Однако для форматированного ввода из файла естественнее использовать функции **fscanf и fwscanf**. Они работают так же, как **scanf,** а параметр stream указывает поток или файл, из которого осуществляется ввод. Например, операторы:

```
#include <stdio.h>
...
FILE *F;
int I;
double R;
char S[80];
if ((F = fopen("input.txt", "rt")) == NULL)
{
 ShowMessage("Файл не удается открыть");
 return;
}
F = fopen("input.txt", "rt");
if (fscanf(F, "%i %le %s ", &I, &R, &S) < 3)
 ShowMessage("Ошибка чтения из файла");
else
 ShowMessage("I = " + IntToStr(I) + ", R = " +
             FloatToStr(R) + ", S = " + S);
fclose(F);
```

обеспечивают чтение из файла *input.txt* записанного там целого числа, действительного числа и лексемы — символьной строки без разделителей. Если чтение прошло нормально, то возвращается число прочитанных и сохраненных полей (в данном примере 3). Если вернулось меньшее число, значит при чтении какого-то поля произошла ошибка.

Обратите внимание на то, что при чтении в переменную R в формате добавлен спецификатор "l". Связано это с тем, что тип R объявлен равным **double,** и надо добавлять этот спецификатор **long,** как указано в описании строки форматирования (см. разд. 3.1.3.2).

При чтении из файла иногда надо пропускать какие-то поля. Пусть, например, файл подготовлен какой-то программой, которая занесла в него строки некоторой таблицы в виде

```
X = 1 Y = 1
X = 2 Y = 4
...
```

И пусть вам надо прочитать в этих строках только численные значения переменных X и Y и занести результаты в соответствующие массивы. Сделать это можно следующим кодом:

```
FILE *F;
int i = 0;     I
double X[10], Y[10];
if ((F = fopen("input.txt", "rt")) == NULL)
{
 ShowMessage("Файл не удается открыть");
 return;
}
while (! feof(F))
{
 fscanf(F, "%*s %*s %le %*s %*s %le", &X[i], &Y[i]);
 i++;
}
```

```
fclose(F);
```

Здесь спецификаторы "%*s" обеспечивают пропуск символьных полей и сохраняют в памяти только цифровые значения.

Функции ввода из памяти

Функции **sscanf** и **swscanf** аналогичны рассмотренным ранее, но читают и форматируют данные не с клавиатуры, не из файла, а из массива символов **buffer.** Исходная строка в буфере может формироваться программно. А может предварительно читаться из файла без форматирования, а затем форматироваться функцией **sscanf.** Например, цикл в приведенном выше примере можно было бы организовать следующим образом:

```
char S[256];
...
while (! feof(F))
{
 fgets(S, 256, F);
 sscanf(S, "%*s %*s %le %*s %*s %le", &X[i], &Y[i]);
 i++;
}
```

Функция **fgets** читает в переменную S очередную строку, а затем из нее извлекаются число функцией **sscanf.** Преимущества подобной организации чтения связаны с тем, что ввод функцией **fgets** осуществляется много быстрее форматированного ввода функцией **fscanf,** а функция **sscanf,** работающая с памятью, работает тоже намного быстрее, чем **fscanf.**

Функции v...

Варианты рассмотренных функций с именами, начинающимися с символа "v", работают так же, как описанные выше, но в них передается не список аргументов, а указатель на список типа **va_list.** Это позволяет вам создавать собственную функцию ввода, принимающую произвольное число аргументов. Ниже приведен пример такой функции ге, в точности воспроизводящей функцию **fscanf.**

```
#include <stdio.h>
#include <stdarg.h>

int re(FILE *F,char *fmt, ...)
{
   va_list arg;
   int cnt;
   va_start(arg, fmt);
   cnt = vfscanf(F,fmt, arg);
   va_end(arg);
   return(cnt);
}
```

Вызов такой функции не отличается от вызова **fscanf** (кроме имени) и работает она точно так же. Но, конечно, реально имеет смысл создавать собственную функцию ввода только в том случае, если она должна чем-то отличаться от **fscanf:** использовать какие-то сложные строки форматирования, производить предварительную обработку вводимых данных и т.п.

SecondOf — дешифрация секунды

Определяет секунду.

См. разд. «DayOf и другие функции дешифрации дат и времени».

SecondOfTheMinute — дешифрация секунды

Определяет секунду.

См. разд. «DayOf и другие функции дешифрации дат и времени».

SecondsBetween и другие функции определения разности секунд

Возвращают число секунд между двумя значениями даты и времени.

Заголовочный файл *DateUtils.hpp*.

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE int__fastcall
                SecondsBetween(const System::TDateTime ANow,
                               const System::TDateTime AThen);
extern PACKAGE double__fastcall
                SecondSpan(const System::TDateTime ANow,
                           const System::TDateTime AThen);
```

Описание

Функции **SecondsBetween** и **SecondSpan** возвращают число секунд между двумя значениями даты и времени **ANow** и **AThen** типа **TDateTime.** Функция **SecondsBetween** возвращает число полных секунд между двумя значениями. А функция **SecondSpan** возвращает действительное число, содержащее дробную часть, отображающую неполную секунду.

Примеры

Операторы

```
TDateTime T1 = EncodeDateTime(2002, 10, 5, 11, 00, 45, 300);
TDateTime T2 = EncodeDateTime(2002, 10, 5, 11, 00, 46, 299);
int i = SecondsBetween(T2, T1);
double r = SecondSpan(T2, T1);
```

зададут переменной i значение 0, а переменной **r** значение 0,998999434523284. В этом примере значения дат и времени T1 и T2 задаются с помощью функции **EncodeDateTime.** Различие между двумя значениями чуть-чуть меньше секунды (не хватает одной миллисекунды). Поэтому функция **SecondsBetween** возвращает О, а функция **SecondSpan** возвращает число, близкое к единице.

SecondSpan — разность секунд

Возвращает число секунд между двумя значениями даты и времени.

См. разд. «SecondsBetween и другие функции определения разности секунд».

SelectDirectory — диалоги выбора каталога

Предоставляет пользователю возможность выбрать каталог с помощью стандартного диалога.

Заголовочный файл *FileCtrl.hpp*.

Синтаксис

```
#include <FileCtrl.hpp>
extern PACKAGE bool__fastcall SelectDirectory(
                const AnsiString Caption,
                const WideString Root, AnsiString &Directory);

enum TSelectDirOpt {sdAllowCreate, sdPerformCreate,
                    sdPrompt};
typedef Set<TSelectDirOpt, sdAllowCreate, sdPrompt>
                                        TSelectDirOpts;
extern PACKAGE bool__fastcall SelectDirectory(
                    AnsiString SDirectory,
                    TSelectDirOpts Options, int HelpCtx);
```

Описание

Функция **SelectDirectory** предоставляет пользователю возможность вызвать стандартный диалог Windows и, работая с ним, выбрать каталог. Функция имеет две перегруженных формы. Остановимся сначала на первой форме.

Функция вызывает стандартный диалог Windows для поиска каталога (папки), примеры которого приведены на рис. 4.6. Параметр **Caption** содержит строку, отображаемую в диалоге как указание пользователю (текст «Укажите каталог установки программы» на рис. 4.6. Параметр **Root** задает корневой каталог, внутри которого пользователь может выбирать подкаталоги. За пределы каталога **Root** пользователь выйти не может. При вызове **SelectDirectory** в примере рис. 4.6 а указано **Root** = "d:\\". Если указать вместо **Root** пустую строку или отсутствующий на компьютере каталог, то в диалоговом окне отобразится дерево всех папок (рис. 4.6 б), и пользователь имеет возможность выбрать на любом диске любой каталог.

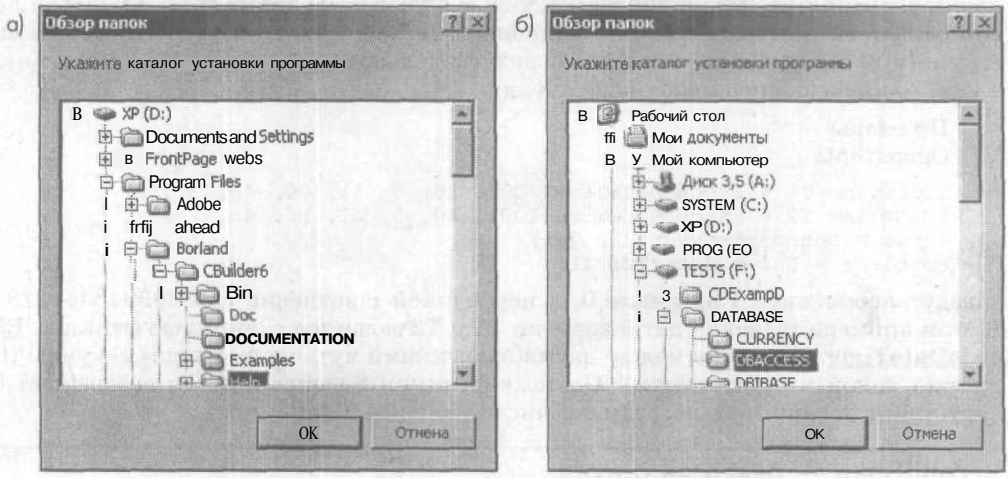


Рис. 4.6. Диалоговое окно поиска каталога при заданном (а) и не заданном (б) значении Root

Выходной параметр **Directory** содержит результат выбора пользователя. Функция возвращает **true**, если пользователь выбрал каталог и нажал OK. Если пользователь нажал Отмена или закрыл каталог, не произведя выбора, то функция возвращает **false.**

Рассмотрим пример. Пусть вы делаете программу установки вашего приложения и хотите, чтобы пользователь указал каталог, в котором надо установить программу. Соответствующий диалог можно оформить следующим образом:

```
#include <FileCtrl.hpp>
...
AnsiString Dir;
...
if(SelectDirectory("Укажите каталог установки программы", "", Dir))
  Dir = InputBox("Можете уточнить каталог",
                 "Программа расположится в каталоге:", Dir);
else
{
  Application->MessageBox("Вы не указали каталог",
                          "Установка прервана !", MB_ICONSTOP);
  Application->Terminate();
}
```

В этом примере вызов функции **SelectDirectory** приводит к появлению окна, представленного на рис. 4.6 б, поскольку параметр **Root** указан пустой строкой. Если пользователь выбрал каталог и нажал OK, то **SelectDirectory** возвращает **true.** В этом случае пользователю предлагается диалоговое окно, вызываемое функцией **Input Box.** Оно показано на рис, 4.7 а. В этом окне пользователь может, если хочет, уточнить каталог. Если же пользователь в окне рис. 4.6 б не выбрал каталог, то функцией **Application.MessageBox** вызывается окно, показанное на рис. 4.7 б, и установка прерывается.

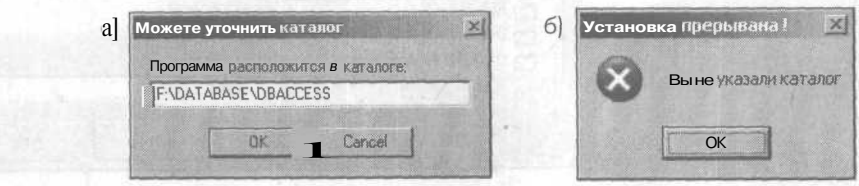


Рис. 4.6. Продолжение диалога выбора каталога

Теперь рассмотрим вторую форму функции **SelectDirectory.** Она предоставляет более гибкий диалог (рис. 4.8). Возвращаемое значение по-прежнему указывает, выбрал ли пользователь каталог. Параметр **Directory,** как и раньше, содержит выбранный пользователем каталог. Если перед вызовом **SelectDirectory** задано начальное значение **Directory,** то именно этот каталог будет раскрыт в окне диалога в первый момент времени. Параметр **HelpCtx** является ссылкой на контекстную справку, содержащую подсказку по действиям пользователя. А параметр **Options** является множеством следующих опций:

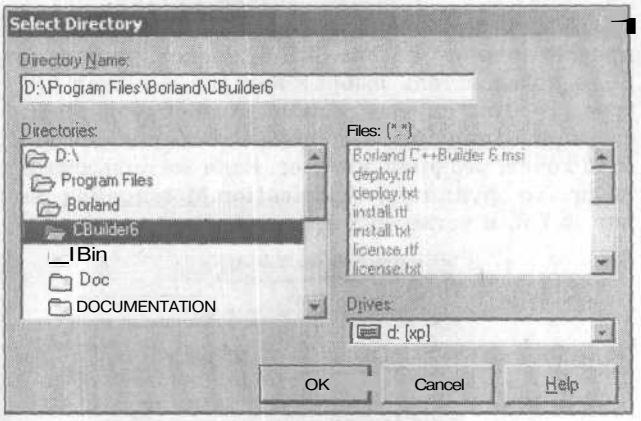
sdAllowCreate	В диалоговом окне отображается окошко редактирования Directory Name (см. рис. 4.8), в котором пользователь может написать каталог, который отсутствует. Эта опция не создает сам каталог. Это задача приложения, которое прочтет имя каталога и при необходимости создаст его.
sdPerformCreate	Применяется только в сочетании с **sdAllowCreate** и обеспечивает создание каталога, если указанный пользователем каталог отсутствует.
sdPrompt	Применяется только в сочетании с **sdAllowCreate.** Если пользователь указал несуществующий каталог, ему предлагается вопрос, надо ли его создавать. Если пользователь ответил утвердительно (нажал OK) и опция **sdPerformCreate** включена в множество **Options,** то каталог будет создан. Если же опция **sdPerformCreate** не задана, то приложение должно само создать нужный каталог.

Если множество **Options** пустое, то пользователь не может указать каталог, которого не существует.

Если применить эту форму функции **SelectDirectory** в приведенном выше примере, то начало кода может иметь вид:

```
#include <FileCtrl.hpp>
...
AnsiString Dir = "d:\\";
...
if(SelectDirectory(Dir, TSelectDirOpts() << sdAllowCreate
                   << sdPerformCreate << sdPrompt, 0))
...
```

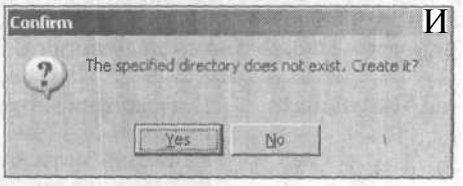
Рис. 4.8
Вторая форма диалога выбора каталога



В этом коде задается начальное значение каталога и в параметр **Options** включены все опции: **sdAllowCreate, sdPerformCreate и sdPrompt.** Вызываемое диалоговое окно подобно приведенному на рис. 4.8, но не будет иметь кнопки Help, поскольку идентификатор контекстной справки задан равным нулю.

Если пользователь напишет в окошке редактирования Directory Name каталог, который отсутствует *в* дереве, ему будет предложено окно запроса, показанное на рис. 4.9. Если пользователь нажмет в нем No, то вернется в окно рис. 4.8. Если же он нажмет Yes, то отсутствующий каталог будет создан.

Рис. 4.9
Запрос создания отсутствующего каталога



Как видно, вторая форма функции **SelectDirectory** дает дополнительную гибкость диалогу. Но она имеет существенный недостаток: окна рис. 4.8 и 4.9 содержат английские тексты. От окна запроса на создание каталога (рис. 4.9) безусловно лучше отказаться. Если допустимо создание нового каталога без дополнительного запроса, то следует задавать **Options** равным **[sdAllowCreate, sdPerformCreate].** Если же запрос все-таки нужен, то лучше задать **Options** равным **[sdAllowCreate],** а запрос и создание каталога обеспечить программно. Так что от английского окна запроса избавиться несложно. Но с английскими надписями в основном диалоговом окне (рис. 4.8) сделать ничего невозможно.

SendMessage — функция API Windows

Функция API Windows, посылает указанное в ней сообщение окну или множеству окон и не возвращается, пока это сообщение обрабатывается.

Модуль *winuser.*

Объявление

```
LRESULT SendMessage(
 HWND hWnd,      // дескриптор окна - приемника
 UINT Msg,       // сообщение
 WPARAM wParam,  // первый параметр сообщения
 LPARAM lParam   // второй параметр сообщения
);
```

Описание

Функция **SendMessage** посылает указанное в ней сообщение окну или всем окнам верхнего уровня в системе, включая недоступные и невидимые, кроме дочерних. Функция не возвращается, пока это сообщение обрабатывается. Таким образом, приложение, пославшее сообщение, блокируется на время его обработки. Этим функция **SendMessage** отличается от функции **PostMessage.** которая возвращается сразу после передачи сообщения.

Параметр **hWnd** — дескриптор окна, которому передается сообщение. Если этот параметр равен **HWND_BROADCAST,** то сообщение передается всем окнам верхнего уровня в системе, включая недоступные, невидимые, перекрытые другими и всплывающие, за исключением дочерних окон.

Параметр Msg определяет передаваемое сообщение. Параметры **wParam** и **1Parат** могут содержать дополнительную информацию.

Значение, возвращаемое функцией, зависит от вида сообщения.

Для определения дескриптора, передаваемого в функцию в качестве параметра **hWnd,** можно использовать функцию **FindWindow.**

Приложения, использующие **hWnd = HWND_BROADCAST** для связи между окнами разных приложений, должны предварительно зарегистрировать уникальность своих сообщений функцией **RegisterWindowMessage.**

Примеры

Оператор

```
SendMessage(Form2->Handle,WM_CLOSE,0,0);
```

посылает форме Form2 сообщение **WM_CLOSE,** закрывающее окно этой формы. Первый параметр функции **SendMessage** содержит дескриптор окна формы **Form2,** полученный с помощью ее свойства **Handle.** Сообщение **WM_CLOSE** не имеет параметров; поэтому параметры **wParam и lParam** заданы равными нулю.

Оператор

```
SendMessage(FindWindow("SciCalc","Калькулятор"), WM_CLOSE,0,0);
```

использует функцию **FindWindow** для получения дескриптора окна приложения, которому надо послать сообщение **WM_CLOSE.** В качестве параметров в **FindWindow** передаются класс формы SciCalc и ее заголовок «Калькулятор». Это стандартное приложение Windows «Калькулятор».

См. также пример в разд. «FindWindow».

set_new_handler и другие функции обработки ошибок выделения памяти

Функция **set_new_handler** устанавливает заказной обработчик события, связанного с невозможностью динамически выделить требуемый блок памяти.

Заголовочный файл *new.h.*

Синтаксис

```
typedef void (new * new_handler)();
new_handler set_new_handler(new_handler my_handler);

typedef void (*pvf)();
pvf _new_handler;
```

Описание

При динамическом выделении памяти операцией new (см. в гл. 1 разд. 1.11) может оказаться, что выделить блок требуемого размера невозможно. В этом случае генерируется исключение **bad_alloc,** которое можно перехватывать в блоке **catch.** Но можно отменить генерацию исключения **bad_alloc,** задавая указатель на свой собственный обработчик событий, связанных с невозможностью выделить па-

мять. Задание этого обработчика можно осуществить или непосредственным присваиванием указателя на него указателю на функцию **_new_handler**, или с помощью функции **set_new_handler.** Возможности функции **set_new_handler** шире и использовать ее проще. Так что далее рассмотрен этот более современный вариант задания обработчика. А функция **_new_handler** оставлена в C++ для обратной совместимости с версией C++ 1.2.

В качестве аргумента **my_handler** в функцию **set_new_handler** передается указатель на введенный вами обработчик. Функция возвращает прежний указатель, который был зарегистрирован до этого.

Например, вы можете описать функцию

```
void F1(void)
{
 ShowMessage("Не хватает памяти");
 exit(1);
}
```

которая обрабатывает ситуацию, связанную с нехваткой памяти, и ввести в программу (например, в обработчик события **OnCreate** формы) оператор

```
set_new_handler(F1);
```

Вводимый таким образом обработчик не может ничего возвращать и должен или освободить память для успешного повторного выполнения new, или сгенерировать исключение **bad_alloc,** или завершить программу (это сделано в приведенном примере). Если не выполнено ни одно из этих действий, возникнет бесконечный цикл обращений к обработчику.

Если обработчик не прерывает выполнение, то после него повторно выполняется операция new, что позволяет надеяться на успех, если обработчик освободил память.

Можно отменить генерацию исключения bad_alloc, не вводя специального обработчика, а просто записав оператор

```
set_new handler(0);
```

В этом случае при недостатке памяти операция new будет возвращать **NULL.** Тогда проверку можно строить, проверяя, не равен ли значению **NULL** указатель, возвращенный new.

Set8087CW — установка управляющего слова FPU

Обеспечивает установку управляющего слова FPU.

См. разд. «_control87 и другие функции доступа к управляющему слову FPU».

SetExceptionMask — установку масок исключений

Обеспечивает установку и запоминание масок исключений управляющего слова FPU.

См. разд. «GetExceptionMask и другие функции доступа к маскам исключений».

SetPrecisionMode — управление точностью

Обеспечивает доступ к битам управления точностью управляющего слова FPU.

См. разд. «GetPrecisionMode и другие функции управления точностью».

SetRoundMode — управление округлением

Обеспечивает доступ к битам управления округлением управляющего слова FPU.

См. разд. «GetRoundMode и другие функции управления округлением».

ShellExecute — функция API Windows

Открывает или печатает указанный файл или открывает указанную папку.

Модуль *ShellAPI*.

Определение

```
HINSTANCE ShellExecute(
 HWND hwnd,              // дескриптор родительского окна
 LPCTSTR lpOperation,    // строка выполняемой операции
 LPCTSTR lpFile,         // строка с именем файла или папки
 LPCTSTR lpParameters,   // строка параметров выполняемого файла
 LPCTSTR lpDirectory,    // строка каталога по умолчанию
 INT nShowCmd            // режим открытия файла
);
```

Описание

Функция **ShellExecute** позволяет выполнить любое приложение Windows. Можно также открыть файл документа, что означает выполнение связанного с ним приложения и загрузку в него этого документа. Например, обычно с документами, имеющими расширение **.doc,** связан Word. В этом случае открыть файл, например, с именем "file.doc" означает запустить Word и передать ему в качестве параметра имя файла "file.doc". Кроме описанных возможностей функция **ShellExecute** позволяет распечатать указанный файл или открыть указанную папку. Последнее означает, что будет запущена программа «Проводник» с открытой указанной папкой.

Для использования функции **ShellExecute** в модуль надо добавить директиву препроцессора

```
#include "ShellAPI.h"
```

подключающую модуль *ShellAPI*, в котором описана функция. Автоматически C++Builder эту директиву не добавляет.

Параметр **hwnd** является дескриптором родительского окна, в котором отображаются сообщения запускаемого приложения. Обычно в качестве него можно просто указать **Handle.**

Параметр **lpOperation** указывает на строку с нулевым символом в конце, которая определяет выполняемую операцию. Эта строка может содержать текст "open" (открыть) или "print" (напечатать). Для 32-разрядных Windows определено еще одно значение: "explore" (исследовать) — открыть папку программой Windows «Проводник». Если параметр **lpOperation** равен **NULL,** то по умолчанию выполняется операция "open".

Параметр lpFile указывает на строку с нулевым символом в конце, которая определяет имя открываемого файла или имя открываемой папки.

Параметр lpParameters указывает на строку с нулевым символом в конце, которая определяет передаваемые в приложение параметры, если **lpFile** определяет выполняемый файл. Если **lpFile** указывает на строку, определяющую открываемый документ или папку, то параметр **lpParameters** задается равным NULL.

Параметр lpDirectory указывает на строку с нулевым символом в конце, которая определяет каталог по умолчанию.

Параметр **nShowCmd** определяет режим открытия указанного файла. Этот параметр может принимать следующие значения:

SW_HIDE	Окно делается невидимым и фокус передается другому окну.
SW_MINIMIZE	Свертывает (минимизирует) указанное окно и активизирует следующее в Z-последовательности окно верхнего уровня в списке системы.

SW_MAXIMIZE	Развертывает (максимизирует) указанное окно.
SW_RESTORE	Активизирует и отображает окно. Если это окно свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции (почти то же самое, что SW_SHOWNORMAL).
SW_SHOW	Активизирует и отображает окно в его текущей позиции и с текущими размерами.
SW_SHOWDEFAULT	Устанавливает состояние в соответствии с флагом SW_ в структуре STARTUPINFO, передаваемой в функцию CreateProcess программой, запускающей приложение. Приложение должно вызывать Show Window с этим флагом, чтобы задать начальное состояние своего главного окна.
SW_SHOWMAXIMIZED	Активизирует и отображает окно в развернутом виде (максимизированном).
SW_SHOWMINIMIZED	Активизирует и отображает окно в свернутом виде (в виде пиктограммы).
SW_SHOWMINNOACTIVE	Отображает окно в свернутом виде (в виде пиктограммы). Активным остается то окно, которое было активным до этого.
SW_SHOWNA	Отображает окно в его текущей позиции и с текущими размерами. Активным остается то окно, которое было активным до этого.
SW_SHOWNOACTIVATE	Отображает окно в его последней позиции и с последними размерами. Активным остается то окно, которое было активным до этого.
SW_SHOWNORMAL	Активизирует и отображает окно. Если это окно свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции (почти то же самое, что SW_RESTORE).

Чаще всего используется значение SW_RESTORE, при котором окно запускаемого приложения активизируется и отображается на экране. Если это окно в данный момент свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции. Для приложений не Windows, для файлов PIF и т.д. состояние окна определяет само приложение.

Функция ShellExecute возвращает дескриптор открытого приложения или дескриптор сервера DDE приложения. Если возвращаемое значение меньше или равно 32, это указывает на ошибку.

Примеры

Пусть вы хотите открыть файл документа с именем "file.doc", т.е. запустить Word (обычно именно он связан с файлами .doc), загрузив в него указанный файл. Тогда вы можете написать:

```
ShellExecute(Handle,NULL,"file.doc",NULL,NULL,SW_RESTORE);
```

Если вы хотите не открыть, а напечатать документ, записывается аналогичный оператор, но изменяется значение параметра lpOperation:

```
ShellExecute(Handle,"print","file.doc",NULL,NULL,SW_RESTORE);
```

Выполнение этого оператора будет протекать следующим образом. Запустится Word, связанный с файлами **.doc,** в него загрузится файл **file.doc,** затем из Word запустится печать с атрибутами по умолчанию, после чего файл **file.doc** выгрузится из Word.

Приведенный ниже оператор открывает приложение Windows «Калькулятор»:

```
ShellExecute(Handle,"open","Calc",NULL,NULL,SW_RESTORE);
```

Следующий пример открывает папку **c:\Program Files\Borland:**

```
ShellExecute (Handle,"open","c:\\Program Files\\Borland",
              NULL,NULL,SW_RESTORE);
```

А оператор

```
ShellExecute(Handle,"explore","c:\\Program Files\\Borland",
              NULL,NULL,SW_RESTORE);
```

открывает программу «Проводник» с открытой папкой **c:\Program Files\Borland.**

SHGetFileInfo — получение информации об объекте файловой системы

Функция API Windows, позволяет получить разнообразную информацию об объекте файловой системы: файле, папке, диске.

Заголовочный файл *ShellAPI.h.*

Синтаксис

```
#include <ShellAPI.h>
unsigned int SHGetFileInfoA(const char * pszPath,
                            unsigned long dwFileAttributes,
                            _SHFILEINFOA *psfi,
                            unsigned int cbFileInfo,
                            unsigned int uFlags);
```

Описание

Функция **SHGetFilelnfo** дает возможность получить разнообразную информацию об объекте файловой системы: файле, папке, диске. В частности, можно получить доступ к спискам пиктограмм Windows. Параметр **pszPath** задает файл или шаблон файлов, о которых требуется получить информацию. Параметр **dwFileAttributes** определяет атрибуты файла и используется только при флаге **SHGFI_USEFILEATTRIBUTES.** В остальных случаях этот параметр игнорируется и его можно задавать равным 0. В передаваемую по ссылке структуру **psfi** типа **_SHFILEINFOA** или эквивалентного ему типа **TSHFileInfo** функция заносит полученную информацию. Параметр **cbFilelnfo** задает размер структуры **psfi.** А параметр uFlags включает наборы флагов. Возвращаемое функцией значение зависит от включенных флагов.

Структура **psfi** содержит поля:

поле	тип
hIcon	HICON
iIcon	int
dwAttributes	DWORD
szDisplayName	char — массив длиной 80 элементов
szTypeName	char — массив длиной 80 элементов

Параметр **uFlags** может включать следующие флаги:

SHGFI_ATTRIBUTES	Получение атрибутов файла. Флаги атрибутов заносятся в поле **dwAttributes** структуры psfi.
SHGFI_DISPLAYNAME	Получение полного имени файла. Имя заносятся в поле szDisplayName структуры psfi.
SHGFI_EXETYPE	Если **pszPath** — имя выполняемого файла, то функция возвращает тип файла (см. пояснения ниже).
SHGFI_ICON	Доступ к дескриптору пиктограммы файла или к системному списку пиктограмм. Применяется вместе с рядом модификаторов. Дескриптор заносится в поле **hIcon** структуры psfi, а индекс пиктограммы — в поле **iIcon**. Функция возвращает дескриптор системного списка пиктограмм.
SHGFI_ICONLOCATION	Доступ к имени файла, содержащего пиктограмму данного файла. Имя заносится в поле szDisplayName структуры psfi.
SHGFI_LARGEICON	Модификатор SHGFI_ICON, дающий доступ к списку больших пиктограмм.
SHGFI_LINKOVERLAY	Модификатор SHGFI_ICON, дающий оверлейную связь с пиктограммами файлов.
SHGFI_OPENICON	Модификатор SHGFI_ICON, дающий доступ к списку пиктограмм, соответствующих открытому контейнеру (папке, каталогу).
SHGFI_PIDL	Указывает, что pszPath — адрес структуры ITEMIDLIST, а не имя файла.
SHGFI_SELECTED	Модификатор SHGFI_ICON, дающий доступ к списку пиктограмм с цветом, соответствующим выделению.
SHGFI_SHELLICONSIZE	Модификатор SHGFI_ICON, дающий доступ к списку пиктограмм, размер которых определяется средой.
SHGFI_SMALLICON	Модификатор SHGFI_ICON, дающий доступ к списку малых пиктограмм.
SHGFI_SYSICONINDEX	Доступ к индексу пиктограммы в системном списке. Индекс заносится в поле ilcon структуры psfi. Функция возвращает дескриптор системного списка пиктограмм.
SHGFI_TYPENAME	Доступ к строке описания типа файла. Строка заносится в поле **szTypeName** структуры psfi.
SHGFI_ USEFILEATTRIBUTES	Указывает, что функция должна использовать параметр **dwFileAttributes**.

Если вы хотите определить тип исполняемого файла, в параметре **uFlags** надо задать только **SHGFI_EXETYPE**. Тогда возвращаемое функцией **SHGetFileInfo** значение указывает тип файла:

0	невыполняемый файл или ошибка
LOWORD = **NE** или PE (17744) **HIWORD** = 3.0, 3.5 или 4.0	приложение Windows

LOWORD = MZ (23117) HIWORD = 0	файлы MS-DOS .EXE, .COM или .BAT
LOWORD = PE (17744)\ HIWORD = 0	консольное приложение Win32

Типичное применение функции **SHGetFileInfo** — доступ к системным спискам пиктограмм. Сочетание флагов **SHGFI_SMALLICON or SHGFI_ICON or SHGFI_SYSICONINDEX** или **SHGFI_LARGEICON or SHGFI_ICON or SHGFI_SYSICONINDEX** дает соответственно доступ к спискам малых и больших пиктограмм Windows.

Примеры

Ниже приведен пример определения типа исполняемого файла. Пусть в вашем приложении имеется кнопка, при щелчке на которой пользователь может с помощью диалога **OpenDialog1** выбрать файл и получить о нем информацию. Это может быть реализовано следующим образом:

```
void _fastcall TForm1 : :Button1Click(TObject *Sender)
{
 TSHFileInfo fi;

 if(OpenDialog1->Execute())
 {
  unsigned int W = SHGetFileInfo (
(OpenDialog1->FileName) .c_str() , 0,
                    Sfi, sizeof(fi), SHGFI_EXETYPE);
  if  (W == 0)
    ShowMessage("Файл '" + OpenDialog1->FileName +
                "' неисполняемый");
  else if  ( (LOWORD(W) == 17744)  && (HIWORD(W) > 0))
    ShowMessage("Файл '" + OpenDialog1->FileName +
                "' - приложение Windows");

  else if  ( (LOWORD(W) == 23117) && (HIWORD(W) == 0))
    ShowMessage("Файл '" + OpenDialog1->FileName +
                "' - приложение MS-DOS");
  else if  ( (LOWORD(W) == 17744) && (HIWORD(W) == 0))
    ShowMessage("Файл '" + OpenDialog1->FileName +
                "' - консольное приложение Win32");
 }
}
```

Следующий пример демонстрирует доступ к системным пиктограммам. Приведенный ниже код создает объекты списков изображений и связывает их с системными списками:

```
TImageList *SmallImages = new TImageList(Form1);
TImageList *LargeImages = new TImageList(Form1);

void _fastcall TForm1::FormCreate(TObject *Sender)
{
 TSHFileInfo  fi;

 SmallImages->Handle = SHGetFileInfo("*.*", 0, &fi,
                    sizeof(fi), SHGFI_SMALLICON |
                    SHGFI_ICON | SHGFI_SYSICONINDEX);
 LargeImages->Handle = SHGetFileInfo("*.*", 0, &fi,
                    sizeof(fi), SHGFI_LARGEICON |
                    SHGFI_ICON | SHGFI_SYSICONINDEX);
 DrawGrid1->ColCount = 10;
 DrawGrid1->RowCount = ceil(LargeImages->Count / 10);
}
```

В этом коде объявляются (и создаются операцией new) две переменные **SmallImages и LargeImages** типа **TImageList.** А в обработчике события формы **OnCreate** в качестве их дескрипторов задаются ссылки на списки соответствующих пиктограмм Windows. В результате список **SmallImages** будет соответствовать списку малых пиктограмм, а список **LargeImages** — списку больших пиктограмм. Далее их можно использовать в приложении как обычные компоненты типа **TImageList.** Например, если ввести в приложение таблицу изображений **TDrawGrid** достаточной вместимости, то следующий обработчик ее события **OnDrawCell** обеспечит отображение в ней списка малых системных пиктограмм:

```
void __fastcall TForm1::DrawGrid1DrawCell(TObject *Sender, int ACol,
                          int ARow, TRect &Rect, TGridDrawState State)
{
 SmallImages->Draw(DrawGrid1->Canvas, Rect.Left+10, Rect.Top+10,
                   ARow * DrawGrid1->ColCount + ACol,true);
}
```

Для того чтобы обеспечить достаточную вместимость таблицы **DrawGrid1,** можно добавить в конце приведенного ранее обработчика события формы **OnCreate** операторы:

```
DrawGrid1->ColCount = 10;
DrawGrid1->RowCount = ceil(SmallImages->Count / 10);
```

ShowMessage и другие функции вывода простых диалоговых окон сообщений

Отображают простые диалоговые окна сообщений.

Заголовочный файл *Dialogs.hpp.*

Синтаксис

```
#include <Dialogs.hpp>
extern PACKAGE void __fastcall ShowMessage(constAnsiString Msg);
extern PACKAGE void __fastcall ShowMessageFmt(constAnsiString Msg,
                               const System::TVarRec *Params,
                               const int Params_Size);
extern PACKAGE void __fastcall ShowMessagePos(constAnsiString Msg,
                               int X, int Y);
```

Описание

В приложениях часто приходится отображать различные простые диалоговые окна, чтобы дать пользователю какие-то указания. В законченном приложении желательно эти окна проектировать самому, обеспечивая единство стиля всех окон приложения, русские надписи на кнопках и т.п. Но при разработке прототипа будущего проекта и в процессе отладки удобно пользоваться готовыми диалоговыми окнами и вызывающими их функциями.

Простейшей из таких функций является **ShowMessage,** отображающая окно сообщения с кнопкой OK. Текст сообщения задается параметром Msg. Заголовок окна совпадает с именем выполняемого файла приложения.

Функция **ShowMessagePos** выводит такое же окно, как и функция **ShowMessage,** но позволяет указать координаты левого верхнего угла окна X и Y. Это в ряде случаев позволяет привязать окно сообщения к тому окну формы или к тому его компоненту, к которому данное сообщение относится. Подобная привязка помогает пользователю понять, на что конкретно указывает данное сообщение.

Функция **ShowMessageFmt,** позволяет выводить в диалоговое окно форматированное сообщение. Параметр Msg в этой функции задает строку описания формата (см. разд. 3.1.3.3), а параметры **Params и Params_Size** задают массив параметров, форматируемых строкой Msg, и размер этого массива. Для передачи мас-

сива в функцию удобно использовать макрос **OPENARRAY** (см. разд. 2.11.3). Тогда вызов функции **ShowMessageFmt** имеет вид:

```
ShowMessageFmt(Msg,OPENARRAY(TVarRec,(arg1,arg2,...)));
```

Более содержательные диалоговые окна, позволяющие задать пользователю вопрос и узнать его реакцию, отображаются функциями **MessageDlg.** Message-**DlgPos и CreateMessageDialog.** См. также метод **MessageBox.** обеспечивающий, пожалуй, наиболее удачное полностью русифицируемое диалоговое окно.

Примеры

Операторы

```
ShowMessage("Работа приложения успешно завершена");
ShowMessageFmt("Задано %d параметров из %d ",
               OPENARRAY(TVarRec,(N1,N2)));
```

вызывают диалоговые окна, вид которых показан на рис. 4.10.

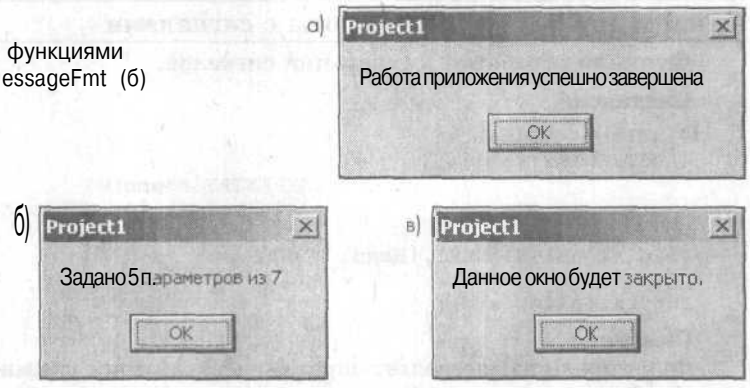


Рис. 4.10
Сообщения, выдаваемые функциями ShowMessage (а), ShowMessageFmt (б) и ShowMessagePos (в)

Операторы

```
ShowMessagePos("Данное окно будет закрыто",
               Form2->Left, Form2->Top + Form2->Height);
Form2->Close();
```

перед закрытием окна формы **Form2** выводят диалоговое окно с сообщением «Данное окно будет закрыто» (рис. 4.10 в), привязанное к левому нижнему углу формы **Гогт2.**

ShowMessageFmt — простое диалоговое окон с форматированным сообщением

Отображает простое диалоговое окно с форматированным сообщением.

См. разд. «ShowMessage и другие функции вывода простых диалоговых окон сообщений».

ShowMessagePos — простое диалоговое окон с сообщением в заданной позиции

Отображает в заданной позиции простое диалоговое окно с сообщением.

См. разд. «ShowMessage и другие функции вывода простых диалоговых окон сообщений».

Sign — функция

Определяет знак аргумента.

Заголовочный файл *Math.hpp.*

Синтаксис

```
extern PACKAGE int __fastcall Sign(const double AValue);
extern PACKAGE int __fastcall Sign(const int AValue);
extern PACKAGE int __fastcall Sign(const __int64 AValue);
```

Описание

Перегруженные формы функции Sign определяют знак аргумента AValue. Они возвращают:

0	**если AValue = 0**
-1	**если AValue < 0**
+1	**если AValue > 0**

signal и другие функции работы с сигналами

Функции обработки и генерации сигналов.

Синтаксис

```
#include <signal.h>
(_USERENTRY *signal(int sig,
                    void (JJSERENTRY *func)
                         (int sig[, int subcode])))(int);

void signal(SIGUSR1, Handl_SIGUSR1);

int raise(int sig);
```

Описание

Функция signal позволяет определить обработчик указанного сигнала прерывания sig. Функция raise генерирует сигнал типа sig (см. о сигналах в гл. 1, разд. 1.13).

В файле *signal.h* предопределены следующие сигналы:

SIGABRT	Аварийное завершение программы. Генерируется только вызовом функций abort, raise и необработанными исключениями. Действие по умолчанию — вызов _exit(3).
SIGBREAK	Прерывание нажатием клавиш Ctrl-Break.
SIGFPE	Ошибка арифметической операции, например, деления на нуль или операции, вызвавшей переполнение. Действие по умолчанию — вызов _exit(l).
SIGILL	Появление в коде недопустимой команды. Действие по умолчанию — вызов _exit(1).
SIGINT	Получение интерактивного сигнала (например, прерывание Ctrl+C). Действие по умолчанию — прерывание INT 23h.
SIGSEGV	Нарушение доступа к памяти. Действие по умолчанию — вызов _exit(l).
SIGTERM	«Мягкое» завершение процесса. Действие по умолчанию — вызов _exit(l).
SIGUSR1, SIGUSR2, SIGUSR3	Определенные пользователем (только в Win32) сигналы пользователя, генерируемые функцией raise. Действие по умолчанию — игнорирование сигнала.

Функция **signal** имеет две формы, приведенные выше. Первая из них основная, вторая сохраняется для обратной совместимости и не рекомендуется в новых приложениях.

В функцию **signal** передаются два параметра: целочисленный номер сигнала sig и указатель **func** на функцию обработки сигнала. Обработчик **func** может быть определенной пользователем функцией или одним из трех предопределенных обработчиков: **SIG_DFL** — завершение программы, **SIG_IGN** — игнорирование сигналов данного вида, **SIG_ERR** — генерация ошибки. Например:

```
// генерация сообщения об ошибке
signal(SIGINT, SIG_ERR);
```

или

```
// игнорирование сигнала SIGINT
signal(SIGINT, SIG_IGN);
```

Если вы хотите задать свой обработчик стандартного сигнала или определенного вами сигнала, это может выглядеть так. Пусть, например, вы хотите предусмотреть в своей программе обработчик некоторого вводимого вами сигнала **SIGUSR1**. Назовем функцию этого обработчика **Handl_SIGUSR1**. Определите в программе тип указателя на функцию **fptr:**

```
typedef void (*fptr)(int);
```

Тогда где-то в начале программы надо ввести оператор:

```
signal(SIGFPE, (fptr)Handl_SIGFPE);
```

Этот оператор установит функцию **Handl_SIGUSR** как обработчик сигнала **SIGUSR1. В** нужных местах программы вставьте оператор генерации вашего сигнала:

```
raise(SIGUSR1);
```

Рассмотрим подробнее последовательность операций системы при выполнении этого оператора. Если в программе задан приведенным выше оператором **signal** обработчик пользователя для данного события, то происходит обращение к этому обработчику, но прежде система сбрасывает установку на **SIG_DFL.** Это значит, что при следующей генерации этого сигнала, если не повторить вызов функции **signal,** система забудет прежнюю установку и не будет опять обращаться к обработчику пользователя. Поэтому обычно в конце обработчика повторяют вызов **signal.**

Примеры

Ниже приведен пример определения обработчика **Handl_SIGUSRl** вводимого пользователем типа сигнала **SIGUSR1.**

```
#include <signal.h>
typedef void (*fptr)(int);
void Handl_SIGUSR1(int N)
{
 ...
 if (...)
 i
  ...
  // повторная установка обработчика для продолжения работы:
  signal(SIGFPE, (fptr)Handl_SIGFPE);
 }
 else exit(EXIT_SUCCESS);
}
```

Этот обработчик принимает одно целое значение, соответствующее номеру сигнала. В обработчике предусматриваются некоторые действия, необходимые при появлении данного сигнала. Затем, если выполнение программы должно продолжаться, надо повторно установить обработчик сигнала с помощью функции **signal,** как показано в приведенном примере. Если этого не сделать, то последующие со-

бытия **SIGUSR1** не будут вызывать этот обработчик. После выполнения команды повторной установки обработчика сигнала управление автоматически передается в точку программы, в которой сигнал был обнаружен. В этом, в частности, коренное отличие сигналов от исключений.

Указание системе на этот обработчик осуществляется оператором

```
signal(SIGFPE, (fptr)Handl_SIGFPE);
```

Такой оператор вы можете вставить, например, в обработчик события **OnCreate** вашей формы.

Генерация сигнала в требуемых местах программы осуществляется с помощью функции raise оператором

```
raise(SIGUSR1);
```

SimpleRoundTo — округление

Округляет действительное число до заданного десятичного порядка.

См. разд. «RoundTo и другие функции округления».

Sleep — функция задержки выполнения

Задерживает выполнение приложения на заданный интервал времени.

Заголовочный файл *winbase.h*.

Синтаксис

```
void Sleep (DWORD dwMilliseconds);
```

Описание

Функция **Sleep** обеспечивает задержку выполнения текущего потока (нити) на **dwMilliseconds** миллисекунд. На это время управление переключается на другие процессы с тем же или более высоким приоритетом.

Например, оператор

```
Sleep(10000);
```

задержит выполнение приложения на 10 сек.

Иногда надо задержать выполнение какой-то функции, но при этом сохранить возможность управлять приложением. Это можно сделать, разбив одну задержку на ряд более коротких и в промежутках между ними вызывать метод **ProcessMessages** приложения **Application.**

Например, создайте форму с двумя кнопками и одним окном редактирования. В обработчик первой кнопки внесите операторы:

```
Sleep(10000);
ShowMessage("End");
...
```

При щелчке на этой кнопке выполнение приложение остановится на 10 сек. В течение этого времени вы не сможете им управлять: нажать какую-то другую кнопку, ввести текст в окно редактирования, изменить размер формы. А в обработчик второй кнопки внесите код:

```
for(int i=1; i <= 100; i++)
{
 Sleep(100);
 Application->ProcessMessages();
}
ShowMessage("End");
...
```

Щелчок на этой кнопке также задержит выполнение процедуры его обработки на 10 сек. Но через каждые 100 миллисекунд будет выполняться метод **Process-**

Messages, который обеспечит реакцию на все сообщения Windows. В результате управление приложением не будет потеряно: вы сможете на протяжении задержки вводить текст в окно редактирования, нажать первую кнопку, изменить размер формы и т.п.

См. также один из примеров в разд. «fgetc и другие функции ввода/вывода символа».

Реализация функции Sleep зависит от платформы. В Windows инкапсулируется соответствующая функция API. В Linux функция реализуется совершенно иначе. Но и в приложениях Windows, и в консольных приложениях, и в приложениях Linux она работает.

Для задержки выполнения или какой-то синхронизации процессов может также использоваться компонент Timer или таймеры Windows.

SmallPoint — формирование точки из координат

Формирует точку из координат.

См. разд. «Point и другие функции формирования точки».

spawn... — функции выполнения порождаемых процессов

Порождают новый процесс.

Заголовочный файл *process.h*.

Синтаксис

```
#include <process.h>
int spawnl(int mode, char *path, char *argO, arg1,
           ..., argn, NULL);
int _wspawnl (int mode, wchar_t *path, wchar_t *argO,
           arg1, ..., argn, NULL);

int spawnle(int mode, char *path, char *arg0, arg1,
           ..., argn, NULL, char *envp[]);
int _wspawnle(int mode, wchar_t *path, wchar_t *argO,
           arg1, ..., argn, NULL, wchar_t *envp[]);

int spawnlp(int mode, char *path, char *argO, arg1,
           ..., argn, NULL);
int _wspawnlp(int mode, wchar_t *path, wchar_t *argO,
           arg1, ..., argn, NULL);

int spawnlpe(int mode, char *path, char *arg0, arg1,
           ..., argn, NULL, char *envp[]);
int _wspawnlpe(int mode, wchar_t *path, wchar_t *argO,
           arg1, ..., argn, NULL, wchar_t *envp[]);

int spawnv(int mode, char *path, char *argv[]);
int _wspawnv(int mode, wchar_t *path,
           wchar_t *argv[]);

int spawnve(int mode, char *path, char *argv[],
           char *envp[]);
int _wspawnve(int mode, wchar_t *path,
           wchar_t *argv[], wchar_t *envp[]);

int spawnvp(int mode, char *path, char *argv[]);
int _wspawnvp(int mode, wchar_t *path,
           wchar_t *argv[]);

int spawnvpe(int mode, char *path, char *argv[],
           char *envp[]);
```

```
int _wspawnvpe(int mode, wchar_t *path,
               wchar_t *argv[], wchar_t *envp[]);
```

Описание

Функции **spawn...** загружают в память и выполняют некоторую внешнюю программу **path,** называемую порожденным процессом.

Имеется родственное рассматриваемому семейству функций семейство функций **exec...** также решающее задачи порождения процессов. Но функции **spawn...** обладают более широкими возможностями, чем **exec...,** благодаря наличию параметра **mode,** задающего режим выполнения порождаемого процесса. Этот параметр может принимать следующие значения:

P_WAIT	Родительский процесс ждет завершения порожденного процесса, после чего продолжается выполнение родительского процесса.
P_NOWAIT	Родительский процесс продолжает выполняться пока выполняется порожденный процесс. Поскольку функция возвращает ID порожденного процесса, можно применить функцию cwait или wait, чтобы обеспечить ожидание завершения порожденного процесса. Этот режим недоступен в 16-разрядных Windows и DOS.
P_NOWAITO	Идентичен **P_NOWAIT,** но ID порожденного процесса не сохраняется операционной системой, так что применение функций cwait или wait невозможно.
P_DETACH	Идентичен **P_NOWAITO,** но порожденный процесс выполняется в фоновом режиме, так что не имеет доступа к клавиатуре и дисплею.
P_OVERLAY	Порождаемый процесс замещает в памяти родительский. То же, что вызов соответствующей функции **exec...**

Различия между функциями семейства **spawn...** определяются их суффиксами, которые обозначают следующее:

L	В процесс передается список указателей на аргументы **arg0, arg1,** ..., argn. Обычно используется, если число аргументов заранее известно.
v	В процесс передается указатель **argv[]** на массив указателей на аргументы **arg0,** arg1, ..., **argn.** Обычно используется, если число передаваемых аргументов может изменяться.
P	Файл загружаемой программы ищется в каталогах, указанных в переменной окружения PATH. Если параметр path не содержит явного указания каталога, поиск ведется сначала в текущем каталоге, а затем в каталогах, указанных в PATH. Если функция не содержит суффикса "p", то файл ищется только в рабочем каталоге.
e	В порождаемый процесс может быть передан аргумент env, указывающий на окружение порождаемого процесса. Если функция не содержит суффикса "e", то порождаемый процесс наследует окружение родительского процесса.

Каждая из функций **spawn...** должна передать в порождаемый процесс хотя бы один аргумент **(argO),** и по соглашению этот аргумент — копия **path.** Впрочем, передача другого значения не является ошибкой. Суммарная длина всех аргумен-

тов (не учитывая нулевых символов, но учитывая пробелы) не должна превышать 128 символов.

В функциях с суффиксом "l" аргументы перечисляются непосредственно в операторе вызова функции как указатели на строки с нулевым символом в конце. Количество аргументов не ограничено. Последним аргументом передается **NULL,** что является признаком окончания списка.

В функции с суффиксом "v" в качестве параметра передается указатель на массив произвольной длины, содержащий указатели на строки, являющиеся аргументами порождаемого процесса. Последним из указателей в массиве должен быть **NULL,** показывающий, что список аргументов завершился.

В функции с суффиксом "e" передается массив указателей **envp** на строки, определяющие переменные окружения порождаемого процесса. Эти строки обычно имеют вид

```
<имя_переменной> = <значение>
```

Если **envp = NULL,** то для функций с суффиксом "e" так же, как и для всех остальных функций, порождаемый процесс наследует окружение родительского процесса.

Файлы, открытые на момент вызова порождаемого процесса, остаются открытыми и для этого процесса. Однако в порожденный процесс не передается режим, в котором отрыты файла (текстовый или двоичный). Если режим отличается от принятого по умолчанию, то в порожденном процессе надо произвести его установку функциями.

Поиск файла **path,** загружаемого функциями **spawn...,** осуществляется следующим образом. Если в параметре **path** явно указано расширение файла или стоит точка, ищется файл такой, который задан. Если же расширение не задано, то сначала ищется файл такой, который задан. Если он не находится, к имени добавляется расширение .exe и поиск повторяется. Если файл опять не находится, к имени добавляется расширение **.com** и поиск повторяется. Функции без суффикса "p" ведут поиск файла только в текущем каталоге (если только каталог не задан явно в **path).** А функции с суффиксом "p" сначала ведут поиск в текущем каталоге, а затем — в каталогах, указанных в переменной окружения PATH.

Все функции возвращают 0 при успешной загрузке порожденного процесса, а при ошибке возвращают -1. В этом случае глобальная переменная **errno** может принимать значения **E2BIG** — слишком длинный список аргументов, **EINVAL** — ошибочный аргумент, **ENOENT** — не найден путь или файл, **ENOEXEC** — ошибка формата, **ENOMEM** — не хватает памяти.

Если в программе требуется организовать ожидание завершения порожденного процесса, используются функции **cwait и wait.**

Примеры

Операторы

```
if(spawnlp(P_WAIT,"arj","arj","e doc.arj a1.txt", NULL))
  ShowMessage("Программа arj не выполнена");
else
 {
  Memo1->Clear();
  Memo1->Lines->LoadFromFile("a1.txt");
  DeleteFile("a1.txt");
 }
```

запускают архиватор arj, извлекающий из архива *doc.arj* файл *a1.txt.* Приложение ждет, пока программа arj закончит работу, затем загружает разархивированный файл в окно редактирования **Memo1** и удаляет этот файл с диска.

В приведенном примере все аргументы, передаваемые в порождаемый процесс, объединены в одной строке. Тот же самый результат получился бы, если передать их все в отдельности:

```
if(spawnlp(P_WAIT,"arj","arj","e","doc.arj","a1.txt",NULL))
  ...
```

Операции, подобные рассмотренным выше, невозможно было бы выполнить функциями **exec...**, поскольку они не обеспечивают возвращения в исходное приложение. Нельзя было бы выполнить эти операции и функциями **spawn...** при режиме, отличном от **P_WAIT**, поскольку в этом случае оператор загрузки файла в окно редактирования выполнялся бы раньше, чем успевал распаковываться архив. Впрочем, можно было бы использовать и режим **P_NOWAIT,** но с добавлением функций **cwait** или **wait:**

```
int ID = spawnlp(P_ NOWAIT,"arj","arj", "e doc.arj a1.txt", NULL);
if (ID == -1}
  ShowMessage("Программа arj не выполнена");
else
{
  if(wait(NULL) != ID)
    ShowMessage("Ошибкаразархивации") ;
  else
  {
    Memo1->Clear();
    Memo1->Lines->LoadFromFile("a1.txt");
    DeleteFile("a1.txt");
  }
}
```

В этом коде функция **spawnlp** выполняется в режиме **P_NOWAIT**, не обеспечивающем ожидание конца порожденного процесса. Но затем вызывается функция **wait,** которая обеспечивает ожидание. Если эта функция вернет значение, отличное от идентификатора порожденного процесса, значит при выполнении порожденного процесса произошло его аварийное завершение.

Надо отметить, что приведенный выше пример разархивации файла обладает двумя недостатками. Первый из них связан с тем, что выполняется программа arj, предназначенная для DOS. Поэтому при ее выполнении вызывается сеанс DOS, и после его окончания пользователь видит окно DOS, которое ему надо закрыть, чтобы продолжить работу. Это, конечно, очень неудобно. Устранить этот недостаток легко, например, написанием пакетного файла *arj.bat* вида:

```
@echo off
arj.exe e doc %1
exit
```

В нем помимо команда разархивации предусмотрена команда **exit** — окончание сеанса работы с окном DOS. Тогда обращение к разархивации в приложении может быть даже короче, чем раньше:

```
if(spawnlp(P_WAIT,"arj.bat","arj.bat","a1.txt", NULL))
  ...
```

Обращение к пакетному файлу *arj.bat* позволяет порожденному процессу автоматически, без вмешательства пользователя вернуться в родительский процесс. Остается еще один недостаток рассмотренного примера — на время выполнения разархивации получаются неприятные изменения экрана, связанные с выходом в DOS.

Рассмотрим еще один пример использования функции **spawnlp.** Пусть вы разработали пользовательский интерфейс, в котором хотите предоставить пользователю возможность запускать различные приложения. Ваш интерфейс достаточно большой и поэтому желательно запускать из него внешние приложения в оверлэйном режиме. Эту задачу можно решить следующим образом.

Пусть имя вашего приложения *POverlay.exe.* Создайте еще одно приложение, названное, например, *OMenage.exe.* Это приложение будет управлять запуском требуемых программ. Оно может быть очень маленьким, не содержать ни одной формы и располагаться в оперативной памяти одновременно с запускаемыми программами. Весь текст его файла следующий:

```
tinclude <vcl.h>
tpragma hdrstop
tinclude <process.h>
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR IpCmdLine, int)
I
  spawnlp(P_WAIT,lpCmdLine,lpCmdLine, NULL);
  spawnlp(P_OVERLAY,"POverlay.exe","POverlay.exe", NULL);
  return 0;
}
```

Первый вызов функции **spawnlp** обеспечивает запуск в режиме ожидания того приложения, имя которого передано через командную строку **IpCmdLine.** Второй вызов **spawnlp** обеспечивает оверлэйный вызов вашего основного приложения *POverlay.exe.*

Предположим, что в вашем основном приложении *POverlay.exe* имя запускаемой программы записано в окне редактирования **Editl.** Тогда вызов этой программы может осуществляться оператором:

```
if(spawnlp(P_OVERLAY,"OMenage.exe","OMenage.exe",Edit1->Text, NULL))
ShowMessage("Программа " + Editl->Text + " не выполнена;"+
            " нет файла OMenage.exe");
```

Этот оператор прервет выполнение приложения *POverlay.exe* и загрузит на его место в памяти короткую (примерно 10 K) программу *OMenage.exe,* передав в нее как параметр имя запускаемого приложения. Программа *OMenage.exe* вызовет в режиме ожидания эту программу, а по окончании ее работы удалится из памяти и опять вызовет основное приложение *POverlay.exe.* Таким образом, во время выполнения вызываемой программы в памяти будет находиться не ваше большое приложение *POverlay.exe,* а только маленькая программа управления *OMenage.exe.*

Описанное взаимодействие программ имеет некоторый недостаток: при возврате в *POverlay.exe* текст в окне **Editl** будет утерян. Этот недостаток легко устранить. Измените основной файл приложения *POverlay* следующим образом:

```
tinclude <vcl.h>
tpragma hdrstop
USERES("POverlay.res");
USEFORM("UOverlay1.cpp", Form1);
tinclude "UOverlay1.h" // включение головного файла приложения

//---------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR IpCmdLine, int)
{
    try
    {
     Application->Initialize();
     Application->CreateForm(__classid(TForm1), &Form1);
     Form1->Editl->Text = IpCmdLine; // Загрузка окна Editl
     Application->Run();
    }
    catch (Exception &exception)
    {
     Application->ShowException(&exception);
    }
    return 0;
}
```

По сравнению со стандартным файлом, созданным C++Builder, в него добавлено две строки (отмечены комментариями): директива, включающая заголовочный файл модуля *UOverlay1.h*, содержащего описание вашей формы **Form1**, и оператор, загружающий в окно **Edit1** текст, переданный через командную строку. Еще одно изменение по сравнению, со стандартным файлом — введение в заголовок функции **WinMain** параметра **lpCmdLine** — ссылки на командную строку. Если в файле приложения *POverlay* сделаны такие изменения, то в приложении *OMenage* второй вызов функции должен быть изменен на следующий:

```
spawnlp(P_OVERLAY,"POVERLAY.exe","POVERLAY.exe",IpCmdLine, NULL);
```

Этот вызов отличается от того, что был раньше, передачей в программу той командной строки, которая была задана при вызове *OMenage*. Таким образом в программу *POverlay* вернется имя запускавшейся программы, которое будет загружено в окно **Edit1**.

sprintf — форматированный вывод в массив символов

Выводит форматированные данные в буферный массив.
См. разд. «fprintf и другие функции форматированного вывода».

srand — генерация псевдослучайных чисел

Рандомизирует последовательность псевдослучайных чисел.
См. разд. «random и другие функции генерации псевдослучайных чисел».

sscanf — форматированный ввод из буфера в памяти

Вводит форматированные данные из буфера в памяти.
См. разд. «scanf и другие функции форматированного ввода».

_status87 и другие функции получения слова состояния FPU

Возвращают текущее значение слова состояния FPU.

Заголовочный файл *float.h*.

Синтаксис

```
#include <float.h>
unsigned int _status87(void);
unsigned int _statusfp(void);
```

Описание

Функции **_status87** и **_statusfp** возвращают текущее значение слова состояния FPU (см. разд. 1.9.3). При возникновении одной из ошибок, связанных с вычислениями с плавающей запятой, она отражается в слове состояния. Если генерация исключений при этих ошибках замаскирована (см. разд. «_control87, _controlfp — доступ к управляющему слову FPU», то получение и анализ слова состояния — основной инструмент реагирования на ошибки вычислений.

Функция **_statusfp** идентична функции _status87. Она введена для совместимости с Microsoft.

Слово состояния можно также получить функциями **control87**, **controlfp** и функциями **_clear87**. **_clearfp**, но последние, возвращая слово состояния, к тому же очищают его.

Пример

```
float x;
double y = 1.5e-100;
Labell->Caption = IntToHex((int)_status87(),4);
x = y; // ошибки потери порядка и точности
Label2->Caption = IntToHex((int)_status87(),4);
```

В этом примере при выполнении присваивания переменной x значения переменной y возникают ошибки потери порядка и точности, поскольку переменная типа **double** не может хранить столь малого значения, которое присвоено переменной x. В результате в метке **Label1** отобразится начальное значение слова состояния — • "0000", а в метке **Label2** — значение "0030", биты которого свидетельствуют о появлении ошибок. Конечно, приведенный код будет нормально выполняться, если замаскирована генерация соответствующих исключений (см. разд. 1.9.3).

_statusfp — текущее значение слова состояния FPU

Возвращает текущее значение слова состояния FPU.

См. разд. «_status87 и другие функции получения слова состояния FPU».

StdDev — вычисление среднего квадратического отклонения

Возвращает среднее квадратическое отклонение элементов массива.

Заголовочный файл *Math.hpp.*

Синтаксис

```
#include <Math.hpp>
extern PACKAGE Extended__fastcall
          StdDev(const double * Data, const int Data_Size);
```

Описание

Функция **StdDev** возвращает несмещенную оценку среднего квадратического отклонения элементов массива действительных чисел **Data.** Параметр Data_Size — индекс последнего элемента массива, учитываемого при подсчете среднего значения. Если массив A содержит **n** элементов, то среднее квадратическое отклонение рассчитывается по формуле

$$\sqrt{\sum_{i=1}^{n}\left(A[i]-\overline{A}\right)^2/(n-1)},$$

где $\overline{A}$ — среднее значение (математическое ожидание) элементов массива. Это несмещенная оценка, статистически более точная, чем корень из суммы квадратов отклонений, деленной на n.

Например, оператор

```
B = StdDev(A, 99);
```

присваивает действительной переменной B значение среднего квадратического отклонения первых 100 элементов, хранящихся в массиве действительных чисел A.

Если необходимо одновременно рассчитывать математическое ожидание и среднее квадратическое отклонение, то лучше воспользоваться более быстрой функцией **MeanAndStdDev**.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

StrCopy и другие функции копирования строк

Копируют одну строку в другую.

Заголовочные файлы *SysUtils.hpp, string.h, wchar.h, mbstring.h.*

Синтаксис

```
extern PACKAGE char *__fastcall
                StrCopy(char * Dest, const char * Source);
```

```
extern PACKAGE char *____fastcall
                  StrECopy(char * Dest, const char * Source);
extern PACKAGE char *____fastcall
                  StrLCopy(char * Dest, const char * Source,
                           unsigned MaxLen);
extern PACKAGE char *____fastcall
                    StrMove(char * Dest, const char * Source,
                            unsigned Count);
char * strcpy(char *dest, const char *src);
wchar_t * wcscpy(wchar_t *dest, const wchar_t *src);
unsigned char *
      _mbscpy(unsigned char *dest, const unsigned char *src);
char * strncpy(char *dest, const char *src, size_t maxlen);
wchar_t *
   wcsncpy(wchar_t *dest, const wchar_t *src, size_t maxlen);
unsigned char *
     _mbsncpy(unsigned char *dest, const unsigned char *src,
              size_t maxlen);
```

Описание

Функции копируют строку **Source (src) в Dest (dest).** Например:

```
char buff[100];
strcpy(buff,"Текст, копируемый в buff");
```

Все функции, кроме **StrECopy,** возвращают указатель на результат копирования. Например, если в приведенном выше коде заменить второй оператор на

```
char *P = strcpy(buff,"Текст, копируемый в buff");
```

то P будет указывать на строку **buff.**

Функция **StrECopy** отличается от остальных тем, что возвращает указатель на последний символ скопированной строки. Это дает возможность вложенными вызовами склеивать несколько строк. Например, операторы

```
char *S1 = "текст 1", *S2 = "текст 2", S[20];
StrECopy(StrECopy(StrECopy(S,S1)," "),S2);
```

приведут к формированию в S строки: "текст 1 текст 2". Использование таких вложенных вызовов **StrECopy** более эффективно, чем многократный вызов **StrCat.**

Функции **StrPCopy** и **StrPLCopy** одновременно с копированием осуществляют преобразование строки типа **AnsiString** в строку с нулевым символом в конце.

Функции копирования не осуществляют проверки размера строки, в которую производится копирование. Так что надо программно проверять размеры строк функцией StrLen. Размер приемника должен быть по крайней мере на 1 больше числа символов в строке-источнике, чтобы разместить заключительный нулевой символ. Если размер приемника меньше, то нарушится распределение памяти, что может привести к непредсказуемым последствиям.

Если приемник не способен вместить всю копируемую строку, можно воспользоваться для копирования функциями **StrLCopy, StrPLCopy, StrMove, strncpy** и другими, в которых параметр **MaxLen (Count, maxlen)** указывает максимальное число копируемых символов. Если в источнике меньше символов, то копируется вся строка и лишние символы заменяются нулевыми. Но если в источнике символов больше, чем **MaxLen,** то копируются только первые **MaxLen** символов. Так что если задать значение **MaxLen** на 1 меньше размера приемника, то будет гарантия, что приемник не переполнится. Например, копирование строки **Source** в строку S можно осуществлять оператором

```
StrLCopy(S,Source, sizeof(S)-1);
```

Приемник не переполнится при любом числе символов в **Source,** хотя, конечно, длинная строка не скопируется в приемник полностью.

Функция **StrMove** отличается от других тем, что источник и приемник могут перекрывать друг друга в памяти.

Помимо перечисленных в данном разделе функций, для копирования можно применять функции **memmove, memcpy** и другие, копирующие блоки памяти, в качестве которых могут выступать и строки (см. разд. «memcpy и другие функции копирования и заполнения блоков памяти»).

strcpy — копирование строк

Копирует одну строку в другую.

См. разд. «StrCopy и другие функции копирования строк».

StrECopy — копирование строк

Копирует одну строку в другую.

См. разд. «StrCopy и другие функции копирования строк».

StrLCopy — копирование строк

Копирует одну строку в другую.

См. разд. «StrCopy и другие функции копирования строк».

StrLower — преобразование строки к нижнему регистру

Преобразует строку к нижнему регистру.

См. разд. «AnsiLowerCase и другие функции преобразования строки к нижнему регистру».

strlwr — преобразование строки к нижнему регистру

Преобразует строку к нижнему регистру.

См. разд. «AnsiLowerCase и другие функции преобразования строки к нижнему регистру».

StrMove — копирование строк

Копирует одну строку в другую.

См. разд. «StrCopy и другие функции копирования строк.

strncpy — копирование строк

Копирует одну строку в другую.

См. разд. «StrCopy и другие функции копирования строк».

StrPos — поиск подстроки

Возвращает указатель на позицию первого вхождения заданной подстроки в строку.

См. разд. «AnsiPos и другие функции поиска подстроки».

StrToCurr, StrToInt, StrToFloat и другие функции преобразования строки в число

Преобразуют строку в монетарное, целое или действительное число.

Заголовочный файл *SysUtils.hpp*.

Синтаксис

```
extern PACKAGE System::Currency __fastcall
                    StrToCurr(const AnsiString S);
```

```
extern PACKAGE System::Currency__fastcall
                 StrToCurrDef(const AnsiString S,
                              const System::Currency Default);

extern PACKAGE Extended___fastcall
                 StrToFloat (const AnsiString S) ;
extern PACKAGE Extended___fastcall
                 StrToFloatDef(const AnsiString S;
                               const Extended Default);

extern PACKAGE int___fastcall StrToInt(const AnsiString S);
extern PACKAGE int___fastcall StrToIntDef(const AnsiString S;
                                          const int Default);
extern PACKAGE bool___fastcall TryStrToInt(const AnsiString S,
                                           int &Value);
```

Описание

Функции преобразуют строку S в число: **StrToCurr** – в монетарное типа **Currency, StrToInt** – в целое, **StrToFloat** – в действительное. Если строка записана в неправильном формате, то все эти функции генерируют исключение **EConvertError.**

Функции с суффиксом "Def" при ошибочном значении формата строки возвращают число, указанное параметром **Default.**

Функция **TryStrToInt** преобразует строку S в число **Value.** Если строка записана в неправильном формате, функция возвращает **false.**

Примеры

Следующие операторы читают тексты, введенный пользователем в окна редактирования **Editl и Edit2.** Если при переводе строки генерируется исключение, то срабатывает оператор раздела **catch:**

```
try
 {
  System::Currency C = StrToCurr(Edit1->Text);
  Extended R = StrToFloat(Edit1->Text);
  int I = StrToInt(Edit2->Text);
 }
catch(EConvertError &)
 {
 // Перехват исключения при ошибочном числе
  Application->MessageBox("Неверный формат чисел",
                          "Исправьте данные",
                          MB_OK+MB_ICONSTOP);
 }
```

Оператор

```
int i = StrToIntDef(Edit1->Text,0) ;
```

присваивает переменной i значение, введенное пользователем в окно **Edit1.** Если пользователь ввел недопустимое число, значение i будет равно нулю.

StrToDate и другие функции преобразования строки в дату и время

Преобразуют строку в дату и время.

Заголовочный файл *SysUtils.hpp.*

Синтаксис

```
#include <SysUtils.hpp>
extern PACKAGE System::TDateTime___fastcall
              StrToDate(const AnsiString S);
extern PACKAGE System::TDateTime___fastcall
              StrToDateTime(const AnsiString S);
```

```
extern PACKAGE System::TDateTime___fastcall
             StrToTime(const AnsiString S);
extern PACKAGE System::TDateTime___fastcall
             StrToDateDef(const AnsiString S,
                          const System::TDateTime Default);
extern PACKAGE System::TDateTime___fastcall
             StrToDateTimeDef(const AnsiString S,
                              const System::TDateTime Default);
extern PACKAGE System::TDateTime___fastcall
             StrToTimeDef(const AnsiString S,
                          const TDateTime Default);
```

Описание

Функции **StrToDate и StrToDateDef** преобразуют строку S в дату типа **TDateTime.** Функции **StrToTime и StrToTimeDef** аналогично преобразуют строку S во время, а функции **StrToDateTime и StrToDateTimeDef** преобразуют строку в значение, содержащее и дату, и время.

Если преобразуемые строки не соответствует формату дат и времени, то функции без суффикса "Def" генерируют исключение **EConvertError,** а функции с суффиксом "Def" возвращают значение по умолчанию, заданное параметром **Default,** не генерируя исключение.

Часть преобразуемой строки, обозначающая дату, должна содержать два или три двузначных числа, разделенных символами, определенными в глобальной переменной **DateSeparator** (для русифицированных версий Windows это обычно точка "."). Последовательность чисел определяется глобальной переменной **ShortDateFormat:** или месяц/день/год — это по умолчанию, или день/месяц/год — это обычно принято в русифицированных версиях Windows, или год/месяц/день. Если заданы только 2 числа, они воспринимаются как месяц и день текущего года.

Двузначное число, обозначающее две последние цифры года, может лежать в пределах 00-99. При его преобразовании в год используется глобальная переменная **TwoDigitYearCenturyWindow.** Если **TwoDigitYearCenturyWindow** = 0, то число обозначает год текущего столетия. Например, в 1999 году числа 99 и 00 обозначали соответственно 1999 и 1900 годы, а в 2000 они обозначают 2099 и 2000 годы. Если же задать **TwoDigitYearCenturyWindow** > 0, то заданное значение вычитается из текущего года, сдвигая точку отсчета. Тогда годы, превышающие эту новую точку отсчета, относятся по-прежнему к текущему столетию, а годы, предшествующие новой точке отсчета, переносятся в следующее столетие. Например, при **TwoDigitYearCenturyWindow** = 50 и в 1999" году, и в 2000 году (точки отсчета 1949 и 1950) числа 99 и 00 обозначают соответственно 1999 и 2000 годы.

Часть преобразуемой строки, определяющая время, должна содержать 2 или 3 двузначных числа, разделенных символами, определенными в глобальной переменной **TimeSeparator** (для русифицированных версий Windows это обычно двоеточие ":"). Первое число обозначает час, второе — минуты, третье — секунды. Если задано два числа, то они воспринимаются как час и минуты, а секунды считаются равными нулю.

Примеры

Ниже приведены строки и результаты их восприятия различными функциями. Подразумевается, что текущий год — 2002. Прочерки означают, что данную строку функция воспринимает как ошибочную и генерирует исключение. Функции с суффиксом "Def" в этих случаях возвращают значения по умолчанию.

Строка	Воспринимается		
	StrToDate	StrToTime	StrToDateTime
1.5.2	01.05.2002	—	01.05.2002

Строка	Воспринимается		
	StrToDate	StrToTime	StrToDateTime
1.5	01.05.2002	—	01.05.2002
17:5	—	17:05:00	17:05:00
17:5:7	—	17:05:07	17:05:07
1.5.2 17:5	—	—	01.05.2002 17:05:00
1.5.0 17:5:7	—	—	01.05.2002 17:05:07

StrToDateDef — преобразование строки в дату

Преобразует строку в дату.

См. разд. «StrToDate и другие функции преобразования строки в дату и время».

StrToDateTime — преобразование строки в дату и время

Преобразует строку в дату и время.

См. разд. «StrToDate и другие функции преобразования строки в дату и время».

StrToDateTimeDef — преобразование строки в дату и время

Преобразует строку в дату и время.

См. разд. «StrToDate и другие функции преобразования строки в дату и время».

StrToTime — преобразование строки во время

Преобразует строку во время.

См. разд. «StrToDate и другие функции преобразования строки в дату и время».

StrToTimeDef — преобразование строки во время

Преобразует строку во время.

См. разд. «StrToDate и другие функции преобразования строки в дату и время».

StrUpper — преобразование строки к верхнему регистру

Преобразует строку к верхнему регистру.

См. разд. «AnsiUpperCase и другие функции преобразования строки к верхнему регистру».

strupr — преобразование строки к верхнему регистру

Преобразует строку к верхнему регистру.

См. разд. «AnsiUpperCase и другие функции преобразования строки к верхнему регистру».

SumInt и Sum — вычисление сумм

Возвращают сумму значений элементов массива.

Заголовочный файл *Math.hpp*.

Синтаксис

```
#include <Math.hpp>
```

```
extern PACKAGE int___fastcall
                SumInt(const int * Data, const int Data_Size);
extern PACKAGE Extended___fastcall
                Sum(const double * Data, const int Data_Size);
```

Описание

Функции **SumInt** и **Sum** возвращают сумму значений элементов массива **Data** соответственно целых или действительных чисел. Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения.

Если необходимо одновременно рассчитывать сумму и сумму квадратов значений элементов массива, то лучше воспользоваться более быстрой функцией **SumsAndSquares**.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

SumOfSquares — вычисление суммы квадратов

Возвращает сумму квадратов значений элементов массива действительных чисел.

Заголовочный файл *Math.hpp.*

Синтаксис

```
#include <Math.hpp>
extern PACKAGE Extended___fastcall
       SumOfSquares(const double * Data, const int Data_Size);
```

Описание

Функция **SumOfSquares** возвращает сумму квадратов значений элементов массива действительных чисел **Data.** Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения.

Если необходимо одновременно рассчитывать сумму и сумму квадратов значений элементов массива, то лучше воспользоваться более быстрой функцией **SumsAndSquares**.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

SumsAndSquares — вычисление суммы и суммы квадратов

Вычисляет сумму и сумму квадратов значений элементов массива.

Заголовочный файл *Math.hpp.*

Синтаксис

```
#include <Math.hpp>
extern PACKAGE void___fastcall
      SumsAndSquares(const double * Data, const int Data_Size,
                     Extended &Sum, Extended &SumOfSquares);
```

Описание

Процедура **SumsAndSquares** рассчитывает для массива **Data** одновременно сумму Sum и сумму квадратов **SumOfSquares** значений элементов. Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения. Время вычислений по процедуре **SumsAndSquares** меньше, чем при последовательном вызове функций **Sum** и **SumOfSquares.** Поэтому, если требуется знать и сумму, и сумму квадратов, то лучше использовать именно эту процедуру.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

swprintf — форматированный вывод в массив символов

Выводит форматированные данные в буферный массив.

См. разд. «fprintf и другие функции форматированного вывода».

swscanf — форматированный ввод из буфера в памяти

Вводит форматированные данные из буфера в памяти.

См. разд. «scanf и другие функции форматированного ввода».

system и другие функции выполнения команд операционной системы

Выполнение команды операционной системы.

Заголовочный файл *process.h*.

Синтаксис

```
#include <process.h>
int system(const char *command);
int _wsystem(const wchar_t *command);
```

Описание

Функции **system** и **_wsystem** выполняют команду **command** и возвращают управление в вызвавшее приложение. Команда **command** может быть командой операционной системы, командой выполнения программы DOS или пакетного (batch) файла. Программа должна быть в текущем каталоге или в одном из каталогов, перечисленных в переменной окружения PATH. Команда выполняется командным процессором DOS, что вызывает в Windows в ряде случаев неприятное изменение экрана на время выполнения команды. Функции возвращают 0 при успешном начале работы командного процессора и -1 в случае неудачи.

Примеры

```
// команда DOS dir с занесением результатов
// в текстовый файл dir.txt
system("dir >> dir.txt");

// команда DOS mkdir, создающая каталог c:\\ttt
system("mkdir c:\\ttt");

// выполнение Norton Commander
system("nc");
```

Time — текущее время

Возвращает текущее время.

См. разд. «Date и другие функции определения даты и времени».

TimeToStr — преобразование времени в строку

Преобразует время в строку.

См. разд. «DateToStr и другие функции преобразования даты и времени в строк».

_tmain — макрос функции main

Макрос, развертывающийся в ту или иную форму функции **main.**

См. разд. «main — функция».

Today — текущая дата

Возвращает текущую дату.
См. разд. «Date и другие функции определения даты и времени».

Tomorrow — завтрашняя дата

Возвращает завтрашнюю дату.
См. разд. «Date и другие функции определения даты и времени».

TotalVariance — вычисление суммы квадратов отклонений

Возвращает сумму квадратов отклонений значений элементов массива от их среднего значения.

Заголовочный файл *Math.hpp.*

Синтаксис

```
#include <Math.hpp>
extern PACKAGE Extended _fastcall
      TotalVariance(const double * Data, const int Data_Size);
```

Описание

Функция **TotalVariance** рассчитывает сумму квадратов отклонений значений элементов массива **Data** от их среднего значения:

$$\sum_{i=1}^{n}\left(A[i]-\overline{A}\right)^2.$$

Это вспомогательная величина, входящая в выражения для дисперсии и среднего квадратического отклонения, вычисляемые функциями **Variance. PopnVariance. MeanAndStdDev.**

Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения.

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

TryEncodeDate — формирование даты типа TDateTime

Формирует дату из отдельных ее составляющих.
См. разд. «EncodeDate и другие функции формирования типа TDateTime».

TryEncodeDateTime — формирование даты времени типа TDateTime

Формирует дату и время из отдельных составляющих.
См. разд. «EncodeDate и другие функции формирования типа TDateTime».

TryEncodeTime — формирование времени типа TDateTime

Формирует время из отдельных его составляющих.
См. разд. «EncodeDate и другие функции формирования типа TDateTime».

_tWinMain — функция

Главная функция приложений Windows.
См. разд. «WinMain — функция».

ungetc — возврат символа во входной поток

Возвращает символ в буфер входного потока.
См. разд. «fgetc и другие функции ввода/вывода символа».

ungetch — возврат символа во входной поток

Возвращает символ в буфер входного потока.
См. разд. «fgetc и другие функции ввода/вывода символа».

ungetwc — возврат символа во входной поток

Возвращает символ в буфер входного потока.
См. разд. «fgetc и другие функции ввода/вывода символа».

UpperCase — преобразование строки к верхнему регистру

Преобразует строку к верхнему регистру.
См. разд. «AnsiUpperCase и другие функции преобразования строки к верхнему регистру».

va_start, va_arg, va_end — макросы

Обеспечивают доступ к параметрам в функциях, в которые передается неопределенное число аргументов,

Синтаксис

```
#include <stdarg.h>
typedef void _FAR *va_list;
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Описание

Макросы **va_start, va_arg и va_end** используются в функциях, в которые передается неопределенное число аргументов (см. гл. 1, разд. 1.7.5).

Макрос **va_start** начинает работу со списком аргументов, устанавливая его указатель ap на первый передаваемый в функцию аргумент. Параметр **lastfix** — это имя последнего из обязательных аргументов функции, предшествующих списку.

Макрос **va_arg** возвращает значение очередного аргумента из списка. Параметр **type** указывает тип аргумента. Перед вызовом **va_arg** значение **ap** должно быть установлено вызовом **va_start** или предшествующим вызовом **va_arg.** Каждый вызов **va_arg** переводит указатель ap на следующий аргумент.

Макрос **va_end** завершает работу со списком, освобождая память. Он должен вызываться после того, как с помощью **va_arg** прочитан весь список аргументов. В противном случае могут быть непредсказуемые последствия.

Примеры

Пусть требуется создать функцию **average,** которая рассчитывает и отображает в метке **Labell** среднее значение передаваемых в нее целых положительных чисел. Функция принимает в качестве первого аргумента некоторое сообщение, которое должно отображаться перед результатами расчета. Список обрабатываемых чисел может быть любой длины и заканчиваться нулем. Такая функция может быть реализована следующим образом:

```
#include <stdarg.h>
...
void average(AnsiString mess,...)
{
 double A = 0;
```

```
 int i = 0,arg;
 va_list ap;
 va_start(ap, mess);
 while ((arg = va_arg(ap,int)) != 0)
 {
  i++;
  A += arg;
 }
 Form1->Label1->Caption = mess + "N = " +IntToStr(i) +
                          ", среднее = " + FloatToStr(A/i);
 va_end(ap);
}
```

Вызов функции может быть, например, таким:

```
average("Результаты экзамена: ",4,2,3,5,4,0);
```

В результате функция выдаст в метку **Label1** сообщение:

```
Результаты экзамена: N = 5, среднее = 3,6
```

Функцию **average** можно было бы организовать иначе, не вводя специальную конечную метку в список (в приведенном примере — 0), а предваряя список аргументов параметром N, указывающим размер списка:

```
void average(AnsiString mess,int N,...)
{
 double A = 0;
 va_list ap;
 va_start(ap, N);
 for(int i = 0; i < N; i++)
  A += va_arg(ap,int);
 Form1->Label1->Caption = mess + "N = " +IntToStr(N) +
                                 ", среднее = " + FloatToStr(A/N);
 va_end(ap);
}
```

Вызов функции может быть, например, таким:

```
average("Результаты экзамена: ",5,4,2,3,5,4);
```

Variance — вычисление дисперсии

Возвращает дисперсию элементов массива.

Заголовочный файл *Math.hpp*.

Синтаксис

```
#include <Math.hpp>
extern PACKAGE Extended __fastcall
          Variance(const double * Data, const int Data_Size);
```

Описание

Функция **Variance** возвращает несмещенную оценку дисперсии элементов массива действительных чисел **Data.** Параметр **Data_Size** — индекс последнего элемента массива, учитываемого при подсчете среднего значения. Если массив A содержит n элементов, то дисперсия рассчитывается по формуле

$$\sum_{i=1}^{n}\left(A[i]-\overline{A}\right)^2/(n-1),$$

где $\overline{A}$ — среднее значение (математическое ожидание) элементов массива. Это несмещенная оценка, статистически более точная, чем сумма квадратов отклонений, деленная на n, возвращаемая функцией **PopnVariance.**

Дисперсия равна квадрату среднего квадратического отклонения, вычисляемого функциями **StdDev и MeanAndStdDev.** Таким образом, выполняется соотношение **Sqr(StdDev(A)) = Variance(A).**

Пример

См. пример в разд. «random и другие функции генерации псевдослучайных чисел».

vfprintf — форматированный вывод в файл

Выводит форматированные данные в файл.
См. разд. «fprintf и другие функции форматированного вывода».

vfscanf — форматированный ввод из файла

Вводит форматированные данные из файла.
См. разд. «scanf и другие функции форматированного ввода».

vfwprintf — форматированный вывод в файл

Выводит форматированные данные в файл.
См. разд. «fprintf и другие функции форматированного вывода».

vprintf — форматированный вывод на экран

Выводит форматированные данные в выходной поток.
См. разд. «fprintf и другие функции форматированного вывода».

vscanf — форматированный ввод с клавиатуры

Вводит форматированные данные из входного потока (с клавиатуры).
См. разд. «scanf и другие функции форматированного ввода».

vsprintf — форматированный вывод в массив символов

Выводит форматированные данные в буферный массив.
См. разд. «fprintf и другие функции форматированного вывода».

vsscanf — форматированный ввод из буфера в памяти

Вводит форматированные данные из буфера в памяти.
См. разд. «scanf и другие функции форматированного ввода».

vswprintf — форматированный вывод в массив символов

Выводит форматированные данные в буферный массив.
См. разд. «fprintf и другие функции форматированного вывода».

vwprintf — форматированный вывод на экран

Выводит форматированные данные в выходной поток.
См. разд. «fprintf и другие функции форматированного вывода».

wait — ожидание завершения порожденного процесса

Ожидает завершения порожденного процесса.
См. разд. «cwait и другие функции ожидания завершения порожденного процесса».

wcscpy — копирование строк

Копирует одну строку в другую.
См. разд. «StrCopy и другие функции копирования строк».

wcsncpy — копирование строк

Копирует одну строку в другую.

См. разд. «StrCopy и другие функции копирования строк».

_wcslwr — преобразование строки к нижнему регистру

Преобразует строку к нижнему регистру.

См. разд. «AnsiLowerCase и другие функции преобразования строки к нижнему регистру».

_wcsupr — преобразование строки к верхнему регистру

Преобразует строку к верхнему регистру.

См. разд. «AnsiUpperCase и другие функции преобразования строки к верхнему регистру».

_wexec... — функции выполнения порождаемых процессов

Порождают новый процесс.

См. разд. «exec... — функции выполнения порождаемых процессов».

_wfindfirst — стандартная функция начала поиска файлов

Обеспечивает начало поиска файлов, удовлетворяющих заданному шаблону и имеющих указанные атрибуты.

См. разд. «findfirst и другие стандартные функции поиска файлов».

_wfindnext — стандартная функция продолжения поиска файлов

Обеспечивает продолжение поиска файлов, начатого функцией **_wfindfirst**.

См. разд. «findfirst и другие стандартные функции поиска файлов».

WinExec — функция API Windows

Функция API Windows, выполняет указанное приложение.

Определение

```
UINT WinExec(
 LPCSTR lpCmdLine,  // адрес командной строки
 UINT uCmdShow      // режим открытия приложения
   );
```

Описание

Функция **WinExec** позволяет выполнить указанное приложение. Параметр **IpCmdLine** является указателем на строку с нулевым символом в конце, содержащую имя выполняемого файла и, если необходимо, параметры командной строки.

Если имя указано без пути, то Windows ищет выполняемый файл в следующей последовательности:

- Каталог, из которого загружено приложение.
- Текущий каталог.
- Системный каталог Windows, возвращаемый функцией **GetSystemDirectorv.**
- Каталог Windows, возвращаемый функцией **GetWindowsDirectory**.
- Список каталогов из переменной окружения PATH.

Параметр **uCmdShow** определяет форму представления окна запускаемого приложения Windows. Он может принимать следующие значения:

SW_HIDE	Окно делается невидимым и фокус передается другому окну.
SW_MINIMIZE	Свертывает (минимизирует) указанное окно и активизирует следующее в Z-последовательности окно верхнего уровня в списке системы.
SW_MAXIMIZE	Развертывает (максимизирует) указанное окно.
SW_RESTORE	Активизирует и отображает окно. Если это окно свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции (почти то же самое, что **SW_SHOWNORMAL**).
SW_SHOW	Активизирует и отображает окно в его текущей позиции и с текущими размерами.
SW_SHOWDEFAULT	Устанавливает состояние в соответствии с флагом **SW_** в структуре **STARTUPINFO**, передаваемой в функцию **CreateProcess** программой, запускающей приложение. Приложение должно вызывать **ShowWindow** с этим флагом, чтобы задать начальное состояние своего главного окна.
SW_SHOWMAXIMIZED	Активизирует и отображает окно в развернутом виде (максимизированном).
SW_SHOWMINIMIZED	Активизирует и отображает окно в свернутом виде (в виде пиктограммы).
SW_SHOWMINNOACTIVE	Отображает окно в свернутом виде (в виде пиктограммы). Активным остается то окно, которое было активным до этого.
SW_SHOWNA	Отображает окно в его текущей позиции и с текущими размерами. Активным остается то окно, которое было активным до этого.
SW_ SHOWNOACTIVATE	Отображает окно в его последней позиции и с последними размерами. Активным остается то окно, которое было активным до этого.
SW_SHOWNORMAL	Активизирует и отображает окно. Если это окно свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции (почти то же самое, что **SW_RESTORE**).

Чаще всего используется значение **SW_RESTORE**, при котором окно запускаемого приложения активизируется и отображается на экране. Если это окно в данный момент свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции. Для приложений не Windows, для файлов PIF и т.д. состояние окна определяет само приложение.

При успешном выполнении запуска приложения функция **WinExec** возвращает значение, большее 31. Если возвращено меньшее значение, это свидетельствует об ошибке.

Достоинством функции WinExec является ее совместимость с ранними версиями Windows. Собственно для этого она и сохраняется в WIN32, хотя для Win32 рекомендуется пользоваться функцией CreateProcess.

При работе с Win32 функция **WinExec** завершает работу, если вызванное приложение вызывает функцию **GetMessage** или заканчивается выделенный лимит времени. Таким образом, ожидание можно прервать, предусмотрев в процессе, запущенном с помощью **WinExec,** в нужный момент вызов функции **GetMessage.**

Примеры

Оператор

```
WinExec("file.exe",SW_RESTORE);
```

запускает программу **file.exe.** Оператор

```
WinExec("nc",SW_RESTORE);
```

запускает Norton Commander. Оператор

```
WinExec("COMMAND.COM",SW_RESTORE);
```

приводит к запуску MS-DOS.

Операторы

```
int i = WinExec(Edit1->Text.c_str(),SW_RESTORE);
if (i < 32)
 ShowMessage("Код ошибки "+ IntToStr(i));
```

обеспечивают выполнение любой программы, имя которой пользователь набрал в окне редактирования **Edit1.** Поскольку первый параметр функции должен иметь тип **(char** *), а текст окна имеет тип **AnsiString,** то для приведения типов приходится использовать метод **c_str().**

Ниже приведен пример процедуры, обеспечивающей выполнение любой выбранной пользователем программы. Откройте новый проект и разместите на форме компонент **OpenDialog,** задав в нем фильтр

программы	*.exe;*.com;*.pif;*.dat
все файлы	*.*

Разместите на форме кнопку **Button** (назовите ее **BExec),** при щелчке на которой пользователь может выбрать в окне Открыть файл программу и выполнить ее. Обработчик события **OnClick** этой кнопки может иметь вид:

```
if (OpenDialog1->Execute())
 {
  int i = WinExec(OpenDialog1->FileName.c_str(),SW_RESTORE);
  switch (i)
   {
    case 0:ShowMessage("Не хватает памяти или ресурсов");
           break;
    case ERROR_BAD_FORMAT:
           ShowMessage("Ошибочный файл " + OpenDialog1->FileName);
           break;
    case ERROR_PATH_NOT_FOUND:
           ShowMessage("Не найден каталог " +
                       ExtractFilePath(OpenDialog1->FileName));
           break;
    case ERROR_FILE_NOT_FOUND:
           ShowMessage("Не найден файл " + OpenDialog1->FileName);
   }
 }
```

Запустите ваше приложение на выполнение и попробуйте вызывать из него различные программы Windows и MS-DOS.

WinMain — главная функция

Главная функция приложений Windows.

Модуль *winbase.*

Определения

```
int PASCAL WinMain(HINSTANCE hCurInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
int PASCAL wWinMain (HINSTANCE hCurInstance,
                     HINSTANCE hPrevInstance,
                     LPWSTR IpCmdLine, int nCmdShow)
int PASCAL _tWinMain (HINSTANCE hCurInstance,
                      HINSTANCE hPrevInstance,
                      LPTSTR IpCmdLine, int nCmdShow)
```

Описание

Функция **WinMain** размещается в головном файле приложения Windows и ей передается управление в начале выполнения приложения.

Вариант **wWinMain** является версией Unicode, в которой третий параметр — строка Unicode. Вариант **_tWinMain** — это макрос, форма развертывания которого автоматически изменяется в зависимости от типа приложения.

Параметр **hCurInstance** типа **HINSTANCE** является дескриптором данного экземпляра приложения. Дескриптор — это некий уникальный указатель, позволяющий Windows разбираться в множестве одновременно открытых окон различных приложений. Дескрипторы используются при обращении к различным функциям API Windows (API Windows — это пользовательский интерфейс Windows, содержащий множество полезных функций).

Параметр **hPrevInstance** типа HINSTANCE — это дескриптор предыдущего экземпляра вашего приложения (если пользователь выполняет одновременно несколько таких приложений).

Параметр **IpCmdLine** типа **LPSTR** является указателем на строку с нулевым символом в конце, содержащую параметры, передаваемые в программу через командную строку. Иногда такие параметры используются для переключения режимов работы программы или для задания различных опций при запуске приложения из диспетчера программ или функциями **WinExec**, **CreateProcess и др.**

Последний параметр **nCmdShow** — целое число, определяющее окно приложения. Этот параметр может в дальнейшем передаваться в функцию **ShowWindow.**

В консольных приложениях Win32 имеется два отличия от описания, приведенного выше для приложений Windows: параметр **hPrevInstance** всегда возвращает **NULL,** а параметр **IpCmdLine** указывает на строку, содержащую всю командную строку, а не только ее параметры. Впрочем, в консольных приложениях C и C++ практически всегда используется в качестве главной функции не **WinMain, а main.**

Значение, возвращаемое функцией **WinMain,** в Windows не используется. Его можно использовать только в процессе отладки.

Примеры реализации функции **WinMain** см. в разд. 1.2.2.

wmain — функция

Главная функция консольных приложений C и C++ в Unicode.

См. разд. «main — функция».

_wmemcpy — копирование блоков памяти

Копирует блок памяти.

См. разд. «memcpy и другие функции копирования и заполнения блоков памяти».

_wmemset — заполнение блока памяти

Заполняет блок памяти заданным символом.

См. разд. «memcpy и другие функции копирования и заполнения блоков памяти».

wprintf — форматированный вывод на экран

Выводит форматированные данные в выходной поток.

См. разд. «fprintf и другие функции форматированного вывода».

wscanf — форматированный ввод с клавиатуры

Вводит форматированные данные из входного потока (с клавиатуры).

См. разд. «scanf и другие функции форматированного ввода».

_wspavn... — функции выполнения порождаемых процессов

Порождают новый процесс.

См. разд. «spavn... — функции выполнения порождаемых процессов».

_wsystem — выполнение команды ОС

Выполнение команды операционной системы.

См. разд. «system и другие функции выполнения команд операционной системы».

wWinMain — главная функция

Главная функция приложений Windows.

См. разд. «WinMain — функция».

YearOf — дешифрация года

Определяет год.

См. разд. «DayOf и другие функции дешифрации дат и времени».

YearsBetween и другие функции определения разности лет

Возвращают число лет между двумя значениями даты и времени.

Заголовочный файл *DateUtils.hpp*.

Синтаксис

```
#include <DateUtils.hpp>
extern PACKAGE int__fastcall
                  YearsBetween(const System::TDateTime ANow,
                               const System::TDateTime AThen);
extern PACKAGE double__fastcall
                     YearSpan(const System::TDateTime ANow,
                               const System::TDateTime AThen);
```

Описание

Функции **YearsBetween и YearSpan** возвращают число лет между двумя значениями даты и времени ANow и **AThen** типа **TDateTime.** Функция **YearsBetween** возвращает число полных лет между двумя значениями. А функция **YearSpan** возвращает действительное число, содержащее дробную часть, отображающую неполный год.

Хотя годы (високосные и не високосные) имеют разную продолжительность, функции **YearsBetween и YearSpan** это не учитывают и исходят из усредненного значения 365.25 дней в году. Например, для дат 01.01 и 31.12 одного года функ-

ция **YearsBetween** выдаст число полных лет 1 для високосного года, и 0 для не високосного.

Примеры

Операторы

```
TDateTime T1 = EncodeDateTime(2001, 01, 1, 00, 00, 00, 300);
TDateTime T2 = EncodeDateTime(2002, 01, 1, 00, 00, 00, 300);
int i = YearsBetween (T2, T1);
double r = YearSpan(T2, T1);
```

зададут переменной i значение 1, а переменной r значение 1,00205338809035. В этом примере значения дат и времени T1 и T2 задаются с помощью функции **EncodeDateTime** и их разность ровно 1 год. Но из-за принятого округления функция **YearSpan** возвращает число, большее единицы, поскольку 2000 год високосный. Если выполнить аналогичные операторы для 2001 и 2002 годов, то функция **YearsBetween** вернет 0, а **YearSpan** — 0,999315537303217.

YearSpan — разность лет

Возвращает число лет между двумя значениями даты и времени.

См. разд. «YearsBetween и другие функции определения разности лет».

Yesterday — вчерашняя дата

Возвращает вчерашнюю дату.

См. разд. «Date и другие функции определения даты и времени».

Глава 5

Обзор стандартной библиотеки шаблонов STL

5.1 Стоит ли знакомиться с STL?

Стандартная библиотека шаблонов STL является мощным инструментом, способным оказать вам большую помощь в разработке сложных программ на C++. Она включает в себя множество шаблонов контейнеров, способных хранить упорядоченные данные различных типов, итераторов, дающих доступ к этим данным, алгоритмов, работающих с данными, и многое другое. При начальном знакомстве с STL может возникнуть законный вопрос: стоит ли тратить немалое время на попытки разобраться в хитросплетениях этой библиотеки? Ведь каждому программисту известно, что часто проще самому написать программу, чем разбираться в программе, написанной кем-то другим. Особенно, если эта программа не слишком хорошо документирована.

У меня самого в свое время возник этот вопрос. Но, затратив определенные усилия на знакомство с STL, я понял, что не зря потерял время. Прелесть стандартной библиотеки заключается, прежде всего, конечно, в готовых алгоритмах, которые вы можете использовать в своих задачах, вместо того, чтобы заново открывать Америку, создавая с нуля подобные алгоритмы. Но, может быть, еще важнее то, что STL предоставляет вам инструментарий для разработки собственных алгоритмов, беря на себя с помощью множества описанных в ней объектов и утилит всю черновую работу. Так что, несомненно, изучать STL стоит. Вопрос только в том, насколько доскональным должно быть это изучение. Если вы не собираетесь дополнять эту библиотеку собственными сложными шаблонами, необходимыми для решения ваших задач, то достаточно знакомства с STL на уровне пользователя. То есть необходимо знать функциональные возможности, параметры и основные функции-элементы и данные-элементы объектов библиотеки. Тонкости реализации шаблонов библиотеки при этом вас не должны интересовать. Да и сами коды шаблонов можно вообще не смотреть. И только если вы намерены своими силами создавать нечто, подобное STL, тогда детальное знакомство с кодами будет для вас прекрасной школой обретения высшей квалификации в C++.

Рассмотрение STL в этой главе ориентировано именно на уровень пользователей, а не разработчиков библиотек. Правда, описание даже на этом уровне всех возможностей STL нереально из-за ограничения на объем книги. Я очень надеюсь, что в недалеком будущем смогу написать отдельную небольшую книгу для пользователей STL. А пока в данной главе ограничиваюсь только поневоле кратким обзором основных элементов этой библиотеки.

5.2 Использование STL в C++Builder

В C++Builder 6 включена новая реализация STL — STLport 4.5. Одновременно, для обратной совместимости оставлена реализация библиотеки STL Rogue Wave разработанная по стандартам 1998 года, выпущенным American National Standards Institute (ANSI) и International Standards Organization (ISO). Но по умолчанию используется STLport. Если вы хотите использовать Rogue Wave, вам надо определить директивой #define макрос **_USE_OLD_RW_STL**.

Помимо основной рабочей версии STLport, в C++Builder имеется отладочная версия библиотеки. Она заметно снижает производительность приложения, так что использовать ее имеет смысл только при отладке. Но зато она предусматривает множество проверок, которые помогут вам легче находить различные ошибки в процессе отладки. Для использования отладочной версии вам надо определить директивой **#define** макрос _STLP_DEBUG.

При работе с STL, помимо подключения необходимых заголовочных файлов, необходимо принимать меры, чтобы идентификаторы классов, функций и т.п. соответствовали пространству имен стандартной библиотеки (см. разд. 1.8.2). Область видимости STL названа **std.** Так что наиболее простой способ использования идентификаторов STL — включить в ваш файл оператор

```
using namespace std;
```

Подобный оператор делает доступными все идентификаторы STL. Но тогда, если вам необходимо использовать в программе одноименные идентификаторы из других файлов, надо будет указывать для них область видимости явным образом.

Если вам требуется использовать только немногие идентификаторы из STL, можно явным образом задать область видимости именно для них. Например, оператор

```
using std::swap;
```

сделает доступной в любом месте вашего файла функцию **swap** из STL.

Третий вариант — указывать область видимости **std** непосредственно в момент использования соответствующего идентификатора. Например,

```
std::swap(x, y);
```

5.3 Основные концепции STL

STL реализована как библиотека шаблонов. Основы построения и использования шаблонов вы можете посмотреть в разд. 2.14.8 и 1.7.8. Применение шаблонов позволяет существенно сократить размер библиотеки и повышает ее универсальность. Действительно, без технологии шаблонов, введенной в C++, разработчикам библиотеки пришлось бы многократно повторять одни и те же классы и алгоритмы для различных типов данных. И универсальность все равно не была бы достигнута, так как невозможно предвидеть, какие типы данных, структуры и т.п. потребуются пользователям в их конкретных задачах.

Правда, система шаблонов в STL достаточно сложна, и попытка проследить в ней все связи приведет вас к серьезным затратам сил и времени. Эта система может вызвать шок, особенно у программистов, привыкших работать с другими языками программирования, где шаблонов нет. Но большинству пользователей вовсе и не надо разбираться в этих премудростях. Ведь не разбираетесь вы, как реализованы стандартные функции, имеющиеся в любом языке, и, как правило, не задумываетесь, как рассчитывается в них sin или cos. Вы просто вызываете стандартные функции, передаете в них нужные аргументы и получаете результат. Так же можно работать и с шаблонами. Надо только привыкнуть, что кроме обычных аргументов в шаблоны надо передавать, заключая в угловые скобки, типы данных, необходимые для работы. А погружение в хитросплетения STL отложите до того момента, когда решите дополнить стандартную библиотеку собственными шаблонами, типами и алгоритмами.

Итак, что же содержит STL, что реализовано в ней в виде шаблонов? Прежде всего, *контейнеры.* Контейнеры — это объекты, которые могут хранить множество *элементов* — объектов некоторого типа. Поэтому при создании контейнера всегда надо указывать для него тип хранимых данных. Но контейнеры не просто хранят данные. Элементы в контейнерах упорядочены. Контейнеры могут содержать

простые последовательности, как в массивах или строках, могут содержать связные списки: очереди, стеки. Есть контейнеры, способные хранить иерархические конструкции типа деревьев. Так что библиотечные контейнеры избавят вас от необходимости программировать подобные структуры данных. И работа со строками становится много проще, чем при использовании стандартного типа **char *.** Впрочем, никто не мешает вам создавать собственные шаблоны контейнеров по образу и подобию библиотечных, содержащие экзотические виды связей между элементами, необходимые в вашей работе. В библиотеке имеется немало шаблонов, которые помогут вам в этой работе.

Для управления памятью контейнеры используют стандартный *распределитель памяти* — шаблон **allocator,** описанный в файле <memory>. Распределитель применяет операции **new** и **delete** (см. разд. 1.11) для динамического выделения и освобождения памяти. Так что контейнеры совместно с распределителями памяти освобождают программиста от ранее неизбежной, требующей особого внимания и достаточно неприятной обязанности организовывать выделение памяти под объекты с изменяющимися размерами.

Распределитель по умолчанию **alloc** обеспечивает хорошие характеристики, так что лучше всего, обычно, использовать именно его. Из других распределителей можно указать еще **pthread_alloc,** который применяется в тех многопоточных приложениях, в которых для каждого потока выделена своя область памяти. Распределитель **pthread_alloc** можно использовать только в многопоточных операционных системах. Он может работать быстрее, чем **alloc,** особенно в многопроцессорных системах. Но зато может приводить к фрагментации памяти, поскольку память, выделенная под один поток, не может использоваться другими потоками.

Вы можете также спроектировать собственный распределитель памяти. Впрочем, для большинства приложений вполне достаточно тех операций с памятью, которые выполняются распределителем по умолчанию. Так что забудьте о проблеме памяти, библиотечные шаблоны решат ее за вас.

Доступ к элементам, размещенным в контейнерах, обеспечивают *итераторы* различных типов. Они позволяют читать значения элементов, изменять их, вставлять в контейнер новые элементы. Итераторы избавляют вас от необходимости знать что-либо о физическом размещении элементов в контейнере и вообще об особенностях реализации контейнера. Все основные операции с контейнерами осуществляются с помощью итераторов, и, разрабатывая какой-либо алгоритм, вы явно или неявно будете иметь дело только с итераторами.

Каждый контейнер имеет набор функций-элементов, обеспечивающих те операции с данными, которые чаще всего требуются для работы с контейнером данного типа. Вы можете вызывать их при выполнении своих алгоритмов обработки данных. Помимо этого, библиотека содержит много глобальных функций, обеспечивающих традиционные процедуры работы с наборами данных различных типов.

И, наконец, библиотека содержит ряд *алгоритмов* — достаточно сложных процедур обработки данных. Поэтому прежде, чем программировать необходимый алгоритм, посмотрите, нет ли его уже в стандартной библиотеке. Если есть и его работа вас устраивает, то не стоит открывать Америку и тратить свое драгоценное время на то, что уже сделали для вас разработчики стандартной библиотеки.

На этом закончим обзор того, что имеется в STL, и рассмотрим основные категории элементов этой библиотеки.

5.4 Контейнеры

5.4.1 Общие сведения

В STL имеются следующие виды контейнеров: string - - строки, **vector, valarray** и bitset — векторы (массивы), **list** — связный список, **queue и deque** – очереди, **stack** — стек, set и **multiset** — множества, **тар** и **multimap** — ассоциативные массивы, **hash_set, hash_multiset, hash_map, hash_multimap** — контейнеры, аналогичные перечисленным ранее, но использующие хэш-функции (функции быстрого поиска).

Строго говоря, не все перечисленные типы являются полноценными контейнерами. Базовыми, полноценными контейнерами являются **vector, list, deque, map.** Некоторые виды контейнеров являются адаптерами. *Адаптер* — это контейнер, построенный на основе базового, но предоставляющий ограниченный интерфейс. Например, **queue и stack** — адаптеры контейнера **deque.** Ряд перечисленных контейнеров: **string, valarray, bitset** являются «почти контейнерами». Они обладают основными чертами контейнеров, но в них реализовано не все то, что присуще контейнерам. Цель подобной неполной реализации — оптимизация выполнения основных операций. Следует отметить также, что в качестве некоторого подобия контейнера, правда, очень несовершенного, может выступать и обычный одномерный массив. Этот вопрос рассмотрен *в* разд. 5.4.3.

Впрочем, для пользователя все эти тонкости классификации значения не имеют. Ему надо знать типы, свойства и методы, доступные в контейнерах. Основные из этих элементов, присущие всем контейнерам, мы и рассмотрим в данном разделе, чтобы избежать повторений при описании конкретных типов контейнеров.

Основные типы, объявляемые как данные-элементы контейнера некоторого класса X:

Тип	Обозначение	Описание
Тип элемента	X::value_type	Тип элементов контейнера
Тип итератора	X::iterator	Тип итератора чтения и записи
Тип итератора	X::const_iterator	Тип итератора чтения
Тип ссылки	X::reference	Тип ссылки на элемент контейнера
Тип ссылки	X::const_reference	Тип константной ссылки на элемент контейнера
Тип указателя	X::pointer	Тип указателя на элемент контейнера
Тип расстояния	X::difference_type	Целый тип со знаком, представляющий расстояние между двумя итераторами
Тип размера	X::size_type	Целый тип без знака, используемый в контейнере для индексации элементов
Тип итератора	X::reverse_iterator	Тип инверсного итератора чтения и записи, позволяющего перемещаться в инверсном направлении
Тип итератора	X::const_reverse_iterator	Тип инверсного итератора чтения, позволяющего перемещаться в инверсном направлении
Тип ключа	X::key_type	Тип ключа ассоциированного контейнера

Из перечисленных типов чаще всего, пожалуй, используется **X::iterator.** Он указывается как тип итератора (см. подробнее в разд. 5.5), создаваемого вами для работы с элементами контейнера. Например, следующий оператор создает итератор **I1** для работы с векторами целых чисел:

```
vector<int>::iterator I1;
```

Остальные типы могут понадобиться только при обращении к каким-то шаблонам, требующим указание соответствующего типа.

Для контейнеров определены следующие основные функции-элементы:

Функция	Синтаксис / Описание
begin	iterator begin() **const_iterator** begin() const Возвращают итератор, указывающий на первый элемент в контейнере
empty	**bool empty()** const Возвращает true (ненулевое значение), если контейнер пуст
end	iterator end () const_iterator end () const Возвращают итератор, указывающий на позицию после последнего элемента в контейнере
max_size	**size_type** max_size() const Возвращает максимальное число элементов, которое может хранить контейнер
size	**size_type** size() const Возвращает число элементов в контейнере
swap	void **swap(X&** x) Обмен элементами с контейнером x того же типа

Функция **begin()** возвращает итератор, указывающий на первый элемент последовательности, хранящейся в контейнере. А функция **end()** указывает позицию, размещенную после последнего элемента. То есть последний элемент расположен в позиции, предшествующей **end().Таким** образом, интервал, в котором расположены элементы: [begin(), end()). Перемещаться по элементам можно только в пределах этого интервала.

Разность итераторов **end() — begin()** равна числу элементов последовательности. Эта же величина возвращается функцией **size().** А функция **max_size()** показывает максимальное число элементов, которое можно разместить в контейнере. Функция **empty()** позволяет проверить, не пуст ли контейнер. Это то же самое, что проверить, не равняется ли нулю **size().**

Функция-элемент **swap(b)** позволяет обменяться всеми элементами данного контейнера с элементами другого контейнера b того же типа. Надо сказать, что имеется также глобальная функция обмена **swap,** применимая к двум объектам любого (но одинакового) типа, доступного для присваивания. Она имеет синтаксис:

```
#include <algorithms>
void swap(Assignable& a, Assignables b);
```

Функция осуществляет обмен значений элементов двух объектов a и **b.** Реализация функции сводится к трем операциям: создается временная копия объекта a, затем в a заносится копия объекта **b,** и затем в b заносится сохраненная временная копия a.

Например, оператор

```
swap(a, b);
```

приводит к обмену данными между объектами a и b одного типа.

Таким образом, если a и b — контейнеры одного типа, то операторы

```
a.swap(b);
b.swap(a);
swap(a, b);
```

приводят к одному и тому же результату. Различие в том, что время выполнения глобальной функции **swap** линейно зависит от размера контейнера, а время выполнения функций-элементов **swap** для ряда контейнеров не зависит от размера контейнера. Так что в общем случае функция-элемент эффективнее.

Помимо перечисленных функций-элементов, каждый контейнер имеет, конечно, конструктор, с помощью которого создается контейнер. Формы конструкторов для различных контейнеров будут рассмотрены в последующих разделах. Но кроме этих конструкторов контейнеры имеют конструкторы копий (см. разд. 2.14.5). Они неявно вызываются компилятором в необходимых случаях. Например, если a — контейнер типа X, то оператор

```
X b(a);
```

объявляет переменную b класса X и вызывает конструктор копии, который заносит в b копию контейнера a. То же самое можно сделать оператором, использующим операцию присваивания,

```
X b = a;
```

или

```
X b = X(a);
```

А оператор присваивания

```
b = a;
```

копирует в уже объявленный контейнер b все элементы контейнера a.

5.4.2 Контейнеры последовательностей

Последовательность — это упорядоченный линейный набор элементов, в котором допускается вставка новых элементов в заданную позицию и удаление существующих элементов. Так что контейнеры последовательностей — это контейнеры, размер которых может изменяться в зависимости от операций вставки и удаления.

К контейнерам последовательностей относятся векторы, связные списки и очереди. Эти контейнеры имеют все функции, описанные в разд. 5.4.1, но дополнительно имеют еще функции-элементы, перечисленные ниже. В их описании T — тип элементов контейнера, a — объект контейнера.

Функция	Синтаксис / Описание
assign	**void assign(InputIterator first, InputIterator last)** Удаляет все элементы, хранившиеся в контейнере, а потом вставляет копии элементов, лежащих в интервале **[first, last]** **void assign(size_type n, const T& t)** Удаляет все элементы, хранившиеся в контейнере, а потом вставляет n элементов со значением t

Функция	Синтаксис / Описание
back	reference back() const_reference back() const Возвращает ссылку на последний элемент последовательности. Эквивалент *(a.end() - 1)
clear	void clear() Удаляет все элементы контейнера
erase	iterator erase(iterator position) Удаляет элемент, на который указывает position. Возвращает итератор, указывающий на элемент, следующий за удаленным, или end(), если удалялся последний элемент iterator erase(iterator first, iterator last) Удаляет элементы, в интервале [first, last), т.е. начиная с first и исключая last. Возвращает итератор, указывающий на элемент, следующий за удаленными, или end(), если удалялись последние элементы
front	reference front() const_reference front() const Возвращают ссылку на первый элемент. Эквивалент *(a.first())
insert	iterator insert(iterator position, const T& x) Вставляет копию значения x в позицию, указанную итератором position. Возвращает итератор, указывающий на вставленный элемент void insert(iterator position, size_type n, const T& x) Вставляет n копий значения x в позицию, указанную итератором position void insert(iterator position, InputIterator first, InputIterator last) Вставляет копии элементов, лежащих в интервале [first, last], в позицию, указанную итератором position
pop_back	void pop_back() Удаляет последний элемент последовательности. Эквивалент a.erase(a.end() - 1)
push_back	void push_back(const T& x) Вставляет копию x в конец последовательности — после последнего элемента. Эквивалент a.insert(a.end(), x)
rbegin	reverse_iterator rbegin() const_reverse_iterator rbegin() const Возвращают итератор, указывающий на первый элемент для обратного итератора, т.е. на последний элемент контейнера (на него же указывает итератор a.end() - 1)

Функция	Синтаксис / Описание
rend	**reverse_iterator rend()** **const_reverse_iterator** rend() const Возвращают итератор, указывающий на конец последовательности для обратного итератора, т.е. на позицию перед первым элементом контейнера (на такую же позицию указывает итератор a.begin() - 1)
resize	void resize(size_type sz) Изменяет число элементов до sz void resize(size_type sz, T c) Изменяет число элементов до sz, задавая новым элементам значение c

Большинство перечисленных функций изменяет число элементов в контейнере, т.е. изменяет size. Функции clear и erase уменьшают size. Первая форма erase удаляет один элемент, а вторая удаляет сразу группу следующих друг за другом элементов. В обоих случаях последовательность смыкается, т.е. последующие элементы перемещаются на место удаленных. Пусть, например, имеется контейнер V, содержащий последовательность целых чисел: (1, 2, 3, 4, 5, 6, 7, 8). И пусть имеется итератор **I1**, которому присвоено значение begin() + 1:

```
I1 = V.begin() + 1;
```

Такой итератор указывает на второй элемент. Тогда оператор:

```
V.erase(I1);
```

удалит второй элемент и последовательность станет равной (1, 3, 4, 5, 6, 7, 8).

Поскольку функция erase возвращает итератор, указывающий на позицию элемента, следующего за удаленным, то вызов erase в цикле удалит несколько следующих друг за другом элементов. Например, при том же значении **I1** цикл:

```
for(int i=1; i <= 5; i++)
 V.erase(I1);
```

удалит элементы со второго по шестой и последовательность станет равной: (1, 7, 8). Аналогичного результата можно достичь (причем, с меньшими затратами времени), если создать для контейнера еще один итератор **I2**, задать ему значение

```
I2 = V.begin() + 6;
```

и выполнить оператор

```
V.erase(I1, I2);
```

Функция insert вставляет новые элементы с заданным значением x в позицию, на которую указывает итератор. Последующие элементы сдвигаются, освобождая место для новых элементов. Возвращается итератор, указывающий на новый элемент, т.е. тот же итератор, который указывал позицию вставки. Например, при прежних предположениях о значении итератора **I1** оператор

```
V.insert(I1, 9);
```

вставит значение 9 во вторую позицию, так что последовательность примет вид: (1, 9, 2, 3, 4, 5, 6, 7, 8). Оператор

```
V.insert(I1, 3, 9);
```

вставит 3 элемента со значениями 9: (1, 9, 9, 9, 2, 3, 4, 5, 6, 7, 8).

Третья форма оператора insert позволяет копировать в новые элементы значения из какого-то другого контейнера или из того же самого контейнера. Например, для того же контейнера V оператор

```
V.insert(I1, V.end()-2, V.end());
```

скопирует два последних элемента во вторую позицию: (1, 7, 8, 2, 3, 4, 5, 6, 7, 8). В подобном операторе итераторы могут указывать диапазон в другом контейнер, и тогда значения будут копироваться именно оттуда.

Две формы функции присваивания **assign** отличаются от второй и третьей форм функции **insert** тем, что предварительно очищает контейнер, удаляя все хранившиеся в нем элементы, а потом вставляют новые элементы, начиная с первой позиции. Так что если выполнить оператор

```
V.assign(3, 9);
```

в контейнере останется последовательность из трех чисел: (9, 9, 9);

Функция **resize** изменяет число элементов, хранящихся в контейнере до sz. Если sz > **size(),** то в конец последовательности вставляется sz - **size()** новых элементов. В первой форме функции значения новых элементов будут равны значениям по умолчанию, установленным для данного типа элементов. Во второй форме функции значения новых элементов равны с. Например, для того же контейнера V оператор

```
V.resize(10);
```

даст последовательность: (1, 2, 3, 4, 5, 6, 7, 8, О, О). А оператор

```
V.resize(10,9);
```

даст результат: (1, 2, 3, 4, 5, 6, 7, 8, 9, 9).

Если в функции **resize** значение sz < **size(),** то лишние элементы будут удалены из последовательности.

Помимо рассмотренных функций-элементов, контейнеры последовательностей имеют ряд конструкторов:

Синтаксис конструктора класса T / Описание
explicit T(const Allocator& alloc = Allocator()) Конструктор по умолчанию. Обычно можно использовать стандартный распределитель памяти (см. разд. 5.3), так что вызов конструктора часто ограничивается просто указанием типа контейнера
explicit T(size_type n) Создает контейнер на n элементов, задавая им значения, принятые в типе элементов по умолчанию
T(size_type n, const T& value, const Allocator& alloc = Allocator()) Создает контейнер на n элементов, задавая им значения **value.** Распределитель памяти обычно можно не указывать
T(InputIterator first, InputIterator last, const Allocator& alloc = Allocator()) Создает контейнер с числом элементов **last – first.** Элементы копируются из того контейнера, на который указывают итераторы **first** и **last,** в интервале [first, last), т.е. исключая **last.** Распределитель памяти обычно можно не указывать

Отметим, что последняя форма конструктора позволяет создавать контейнер, содержащий копию части другого контейнера того же типа.

5.4.3 Векторы

Вектор — это последовательность элементов, каждый из которых занимает в последовательности ту позицию, которая ему назначена при записи, или которую он приобретает в результате вставки и удаления каких-то элементов. Позиция элемента в последовательности никак не связана со значением элемента. Число элементов вектора может изменяться во время выполнения. Доступ к элементам

произвольный (см. разд. 5.5.4). Удалять и вставлять элементы можно в начале, в середине, или в конце последовательности. Аналогом вектора является одномерный массив с изменяющимся размером.

Базовый шаблон вектора:

```
template <class T, class Allocator = allocator<T> >
class vector (
...
)
```

подключается к проекту заголовочным файлом *vector.* Класс T — это класс элементов вектора. Распределитель памяти **Allocator,** как было сказано в разд. 5.3, можно обычно не задавать, используя распределитель по умолчанию.

Вектор имеет все типы конструктора, описанные в разд. 5.4.2, и все функции-элементы, описанные в разд. 5.4.1 и 5.4.2. Например, создание пустого вектора может быть выполнено операторами:

```
#include <vector>
using namespace std;
...
vector<int> V;
```

Они создают вектор целых чисел **int** *с* длиной **size = 0.** Следующий оператор вставляет в созданный вектор 5 элементов, задавая всем им значение 1.

```
V.insert(V.begin(), 5, 1);
```

То же самое можно сделать следующим оператором:

```
V.resize(5,1);
```

или следующим:

```
V.assign(5, 1);
```

Ну, а проще всего воспользоваться соответствующим конструктором и сразу создать вектор с пятью элементами, равными 1:

```
vector<int> V(5, 1);
```

Операторы

```
vector<int> V;
for (int i = 1; i <= 10; i++)
  V.push_back(i);
```

создают вектор целых чисел и заполняют его элементами от 1 до 10.

Все использованные в приведенных примерах конструкторы и функции-элементы описаны в предыдущих разделах. А ниже приведена таблица тех функций-элементов, которые в предыдущих разделах не рассматривались.

Функция	Синтаксис / Описание
at	reference at(size_type n) **const_reference at(size_type)** const Возвращают ссылку на элемент с индексом n. Индексы начинаются с нуля, так что значение n должно быть в пределах [0, size - 1]
capacity	size_type capacity() const Возвращает емкость контейнера — число элементов, которые можно разместить в выделенной под вектор памяти
reserve	void **reserve(size_type** n) Увеличивает емкость контейнера capacity, не добавляя в вектор новых элементов

Функция **at** возвращает ссылку на элемент вектора с индексом n. Индексы начинаются с 0 и заканчиваются значением **size()** - 1. Так что выражение

```
V.at(0)
```

возвращает значение первого элемента, а выражение

```
V.at(V.size() - 1))
```

возвращает значение последнего элемента.

Если значение индекса n лежит вне допустимых пределов, генерируется исключение **out_of_range.** Этим можно воспользоваться для проверки, имеет ли некоторый итератор, с которым вы хотите работать, правильное значение. Пусть, например, **I1** — итератор, в значении которого вы не до конца уверены. Тогда применение этого итератора имеет смысл оформить следующим образом:

```
try
{
  V.at(I1 - V.begin());
  <операторы, использующие I1>
}
catch(out_of_range)
{
 ShowMessage("Ошибка диапазона");
}
```

Теперь рассмотрим функцию **capacity.** Емкость, которую возвращает **capacity,** — это число элементов, под которые в данный момент выделена память. Значение **capacity()** всегда не меньше, чем size(), но обычно больше. Т.е. память выделяется под число элементов, большее, чем пока имеется в векторе. Делается это для того, чтобы предотвратить потери времени на перераспределение памяти при добавлении каждого нового элемента. Если число добавляемых элементов меньше, чем **capacity()** — size(), перераспределения памяти не происходит и добавление производится быстро. Если при добавлении число элементов превышает **capacity(),** память перераспределяется, и емкость вектора скачком увеличивается, обеспечивая запас для добавления последующих элементов. В стандарте не оговорен алгоритм изменения емкости. В версии STL, используемой в C++Builder 6, емкость при перераспределении удваивается.

Перераспределение памяти неприятно не только из-за затрат времени. При перераспределении теряются значения всех итераторов, которые вы создали для вектора (подробнее об итераторах см. в разд. 5.5). Это заставляет вас усложнять алгоритмы работы, если они используют итераторы и добавляют в вектор новые элементы. Контролировать момент перераспределения памяти можно по изменению значения, возвращаемого **capacity().** Например, в начале работы вашего алгоритма вы можете записать оператор

```
int old_capacity = V.capacity();
```

А затем, после вставки очередных элементов, организовать такую проверку:

```
if (old_capacity != V.capacity())
{
  <операторы восстановления итераторов>
  old_capacity = V.capacity();
}
```

Емкость **capacity** вы не можете задавать программно. Но емкостью можно управлять с помощью функции **reserve,** которая позволяет перераспределить память заранее, если вы знаете, сколько элементов будет добавляться. Тогда вы задаете в качестве аргумента этой функции число, обеспечивающее выделение памяти под все ожидаемые элементы. Например, если вы ожидаете, что вектор в итоге будет включать в себя не больше 1000 элементов, вы можете после создания вектора написать оператор

```
V.reserve(500);
```

Поскольку емкость вектора в C++Builder 6 устанавливается с двойным запасом, то **capacity** станет равна 1000, и вы не будете знать проблем с перераспределением памяти.

Если вы зададите значение аргумента в **reserve,** превышающее допустимую величину **max_size(),** будет сгенерировано исключение **length_error,** которое вы можете перехватить и как-то прореагировать на невозможность работать с таким большим вектором.

Значения итераторов теряются при перераспределении памяти функцией reserve или при описанном выше автоматическом увеличении емкости. В документации утверждается, что при вставке и удалении элементов где-то в середине вектора должны теряться значения итераторов, указывающих на позиции, расположенные после позиции вставки и удаления. Правда, эксперименты показывают, что в C++Builder 6 этого не происходит. При любых вставках и удалениях, не приводящих к перераспределению памяти, значения все итераторов сохраняются. Но, поскольку это не оговорено в стандарте, может быть все-таки лучше прислушаться к совету, данному в документации, и по возможности заранее выделять память функцией **reserve, а** все вставки и удаления делать в конечной позиции вектора.

К элементам вектора можно получить доступ не только с помощью итераторов. Для вектора определена операция индексации **[n],** дающая доступ к элементу с индексом n. Индексы отсчитываются от 0, так что максимальный индекс на 1 меньше size(). Например, следующий код обеспечивает поочередное отображение всех элементов вектора

```
for (int i=0; i<V.size()-1; i++)
   ShowMessage(V[i]);
```

Благодаря операции индексации, с вектором можно работать как с обычным массивом, но допускающим изменение размера и обладающим множеством полезных методов, описанных выше. В разд. 5.5.1 показано, что и с обычными массивами можно работать с помощью итераторов, и что к ним применим ряд функций, предназначенных для работы с контейнерами.

Конструктор вектора позволяет также создать копию обычного массива в виде вектора. Например:

```
int v[10] = {1,2,3,4,5,6,7,8,9,10};
vector<int> V(v,v+10);
```

Но имейте в виду, что вектор V — это копия массива v, в которую загружены значения элементов массива на момент создания копии. Дальнейшие изменения значений элементов вектора и массива совершенно не влияют друг на друга.

Мы рассмотрели многие особенности базового класса векторов **vector.** В последующих разделах вы найдете еще немало примеров работы с этим классом. А в данном разделе рассмотрим коротко некоторые другие классы векторов.

Тип элементов, для которого строится вектор, может быть самым различным: целые или действительные числа, структуры и т.п. За такую универсальность приходится расплачиваться не самой высокой производительностью при решении конкретных задач. Поэтому в библиотеке имеется несколько вариантов векторов, в которых, за счет сужения множества допустимых типов элементов, достигается более высокая производительность и вводятся новые, специализированные функции. К сожалению, из-за ограничения на объем книги нет никакой возможности подробно рассмотреть эти классы векторов. Поэтому ограничусь кратким обзором, надеясь, что смогу вернуться к этим и другим классам библиотеки в небольшой отдельной книге.

Прежде всего, надо сказать о варианте вектора с элементами булева типа **bool — vector<bool>.** Он реализован как специальный случай вектора и отличается, прежде всего, тем, что каждый элемент хранится в отдельном бите, в то время,

как в остальных векторах для хранения элементов используется, по крайней мере, один байт. Весь интерфейс векторов сохраняется и для **vector<bool>.** Только операции выполняются более эффективно. Более того, в **vector<bool>** добавлены две новые функции:

Функция	Синтаксис / Описание
flip	**void flip()** **Инвертирует значения всех элементов: true изменяется на false и наоборот**
swap	**void swap(reference x, reference y)** **Обмен элементами x и y**

Функция **flip** изменяет значения всех элементов вектора на противоположные: те, которые имели значения **true,** становятся равными **false** и наоборот.

Например, следующий код создает, заполняет и отображает вектор булевых значений:

```
vector<bool> V(4);
V[0] = true;
V[1] = false;
V[2] = false;
V[3] = true;

for (int i=0; i < V.size(); i++)
  V[i] ? ShowMessage("true") : ShowMessage("false");
```

Созданный вектор содержит элементы: **(true, false, false, true).** Но если вы после создания вектора выполните оператор

```
V.flip();
```

то значения элементов будут равны: **(false, true, true, false).**

Введенная для **vector<bool>** специальная форма функции **swap,** рассмотренной в разд. 5.4.1, связана с хранением элементов в битах, а не байтах.

Не все компиляторы поддерживают особые возможности **vector<bool>.** Поэтому в библиотеке временно оставлен для обратной совместимости класс **bit_vector,** который реализует вектор булевых значений и не отличается по интерфейсу от **vector<bool>.**

Еще большее удобство при работе с векторами битовых величин дает класс **bitset.** В шаблон **bitset<N>,** объявленный в заголовочном файле *bitset,* передается параметр N, указывающий размер вектора. В дальнейшем этот размер не может изменяться, т.е. вектор **bitset** имеет постоянную длину. Отмечу только, что в этом классе введены поразрядные логические операции (см. разд. 1.9.7), которые облегчают работу с битовыми величинами.

Теперь рассмотрим коротко вектор **valarray,** объявленный в заголовочном файле *valarray.* Честно говоря, именно этот шаблон, а не **vector,** является настоящим вектором, поддерживающим векторную арифметику. Основная особенность реализации **valarray** — оптимизация вычислений для больших векторов. А с точки зрения пользователя, основное — это множество операций, недоступных в **vector.**

Приведенный ниже код создает вектор из 10 действительных чисел и заполняет его числами от 1 до 10:

```
#include <valarray>
...
valarray<double> V(10);
for (int i = 1; i <= 10; i++)
  V[i] = i;
```

Теперь посмотрим использование ряда операций, введенных в **valarray.** Оператор

```
V *= 10;
```

масштабирует вектор, умножая каждый его элемент на 10. Не правда ли, компактно и просто?

Следующие операторы манипулируют тремя векторами **valarray** V, V1 и V2 одинакового размера и с элементами одного типа:

```
V = 10 * V1; // V[i] = 10 * V1[i]
V = V1 + V2; // V[i] = V1[i] + V2[i]
V = V1 * V2; // V[i] = V1[i] * V2[i]
```

Первый из этих операторов заносит в V элементы вектора V1, умноженные на 10. Второй оператор осуществляет сложение двух векторов V1 и V2, занося в элементы V суммы соответствующих элементов складываемых векторов. Третий оператор выглядит формально как умножение векторов. Но это не так. Это не скалярное или векторное произведение. Просто в элементы вектора V заносятся произведения соответствующих элементов векторов V1 и V2.

В векторах valarray определен целый ряд полезных функций-элементов: сдвиг (shift) и циклический сдвиг (cshift) элементов на заданное число позиций, определение суммы значений элементов (sum), минимального (min) и максимально (max) значений, вызов функций, вычисляемых сразу для всех элементов. Имеется также возможность с помощью срезов (slice) работать с одномерной последовательностью как с матрицей. Словом, мы, к сожалению, должны завершить рассмотрение векторов и перейти к контейнерам других типов. Но, надеюсь, что даже приведенных сведений достаточно, чтобы можно было эффективно использовать векторы во многих задачах. А остальные сведения придется черпать из встроенной в C++Builder 6 справки по STL или из специальной литературы.

5.4.4 Связные списки

Базовым шаблоном класса связных списков является list. Он описывает последовательность, оптимизированную для вставки и удаления элементов. Каждый элемент имеет указатели на предыдущий элемент и последующий. Такой список называется двусвязным. Аналогичные связи, только однонаправленные, описаны в разд. 2.12.2 при рассмотрении самоадресуемых структур. По списку можно перемещаться в обе стороны, но только от одного элемента к соседнему. Поэтому, в отличие от векторов, в списках не предусмотрена операция индексации и нет итераторов с произвольным доступом (см. разд. 5.5.4). Доступен только двунаправленный итератор (разд. 5.5.4), который позволяет смещаться только на одну позицию к концу или началу последовательности операциями инкремента и декремента. Зато быстро осуществляются вставки и удаления элементов, поскольку при этом остальные элементы не перемещаются. Они остаются на отведенных им в памяти местах. Манипуляции осуществляются только над указателями, которые и объединяют элементы в некоторую последовательность.

Базовый шаблон списка list:

```
template <class T, class Allocator = allocator<T> >
 class list (
...
)
```

подключается к проекту заголовочным файлом *list.* Класс T — это класс элементов списка. Распределитель памяти **Allocator,** как было сказано в разд. 5.3, можно обычно не задавать, используя распределитель по умолчанию.

Создание объектов списков производится так же, как для других контейнеров. Например, следующие операторы создают и заполняют два списка строк: **Dep1 и Dep2.**

```
#include <list>
#include <string>
using namespace std;
...
list<string> Depl, Dep2;
Dep1.pushback("Сидоров");
Dep1.push_back("Иванов");
Dep1.pushback("Петров");
Dep2.pushback("Семенов");
Dep2.pushback("Иванов");
Dep2.pushback("Павлов");
```

В результате список **Depl** будет иметь вид: ("Сидоров", "Иванов", "Петров"), а список **Dep2:** ("Семенов", "Иванов", "Павлов").

В этом примере для строк использован тип **std::string,** рассмотренный в разд. 5.6.

В list имеются все типы и операции, описанные в разд. 5.4.3 для класса **vector,** кроме операции индексации и функций **capacity и reserve.** В частности, вставка и удаление элементов могут осуществлять функциями **insert и erase.** Но, кроме того, в классе имеются следующие ранее не описанные функции-элементы:

Функция	Синтаксис / Описание
merge	**void merge(list<T, Allocator>& x)** Объединяет два списка, сортированных с использованием операции "<", перемещая элементы списка x на соответствующие позиции данного списка **template<class BinaryPredicate>** **void merge(list<T, Allocator>& x, BinaryPredicate comp)** Объединяет два списка, сортированных с помощью функции **comp**, перемещая элементы списка x на соответствующие позиции данного списка
pop_front	**void pop_front()** Удаляет первый элемент списка
push_front	**void push_front(const T& x)** Вставляет копию элемента x в начало списка
remove	**void remove(const T& value)** Удаляет все элементы, значение которых равно **value** **template <class Predicate>** **void remove_if(Predicate pred)** Удаляет все элементы, для значений которых **pred** возвращает **true**
sort	**void sort()** Сортирует список с использованием операции "<" **template<class BinaryPredicate>** **void sort(BinaryPredicate comp)** Сортирует список с использованием **comp()**

Функция	Синтаксис / Описание
splice	**void splice(iterator position, list<T, Allocator>& x)** Перемещает все элементы списка x в позицию **position** **void splice(iterator position, list<T, Allocator>& x, iterator i)** Перемещает элемент списка x, на который указывает итератор i, в позицию **position** **void splice(iterator position, list<T, Allocator >& x, iterator first, iterator last)** Перемещает элементы списка x, лежащие в интервале [first, last), в позицию **position**
unique	**void unique()** Удаляет из списка дубли элементов, оставляя из группы дублей первый элемент **template <class BinaryPredicate>** **void unique(BinaryPredicate binary_pred)** Оставляет в списке по одному из групп элементов, для которых **binary_pred** сообщает об их эквивалентности

Функция **splice** позволяет перемещать элементы из одного списка в другой или осуществлять перемещение элементов в пределах одного списка. Перемещение означает, что из прежней позиции элемент удаляется. Первая форма функции перемещает все элементы из списка x в позицию, указанную итератором **position.** В результате список x оказывается пустым. Например, оператор

```
Dep1.splice(Dep1.begin(), Dep2);
```

перенесет весь список **Dep2** в начало списка **Depl.** В результате в **Depl** сформируется список: ("Семенов", "Иванов", "Павлов", "Сидоров", "Иванов", "Петров"), а список **Dep2** окажется пустым.

Вторая форма функции **splice** перемещает элемент списка x, на который указывает итератор i, в позицию итератора **position.** Третья форма функции перемещает элементы списка x, лежащие в интервале **[first, last),** в указанную позицию. Обратите внимание, что перемещаются элементы, включая тот, на который указывает **first,** и исключая тот, на который указывает **last.** Например, следующий оператор переставляет в созданном ранее объединенном списке **Depl** элементы, начиная с четвертого и до конца, в начало списка:

```
Dep1.splice(Dep1.begin(), Depl, ++(++(++Dep1.begin())), Dep1.end());
```

В результате получится список: ("Сидоров", "Иванов", "Петров", "Семенов", "Иванов", "Павлов").

В приведенном примере для того, чтобы указать на четвертый элемент списка пришлось три раза применить к итератору **Dep1.begin()** операцию инкремента. Это следствие того, что в списках нет итераторов с произвольным доступом, и приходится многократно применять инкремент или декремент, чтобы добраться до нужного элемента. Но зато перестановка элементов осуществляется очень быстро, так как сводится к изменению указателей и не требует перемещения самих элементов в памяти.

Функция **sort** обеспечивает сортировку списка. Например, оператор

```
Dep1.sort();
```

сортирует список **Dep1**. Если в **Depl** ранее был создан описанный выше объединенный список, то после сортировки он приобретет вид: ("Иванов", "Иванов", "Павлов", "Петров", "Сидоров").

Если есть элементы с одинаковыми значениями (в нашем примере "Иванов"), то после сортировки их последовательность будет той, какая была в несортированном списке. Сортировка не нарушает никаких итераторов, указывавших ранее на элементы списка. После сортировки они будут продолжать указывать на те же элементы.

Функция **unique** просматривает значения элементов списка, и если обнаруживаются группы элементов с одинаковыми значениями, то оставляется первый из них, а остальные удаляются. Правда, дубли удаляются только в том случае, когда они расположены в списке рядом друг с другом. В частности, это гарантировано в сортированном списке. Так что применение **unique** к полученному выше сортированному списку **Depl** оператора

```
Dep1.unique();
```

удалит из списка второй из элементов со значением "Иванов".

Функция **remove** удаляет из списка все элементы, значения которых равны заданному. Например, оператор

```
Dep1.remove("Иванов");
```

удалит из списка **Depl** все элементы со значением "Иванов".

Функция **merge** объединяет два сортированных списка, перемещая элементы списка x на соответствующие позиции данного списка. Например, если применить сортировку к рассмотренным в начале спискам **Depl и Dep2,** а затем выполнить оператор

```
Dep1.merge(Dep2);
```

то получим тот же сортированный список, который получали ранее последовательным применением функций **splice и sort.**

Мы рассмотрели функции-элементы списка. Но к списку могут применяться и глобальные функции библиотеки. Например, для поиска и замены в списках может использоваться функция **find,** объявленная в файле *algorithm* следующим образом:

```
#include <algorithm>
template <class InputIterator, class T>
  InputIterator find(InputIterator first,  InputIterator last,
                     const T& value);
```

Параметры **first и last** указывают итераторы, определяющие диапазон поиска в контейнере: **[first, last),** т.е. включая элемент, на который указывает **first,** и исключая тот, на который указывает **last.** Параметр **value** — это искомое значение элемента. Функция возвращает итератор, указывающий на найденный элемент, или **last,** если элемент не найден. Пусть, например, в нашем списке, который получился после объединения **Depl и Dep2,** мы хотим различить двух Ивановых, добавив в значения соответствующих элементов их инициалы. Это может быть сделано, например, следующим кодом:

```
tinclude <functional>
...
*find(Dep1.begin(), Dep1.endO, "Иванов") += " A.A.";
*find(Dep1.begin(), Dep1.end(), "Иванов") += " И.И.";
```

После выполнения первого оператор первый элемент со значением "Иванов" изменит свое значение на "Иванов A.A.". Поэтому при выполнении следующего оператора функций **find** укажет второго Иванова и присвоит ему инициалы "И.И.".

Приведенный пример неэффективен, так как второй поиск ведется опять во всем списке, хотя надо было бы проводить его в оставшейся части списка. Поэтому более разумен следующий код:

```
list<string>::iterator I = find(Dep1.begin(), Dep1.end(), "Иванов");
*I += " A.A.";
*find(I, Dep1.end(), "Иванов") += " И.И.";
```

Следующий пример демонстрирует замену значений всех элементов списка, первоначально имевших некоторое заданное значение. Конечно, пример странный, но пусть мы хотим переименовать всех Ивановых в списке в Петровых. Это можно сделать следующим образом:

```
list<string>::iterator I = Dep1.begin();
while ((I = find(I, Depl.endO, "Иванов"))!= Depl.endO)
      *I = "Петров";
```

Здесь перед присваиванием нового значения проверяется сравнением с **Depl.endO,** действительно ли найден очередной элемент. И только если найден, его значение изменяется.

Мы рассмотрели двусвязный список **list.** В библиотеке STL, используемой в C++Builder 6, имеется также класс **slist** — односвязный список, в котором каждый элемент имеет указатель только на следующий элемент и «не знает» предыдущего. Класс **slist** во многом похож на **list.** К тому же, он не входит в стандарт библиотеки. Так что останавливаться на нем не будем.

5.4.5 Очереди

Очередь представляет собой последовательность, отличающуюся тем, что вставка и удаление элементов производится, как правило, только в ее начале или в конце. Очереди используются в приложениях достаточно часто. Например, они применяются, если в ваше приложение поступают какие-то сообщения от других приложений. Пока приложение не готово к их обработке, они накапливаются в очереди, добавляясь в ее конец. А при обработке они поочередно извлекаются из начала очереди, т.е. в порядке их поступления. Это дисциплина обслуживания FIFO: первым пришел — первым ушел. Возможна и другая организация очереди — стек, в котором и ввод и вывод элементов производится в конце очереди. В этом случае первым обрабатывается элемент, который был поставлен в очередь последним. Стеки используются при обработке вложенных запросов, вложенных вызовов каких-то процедур, при грамматическом разборе выражений. Во всех этих случаях обработку надо начинать с самого внутреннего из вложенных элементов, пришедшего последним, поскольку результат обработки должен передаваться во внешний элемент.

Наиболее общий вид очереди реализован шаблоном **deque** (произносится «дек»). Это очередь, допускающая итераторы с произвольным доступом и эффективно работающая как с последним, так и с первым элементом. Отсюда и название «deque» — double-ended queue (очередь с двумя концами).

Класс **deque** очень похож на вектор **vector** (см. разд. 5.4.2). Основное отличие в том, что в данном классе оптимизированы операции с первым и последним элементами. А в остальном, с точки зрения пользователя, эти классы идентичны.

Базовый шаблон списка **deque:**

```
template <class T, class Allocator = allocator<T> >
        class deque;
```

подключается к проекту заголовочным файлом *deque.* Класс T — это класс элементов списка. Распределитель памяти **Allocator,** как было сказано в разд. 5.3, можно обычно не задавать, используя распределитель по умолчанию.

Создание объектов списков производится так же, как для других контейнеров. Например, следующие операторы создают и заполняют очередь текстовых элементов **Equ:**

```
#include <list>
#include <string>
using namespace std;
...
deque<string> Equ;
for(int i=1; i<4; i++)
  Equ.push_back(("Сообщение " + IntToStr(i)).c_str());
```

В этом примере для строк использован тип **std::string,** рассмотренный в разд. 5.6.

В deque имеются все типы, операции и функции-элементы, описанные в разд. 5.4.3 для класса **vector,** кроме операции индексации и функций **capacity** и reserve. Так что работа с очередью проводится так же, как с вектором. Пусть, например, в приложении объявлена строковая переменная s, которая должна содержать сообщения, которые заносятся и читаются из очереди:

```
string s;
```

Занесение в нее поступающего сообщения из какого-то элемента VCL может оформляться, например, следующим образом:

```
s = Edit1->Text.c_str();
```

Тогда при организации очереди FIFO занесение очередного сообщения в очередь производится оператором

```
Equ.push_back(s);
```

А чтение и удаление сообщения из очереди производится операторами

```
if(!Equ.empty())
{
 s = Equ.front();
 Equ.pop_front();
 ...
)
else ...
```

При организации стека занесение в стек осуществляется так же, как показано выше, а чтение тоже аналогично, только операции с первым элементом надо заменить аналогичными операциями с последним элементом:

```
if(! Equ.empty())
{
 s = Equ.back();
 Equ.pop_back();
 ...
}
else ...
```

Несмотря на то, что **deque** — это очередь, в классе может использоваться произвольный доступ к любым элементом, функция **insert** и др. Эта избыточность возможностей устранена в адаптерах (см. разд. 5.4.1) **stack** (стек) и **queue** (очередь), обеспечивающих построение обычного стека и обычной очереди на основе заданного типа контейнера. Они представляют собой просто интерфейсы, которые могут относиться к различным видам контейнеров.

Заголовок шаблона **stack,** подключающийся к проекту заголовочным файлом *stack,* имеет вид:

```
# include <stack>

template <class T, class Container>
  class stack
```

Параметр T, как всегда, указывает класс элементов стека. А параметр **Container** указывает класс контейнера, на основе которого создается стек. По умолчанию **Container** — класс **deque** с элементами T. Но в качестве контейнера может использоваться любой другой тип, который имеет функции back(), **push_back() и pop_back().**

Например, операторы

```
# include <stack>
#include <string>
using namespace std;
...
stack<string>  St;
```

создают стек на основе **deque** с элементами типа **string.** А оператор

```
stack<string, vector<string> > St;
```

создают стек на основе **vector** с элементами типа **string.** Обратите внимание на необходимость пробела между двумя символами ">", чтобы они не были восприняты как операция ">>".

Шаблон стека отменяет доступ ко всем функциям контейнера, на основе которого он создан, оставляя только **empty()** для проверки, не пуст ли стек, **size()** – число элементов в стеке, и переименовывая три функции, работающие с вершиной стека (так в стеках называют последний элемент последовательности):

Функция	**Синтаксис** и реализация / Описание
top	reference top() { return **c.back();** } const_reference top() const { return c.back(); **}** Возвращает, не удаляя, вершину стека — последний элемент последовательности
push	void **push(const value_type&** x) { c.push_back(x); } Заносит значение x в вершину стека — в последний элемент последовательности
pop	void pop(){ **c.pop_back();** } Выталкивает (удаляет) элемент в вершине стека (последний элемент последовательности)

Таким образом, занесение в стек очередного элемента в нашем примере может осуществляться операторами:

```
string s;
...
St.push(s);
```

Чтение вершины стека без удаления осуществляется оператором:

```
if(! St.empty()) s = St.top();
```

Удаление элемента из вершины осуществляется оператором:

```
if(! St.empty()) St.pop();
```

В двух последних операторах перед чтением и удалением проверяется, не является ли стек пустым.

Очередь **queue** также является адаптером, призванным выделить из контейнера, на базе которого строится очередь, только то, что для этой очереди требуется. Заголовок шаблона **queue,** подключающийся к проекту заголовочным файлом *queue,* имеет вид:

```
template <class T, class Container>
  class queue
```

Как и в стеке, параметр T указывает класс элементов очереди, а параметр **Container** указывает класс контейнера, на основе которого создается очередь. По умолчанию **Container** — класс **deque** с элементами T. Но в качестве контейнера может использоваться любой другой тип, который имеет функции **front(), back(), push_back() и pop_back().**

Например, операторы

```
#include <queue>
#include <string>
using namespace std;
...
queue<string> Que;
```

создают очередь на основе **deque** с элементами типа **string.** А оператор

```
queue<string, vector<string> > Que;
```

создает очередь на основе **vector** с элементами типа string. Опять обратите внимание на необходимость пробела между двумя символами ">", чтобы они не были восприняты как операция ">>".

Адаптер подразумевает очередь с дисциплиной FIFO. Так что он содержит только такие функции-элементы, которые дают доступ к первому и последнему элементам, причем первый элемент можно только читать и удалять, а последний — записывать и читать. Набор функций весьма скромный: **empty(), size(), front()** — чтение первого элемента, **back()** — чтение последнего элемента, **push (const value_type& x)** — запись в конец очереди значения **x, pop()** — удаление первого элемента. Приведем примеры:

```
string s;
...
Que.push(s);                    // запись в очередь
if(! Que.empty()) s = Que.back(); // Чтение последнего элемента
if(! Que.empty()) s = Que.front(); // Чтение первого элемента
if(! Que.empty()) Que.pop();     // Удаление последнего элемента
```

В библиотеке имеется еще адаптер **priority_queue** — очередь с приоритетом. Она может реализовываться на основе контейнеров **vector** (это реализация по умолчанию) или **deque.** Набор функций еще более скромный, чем в **queue: empty(), size(), top()** - - чтение элемента с наивысшим приоритетом, **push (const value_type& x)** — запись элемента x в очередь, **pop()** — удаление элемента с наивысшим приоритетом.

Особенность очереди с приоритетом состоит в том, что вставка нового элемента осуществляется в позицию, зависящую от его величины (приоритета). Элементы всегда располагаются так, что элемент с наивысшим приоритетом располагается в вершине очереди. Именно к нему обеспечивает доступ функция top. И именно он удаляется операцией **pop.**

Рассмотрим примеры. Операторы

```
# include <queue>
using namespace std;
...
priority_queue<int> PQ;
PQ.push(1);
PQ.push(10);
PQ.push(5);
```

создают очередь целых чисел с приоритетом. Тогда оператор

```
int i = PQ.top();
```

вернет значение 10 — максимальное число, а оператор

```
PQ.pop();
```

удалит значение 10 из очереди, после чего максимальным числом, расположенным в вершине, станет 5.

Для сравнения элементов при упорядочивании очереди используется по умолчанию функция-объект **less<T>.** Эту функцию можно указать явно в объявлении очереди. Например, приведенный ранее оператор создания очереди **PQ** может иметь вид:

```
priority_queue<int, vector<int>, less<int> > PQ;
```

Здесь вторым аргументом конструктора задается контейнер **vector<int>**, на основе которого создается очередь, а третьим аргументом задается функция сравнения **less<int>**. В данном случае эти аргументы соответствуют значениям по умолчанию. Но вы можете указать во втором аргументе **deque<int>,** изменив тем самым базовый контейнер. Можете изменить функцию сравнения. Например, если вы создадите очередь оператором

```
priority_queue<int, vector<int>, greater<int> > PQ;
```

то в вершине очереди будет располагаться не максимальное, а минимальное число. Можете написать и свою собственную функцию сравнения и упорядочивать очередь по собственным критериям.

5.4.6 Ассоциативные контейнеры

5.4.6.1 Общие сведения

Ассоциативные контейнеры обеспечивают доступ к элементам с помощью *ключей*. Элементы в контейнере упорядочены в соответствии с заданными ключами. Для упорядочивания используются функции сравнения. По умолчанию это функция **less<T>,** обеспечивающая упорядочивание по нарастанию. При этом элементы контейнера должны поддерживать операцию отношения "<".

В контейнерах set и **multiset** ключами являются сами значения элементов. В контейнерах **map** и **multimap** каждый элемент имеет ключ и ассоциированное с ним значение. Контейнеры могут допускать наличие элементов с одинаковыми значениями ключей **(multiset и multimap),** или запрещать наличие дубликатов **(set** и **map).**

Тип ключа задается как **key_type.** Он может совпадать с типом значения элемента **value_type** (для **set** и **multiset** это обязательно), или может отличаться от него.

Помимо обычных для контейнеров функций-элементов, в ассоциативных контейнерах определены следующие функции-элементы:

Функция	Синтаксис / Описание
find	iterator find(const key_type& x) const; Возвращает итератор, указывающий на элемент со значением ключа x
lower_bound	iterator lower_bound(const key_type& x) const; Возвращает итератор, указывающий на первый элемент, значение ключа которого больше или равно x. Если такой элемент не находится, возвращается end()
upper_bound	iterator upper_bound(const key_type& x) const; Возвращает итератор, указывающий на первый элемент, значение ключа которого строго больше x. Если такой элемент не находится, возвращается end()
count	size_type count(const key_type& x) const; Возвращает число элементов, имеющих значение ключа x

Ассоциативные контейнеры поддерживают двунаправленные итераторы, но не поддерживают итераторы с произвольным доступом. Впрочем, с помощью функции **find** можно получить доступ к любому элементу.

5.4.6.2 Контейнеры multiset и set

Эти контейнеры могут содержать упорядоченное множество элементов. Ключами, по которым проводится упорядочивание, являются значения элементов. По умолчанию для упорядочивания используется функция **less<T>,** что обеспечивает упорядочивание в порядке возрастания. Различие между **multiset** и **set** проявляется в том, что set не допускает включения нескольких элементов с одинаковыми значениями ключей. В остальном контейнеры эквивалентны. Оба типа контейнеров описаны в заголовочном файле <set>.

Создание и заполнение множеств осуществляется несколькими способами, которые иллюстрируются приведенными ниже примерами.

```
#include <set>
using namespace std;

multiset<int> MS;
MS.insert (7);
MS.insert(4);
MS.insert(4);
MS.insert(1);
MS.insert(9);
// Результат {1, 4, 4, 7, 9}

int A[4] = {6, 4, 1, 10};
set<int> MS1(A, A+4);
// Результат {1, 4, 6, 10}

multiset<int, greater<int> > MS2(MS.begin(), MS.endO);
// Результат {9, 7, 4, 4, 1}

multiset<int> MS3;
MS3.insert(MS.begin(), MS.end());
// Результат {1, 4, 4, 7, 9}
```

Множество **MS** типа **multiset** заполнено с помощью последовательного применения функции **insert.** Как видно из комментария, приведенного в тексте, последовательность элементов определяется их значениями, а не той последовательностью, в которой они добавлялись в контейнер.

Множество **MS1** типа set сформировано из обычного одномерного массива целых чисел. Применен конструктор, указывающий итераторы начала и конца контейнера или его части, из которой загружаются значения элементов. В результате создается множество, содержащее те же элементы, которые были в массиве, но в множестве они упорядочены по значениям.

При создании множества **MS2** применен аналогичный конструктор, загружающий элементы множества MS. Но вместо используемой по умолчанию функции сравнения **less** указана функция **greater.** Поэтому элементы упорядочиваются в порядке убывания.

При создании множества **MS3** применена функция **insert,** в которую передаются итераторы начала и конца контейнера или его части, из которой загружаются значения элементов. В данном случае в MS3 вставляются все элементы множества **MS.**

Можно задавать и собственную функцию сравнения. Она должна принимать два параметра соответствующего типа и возвращать **true,** если значение первого параметра меньше второго. Пусть, например, мы хотим создать копию MS4 множества MS, в которой сначала располагаются нечетные числа, а потом четные, причем внутри каждой группы чисел элементы располагаются в порядке нараста-

ния. Это можно оформить следующим образом (см. подробнее о функциях-объектах в разд. *5.8).* Введите глобальную переменную:

```
struct comp
{
  bool operator () (const int I1, const int I2) const
  {
   if ( (I1 % 2) == (I2 % 2) )
    return (I1 < I2);
   else return (I1 % 2);
  }
}
```

Далее вы можете ссылаться на эту переменную как на функцию сравнения:

```
multiset<int, comp > MS4(MS.begin(), MS.end O);
```

В результате будет создан контейнер, в котором элементы расположатся в следующей последовательности: {1, 7, 9, 4, 4}.

Теперь рассмотрим некоторые примеры работы с множествами. Следующий код обеспечивает просмотр по порядку всех элементов множества и вывод их значений пользователю:

```
AnsiString S = "";
multiset<int>::iterator It;
It = MS.begin();
while (It != MS.end())
 S += IntToStr(*It++) + "\t";
ShowMessage(S);
```

В коде создается итератор **It,** через который в цикле читаются все значения множества и заносятся в строку S.

Следующий оператор определяет число элементов, имеющих значение, указанное пользователем в окне **Edit1:**

```
int i = MS.count(StrToInt(Edit1->Text))
```

Например, если пользователь укажет "4", i равно 2, а если пользователь укажет "5", *i* равно 0.

Следующий код с помощью функции **find** (см. разд. 5.4.6.1) сообщает пользователю, имеется ли в множестве значение, указанное в окне **Editl:**

```
if (MS.find(StrToInt(Edit1->Text)) == MS.end())
   ShowMessage("отсутствует");
 else ShowMessage("найдено");
```

Оператор

```
MS.erase(MS.find(4));
```

удаляет из множества элемент, значение которого равно 4. Как видим, с помощью функции **find** легко получить доступ к любому элементу, хотя произвольный доступ в ассоциированных контейнерах не предусмотрен.

Оператор

```
MS.erase(MS.find(4), MS.end());
```

удаляет из множества все элементы, начиная с того, значение которого равно 4, и до конца.

Оператор

```
ShowMessage(IntToStr(*MS.lower_bound(StrToInt(Edit1->Text))) +
            " - " + IntToStr(*MS.upper_bound(StrToInt(Edit1->Text))));
```

с помощью функций **lower_bound и upper_bound** (см. разд. 5.4.6.1) определяет значения верхней границы множества, включая или не включая значение самого элемента, записанное в окне **Editl.** Для нашего примера для значения "7" будет выдан текст "7 - 9", а для значения "5" — текст "7 – 7".

Операторы

```
#include <algorithm>

multiset<int> MS4;
insert_iterator<multiset<int> > It1(MS4, MS4.begin());
set_union(MS.begin(), MS.end(), MS1.begin(), MS1.end(), It1);
```

создают множество MS4, итератор **Itl** этого множества, а затем алгоритмом **set_union** (см. алгоритмы в разд. 5.7) заносят в это множество объединение множеств MS и **MS1.** В качестве аргументов в алгоритм передаются итераторы начала и конца первого множества, итераторы начала и конца второго множества и итератор результирующего множества. Результатом объединения является множество, содержащее все элементы первого, плюс элементы второго, отсутствующие в первом. Отличие от классического определения объединения множеств в том, что если какой-то элемент входит в первое множество п раз, а во второе m раз, то в результат этот элемент войдет max(n, m) раз. Для нашего примера множество MS4 получится следующим: {1, 4, 4, 6, 7, 9, 10}.

Если в приведенном коде заменить вызов **set_union** на вызов **set_intersection,** то будет получено пересечение двух множеств: множество, состоящее из элементов, входящих и в первое, и во второе множество. Отличие от классического определения пересечения множеств в том, что если какой-то элемент входит в первое множество п раз, а во второе m раз, то в результат этот элемент войдет min(n, m) раз. Для нашего примера множество **MS4** получится следующим: {1, 4}.

Если в приведенном коде вызывать **set_difference,** то будет получена разность двух множеств: множество, состоящее из элементов первого множества, отсутствующих во втором. Для "нашего примера множество **MS4** получится следующим: {4, 7, 9}.

Наконец, если в приведенном коде вызывать **set_symmetric_difference,** то будет получено множество, в которое входят элементы первого множества, отсутствующие во втором, плюс элементы второго множества, отсутствующие в первом. Для нашего примера множество MS4 получится следующим: {4, 6, 7, 9, 10}.

Оператор

```
if (includes(MS.begin(), MS.end(),MS1.begin(), MS1.end()))
   ShowMessage("найдено");
else  ShowMessage("отсутствует");
```

показывает, имеется ли в множестве, заданном первыми двумя итераторами (в нашем случае MS), подмножество, заданное следующими двумя итераторами (в нашем случае **MS1).**

Теперь остановимся на различиях множеств **multiset** и **set.** Различие заключается в том, что множество set не допускает включения в него дубликатов — элементов с одинаковыми значениями. При попытке включить в set дубликат функцией **insert** с одним аргументом включение просто не произойдет. В связи с этой особенностью функции **insert** с одним аргументом различаются для разных классов множеств. В **multiset** эта функция возвращает итератор, указывающий на вставленный элемент. А в set функция **insert** возвращает объект типа **pair.** Этот тип содержит значения двух величин указанных типов. При создании объекта типа **pair** в шаблон передается два типа значений. А сами значения хранятся в полях **first и second.**

Функция **insert** заносит в поле **first** итератор, указывающий на вставленный элемент (если вставка произошла), а в поле **second** — булево значение: **true,** если элемент вставился, и **false,** если вставка не получилась. Следующий код демонстрирует применение типа **pair** при вставке элемента в множество set:

```
pair< set<int>::iterator, b6ol> p;
p = MS1.insert(StrToInt(Edit1->Text));
if (p.second)
```

```
  ShowMessage("Вставлен элемент " + IntToStr(*p.first));
else ShowMessage("Элемент " + IntToStr(*p.first) +
                 " не вставлен");
```

Впрочем, если проверка вставки не требуется, можно просто применять функцию **insert,** игнорируя возвращаемый ею результат.

5.4.6.3 Контейнеры multimap и map

Эти контейнеры могут содержать упорядоченное множество элементов. Каждый элемент характеризуется парой величин: ключом и значением. Они хранятся в описанном в предыдущем разделе типе **pair<const Key, Data>.** Здесь Key – ключ, **Data** — значение элемента. Например, ключом может быть фамилия сотрудника, а значением элемента — его год рождения, или ключи — названия отделов, а значения — фамилии сотрудников соответствующего отдела.

Упорядочивание элементов производится по ключам. Ключ — неотъемлемая и неизменяемая часть элемента, в то время как значение элемента можно изменять. По умолчанию для упорядочивания используется функция **less<T>,** что обеспечивает упорядочивание в порядке возрастания ключей, которые сравниваются по операции "<<".

Различие между **multimap** и **map** проявляется в том, что map не допускает включения нескольких элементов с одинаковыми значениями ключей. В остальном контейнеры эквивалентны. Оба типа контейнеров описаны в заголовочном файле **<map>.**

Рассмотрим в качестве примера контейнер, ключами элементов которого являются фамилии сотрудников (строки), а значениями элементов — года рождения (целые числа). Создание и заполнение такого контейнера может быть сделано следующими операторами:

```
#include <map>
using namespace std;

typedef multimap<AnsiString, int> TMM;
TMM MM1;
MM1.insert(TMM::value_type("Сидоров", 1970));
MM1.insert(TMM::value_type("Иванов", 1980));
MM1.insert(TMM::value_type("Петров", 1975));
MM1.insert(TMM::value_type("Иванов", 1960));
```

В приведенном коде объявление типа **TMM** введено просто для того, чтобы в дальнейшем избежать многократного повторения описания типа **multimap<AnsiString, int>.**

В результате выполнения приведенного кода в контейнере **MM1** элементы расположатся следующим образом:

Иванов	1980
Иванов	1960
Петров	1975
Сидоров	1970

Элементы упорядочены по алфавиту, а при одинаковых ключах (два Иванова) размещаются в последовательности добавления элементов в контейнер.

Можно создавать контейнер на основе другого, уже существующего:

```
map<AnsiString, int, greater<AnsiString> > MM2(MM1.begin(), MM1.end());
```

Данный оператор создает контейнер **MM2** типа map, загружая в него элементы ранее созданного контейнера **MM1.** Одновременно задана функция сравнения **greater** вместо используемой по умолчанию **less.** В результате в контейнере **MM2** элементы расположатся следующим образом:

```
Сидоров          1970
Петров           1975
Иванов           1980
```

Элементы расположены в последовательности убывания ключей, а из двух Ивановых остался один, так как контейнер типа **tap** не допускает наличия дубликатов.

Следующий код отображает список всех элементов контейнера:

```
AnsiString S = "";
TMM::iterator It = MM1.begin();
while (It != MM1.endO)
{
 S += (*It).first + " - " + IntToStr((*It).second) + "\n";
 It++;
}
ShowMessage(S);
```

Итератор It в цикле проходит все элементы контейнера. А доступ к ключам и значениям элементов осуществляется через поля **first и second.** В результате отобразится такой текст:

```
Иванов - 1980
Иванов - 1960
Петров - 1975
Сидоров — 1970
```

Следующий оператор иллюстрирует изменение значения элемента (уточняется год рождения Петрова):

```
if ((It = MM1.find("Петров")) == MM1.endO)
   ShowMessage("отсутствует");
else (*It).second = 1976;
```

Для доступа к элементу используется функция **find,** о которой уже говорилось в предыдущем разделе.

Мы рассмотрели специфику контейнеров **multimap** и **tap.** А в остальном работа с ними ведется так же, как было описано в предыдущем разделе для контейнеров **multiset и set.**

5.5 Итераторы

5.5.1 Общая характеристика итераторов

Итераторы являются обобщением указателей. Как и указатели, итераторы указывают на объекты. Но применительно к итераторам речь идет об указании на один из элементов набора объектов, хранящихся в некоем контейнере. Впрочем, имеется два итератора: **istream_iterator и ostream_iterator,** связанных не с контейнерами, а с входным и выходным потоками. Итераторы позволяют перемещаться по набору объектов в некоторой *последовательности:* к предыдущему элементу или к последующему.

Итераторы обеспечивают интерфейс между контейнером и алгоритмами, оперирующими с объектами в контейнере. При разработке алгоритма можно ничего не знать о деталях реализации контейнера, о том, как расположены в памяти его элементы. Достаточно оперировать с итераторами, и вы получите доступ к интересующим вас элементам.

В STL имеется 6 видов итераторов. Простейший итератор (Trivial Iterator) не имеет самостоятельного значения и используется только как базовый для построения других видов итераторов. Входной итератор (Input Iterator) обеспечивает только чтение и может перемещаться только от начала к концу последовательности.

Выходной итератор (Output Iterator) обеспечивает только запись, и может перемещаться только от начала к концу последовательности. Остальные виды итераторов обеспечивают и чтение, и запись. При этом однонаправленный итератор (Forward Iterator) может перемещаться только от начала к концу последовательности, двунаправленный итератор (Bidirectional Iterator) может перемещаться в оба конца последовательности, а итератор с произвольным доступом (Random Access Iterator) обеспечивает прямой доступ к любому элементу последовательности. Может возникнуть вопрос: зачем так много итераторов, если итератор с произвольным доступом может реализовывать все функции других итераторов? Основной ответ заключается в том, что *чем* больше функциональные возможности итератора, тем он сложнее и, значит, менее эффективен. Так что для каждой конкретной задачи надо стараться выбирать тот итератор, который обеспечивает ее решение и не имеет дополнительных возможностей, не реализуемых в данной задаче.

В каждый момент итератор указывает на некоторый текущий элемент последовательности. С помощью функции-элемента контейнера begin() вы можете присвоить итератору значение, указывающее на первый элемент последовательности. Аналогично функция-элемент **end()** указывает на позицию в контейнере, расположенную после последнего элемента последовательности. Таким образом, интервал, в котором расположены элементы: [begin(), end()). То есть последний элемент расположен в позиции, предшествующей **end()**. А разность указателей **end()** и begin() равна числу элементов последовательности.

Операция разыменования итератора (*) дает доступ к тому элементу, на который указывает итератор. Она является основной для записи и чтения с помощью итератора. Если p является итератором, а **x** — переменной, то оператор

```
*P = x;
```

записывает в контейнер или в выходной поток значение x, а оператор

```
x = *p;
```

читает значение текущего элемента в переменную x. В обоих случаях работает конструктор копии, создающий в соответствующем контейнере копию x или создающий в x копию элемента.

Все реально используемые итераторы поддерживают префиксную и постфиксную формы записи операции инкремента (++). Таким образом, они позволяют перемещаться от текущего элемента к последующему. Префиксная форма операции перемещает итератор на следующий элемент и возвращает новую позицию итератора. Постфиксная форма возвращает прежнюю позицию и затем перемещает итератор на новую позицию. Вообще говоря, постфиксный инкремент выполняется менее эффективно, чем префиксный, так как его реализация требует промежуточного возврата на прежнюю позицию. Так что в случаях, когда форма записи безразлична, лучше применять

```
++p;
```

чем

```
p++;
```

Приведем пример использования рассмотренных функций и операций. Пусть, например, вы имеете вектор (см. разд. 5.4.3) целых чисел V, заполненный некоторыми значениями. Создание и заполнение подобного вектора может быть осуществлено операторами:

```
#include <vector>
using namespace std;
...
vector<int> V;
for(int i = 1; i <= 10; i++)
  V.push_back(i);
```

Использованная в этом коде функция-элемент вектора **push_back** создает в контейнере V 10 элементов, содержащих числа от 1 до 10.

Большинство контейнеров имеют элементы с именем **iterator.** Это тип итератора соответствующего контейнера. Пусть после описанного выше создания вектора V вы записали операторы:

```
vector<int>::iterator I1 = V.begin O;
*I1++ = 11;
*I1 = 12;
Labell->Caption = *(++I1);
```

Первый из этих операторов создает итератор **I1** вектора и присваивает ему значение, указывающее на первый элемент последовательности. Второй оператор заносит в первый элемент последовательности число 11 и перемещает итератор к следующему элементу. Третий оператор задает значение 12 этому элементу (второму в последовательности). Третий оператор перемещает итератор **I1** на следующую позицию (к третьему элементу последовательности) и выводит в метку **Labell** значение этого элемента.

При итерациях по элементам последовательности надо следить, чтобы итератор не вышел за пределы числа элементов. Это можно делать, сравнивая его со значением **end().** Например, следующий код продолжает приведенный выше пример, отображая в диалоговом окне поочередно значения всех элементов вектора:

```
I1 = V.begin();
while (I1 != V.end())
   ShowMessage(*I1++);
```

Рассмотрим еще один пример. Пусть вы хотите в векторе V найти элемент, имеющий целое значение Num, и, если такой элемент найден, умножить его на 10. Эта задача решается следующим кодом:

```
vector<int>::iterator I1 = find(V.begin(), V.endO, Num);
if(I1 != V.endO )
{
 *I1 *=, 10;
 ShowMessage("Позиция: " + (AnsiString)(I1 - V.begin() + 1) +
             ", значение:" + *I1);
}
else ShowMessage("Элемент не найден");
```

Первый оператор создает итератор **I1** и заносит в него результат поиска, осуществляемого функцией **find.** Ее объявление:

```
#include <algorithm>
template <class InputIterator, class T>
  InputIterator find(InputIterator first, Inputlterator last,
                     const T& value);
```

Параметры **first** и **last** определяют диапазон поиска, а параметр **value** — это искомое значение. Функция возвращает позицию найденного элемента контейнера, или значение **end(),** если элемент не найден.

Выше была показана работа с итератором контейнера **iterator.** Многие контейнеры имеют также итераторы с именем **reverse_iterator.** Это итераторы, просматривающие последовательность в обратном направлении, от конца к началу. Для инверсных итераторов вместо **begin() и end()** используются **rbegin() и rend().** При этом **rbegin()** указывает на последний элемент последовательности, а **rend()** указывает позицию, предшествующую первому элементу последовательности. Инкремент инверсного итератора означает его перемещение на 1 позицию к началу последовательности. Так что если в приведенном ранее примере просмотра вектора вы объявите итератор

```
vector<int>::revers_iterator Ir;
```

то код:

```
Ir = V.rbegin();
while(Ir != V.rend())
  ShowMessage(*Ir++);
```

обеспечит просмотр вектора от последнего элемента к первому.

Помимо итераторов **iterator и reverse_iterator** в контейнерах обычно предусматриваются итераторы с именами **const_iterator и const_reverse_iterator.** Они отличаются тем, что разрешают только чтение данных. При попытке записать данные через такой итератор компилятор выдаст сообщение об ошибке.

Для всех итераторов, кроме итераторов записи, можно определить расстояние между двумя итераторами. Расстояние (число элементов, размещенных между позициями, на которые они указывают) определяется функцией **distance.** Она имеет две формы:

```
template <class InputIterator>
inline iterator_traits<InputIterator>::difference_type
            distance(InputIterator first, InputIterator last);

template <class Inputlterator, class Distance>
        void distance(Inputlterator first, Inputlterator last,
                      Distances n);
```

Первая форма возвращает расстояние между двумя итераторами **first и last,** указывающими на одну и ту же последовательность. Например, оператор

```
Label1->Caption = distance(V.begin(), V.end());
```

выведет в метку **Labell** число элементов, имеющихся в векторе V. А оператор

```
Label2->Caption = distance(V.begin(), I1);
```

выведет в метку **Label2** номер того элемента последовательности, на который в данный момент указывает итератор **I1.**

Вторая форма функции **distance** была определена в прежнем стандарте STL. Пока она оставлена для обратной совместимости, поскольку еще не все компиляторы обрабатывают новую версию. Но в будущем поддержка второй формы функции может прекратиться. Эта форма ничего не возвращает, а добавляет (не записывает, а именно добавляет) в переменную n расстояние между итераторами **first и last.** Она менее удобна, чем первая форма, так как требует объявить дополнительную переменную n и инициализировать ее нулем, чтобы получить требуемое расстояние.

При использовании любой формы **distance** надо иметь в виду, что итератор **first** должен указывать на элемент, предшествующий итератору **last.** В противном случае поведение функции не определено.

Функция **distance** применима к большинству видов итераторов (кроме выходных), но она относительно медленная, так как ее реализация сводится к инкременту итератора **first** до тех пор, пока он не сравняется с **last.** Для длинных последовательностей это может требовать большого времени. Итераторы с произвольным доступом позволяют определить расстояние проще и намного эффективнее операцией вычитания. Так что для итераторов вектора в приведенных примерах целесообразнее использовать операцию вычитания:

```
Label1->Caption = V.end() - V.begin O;
Label2->Caption = I1 - V.begin();
```

В качестве контейнеров могут выступать обычные массивы языка С, а в качестве их итераторов — обычные указатели на элементы массивов. Повторим рассмотренные выше примеры, используя в качестве контейнера массив:

```
int V[10] = {1,2,3,4,5,6,7,8,9,10};
int * Vbegin = V, * Vend = V + 10;
int * I1;
```

Я специально использовал в этом примере те же идентификаторы, что и при работе с вектором. Указатели **Vbegin и Vend** — аналоги итераторов **begin() и end()**. А указатель **I1** аналог итератора, использованного в предыдущих примерах.

Рассмотренный ранее пример просмотра последовательности в данном случае может выглядеть так:

```
I1 = Vbegin;
while(I1 != Vend)
 ShowMessage(*I1++);
```

А пример поиска элемента с заданным значением Num и его умножением на 10 имеет вид:

```
I1 = find(Vbegin, Vend, Num);
if(I1 != Vend)
{
*I1 *= 10;
 ShowMessage("Позиция: " + (AnsiString)(I1 - V.begin() + 1) +
             ", значение:" + *I1);
}
else ShowMessage("Элемент не найден");
```

Сравнив эти коды с ранее приведенными примерами, вы увидите, что они различаются только идентификаторами **Vbegin и Vend.** Может быть, эти примеры, использующие обычные указатели, помогут вам глубже почувствовать смысл итераторов. Кроме того, как видите, обычные указатели могут использоваться в вызовах функций STL (в данном примере — в вызове **find).** Так что вы можете пользоваться многими функциями STL при работе с обычными массивами.

5.5.2 Итераторы чтения

Рассмотрим теперь различия между видами итераторов. Простейший итератор является базовым шаблоном для построения иных видов итераторов и самостоятельного значения не имеет. Этот итератор является аналогом обычного указателя. Для него определены операции разыменования (*), эквивалентности (==) и операция стрелка (->) для доступа к элементам класса. Впрочем, для тех компиляторов C++, которые не поддерживают операцию стрелка, предусмотрена возможность доступа к элементам класса совместным применением операций точка (.) и разыменования. Например, если объект it является итератором, тип которого предусматривает элемент т, то получить доступ к m можно выражением **it->m,** или, если подобная нотация компилятором не поддерживается, то выражением **(*it).m.**

Итератор чтения, помимо операций разыменования, эквивалентности и доступа к элементу класса, поддерживает префиксную и постфиксную формы записи операции инкремента. Таким образом, он позволяет перемещаться от текущего элемента к последующему. Определена также разность значений итераторов, которая показывает в виде целого числа количество элементов, расположенных между двумя итераторами. В приведенных ранее примерах практически во всех случаях, кроме нескольких операторов записи, мы имели дело с итераторами чтения.

Операция разыменования итератора чтения дает значение элемента в той позиции, на которую указывает итератор. Это значение можно использовать для копирования в какую-то переменную, в операциях сравнения и т.п. Но итератор чтения нельзя использовать для записи нового значения элемента. Иначе говоря, разыменование итератора не должно быть левым операндом оператора присваивания.

Особо надо сказать об итераторе чтения стандартного входного потока **istream_iterator.** При объявлении в его шаблон надо передать параметр T — тип объектов, которые читает данный итератор. Это должен быть такой тип, который имеет конструктор с параметрами по умолчанию и который может использоваться в выражениях вида cm >> T.

Итератор имеет два конструктора. Один из них имеет синтаксис:

```
istream_iterator(istream& s)
```

В качестве аргумента s в него передается входной поток. Обычно это стандартный входной поток cin — клавиатура. Второй конструктор не имеет параметров:

```
istream_iterator()
```

Он создает итератор стандартного входного потока с позицией после конца этого потока. Так что такой итератор можно использовать для итераций по потоку до его конца.

Ниже приведен пример консольной программы, читающей из входного потока два целых числа:

```
#include<iterator>
#include <iostream>
#include <conio.h>
using namespace std;
int main(void)
{
 istream_iterator<int> I (cin);
 int I1 = *I++;
 int I2 = *I;
 cout << I1 << ' ' << I2 << endl;
 getch();
}
```

Первый оператор функции **main** объявляет итератор I для чтения целых чисел из входного потока. Второй оператор читает из потока целое число, заносит его в переменную **I1** и смещает итератор операцией инкремента на следующую позицию. Далее аналогично читается в **I2** второе число, после чего прочитанные числа выводятся на экран и функцией getch создается ожидание нажатия пользователем какой-либо клавиши.

В приведенном коде смещение итератора осуществляется постфиксным инкрементом после чтения. Но в стандарте не оговорено, как должна выполняться такая операция, и разные компиляторы могут выполнять ее по-своему. Поэтому, если вы заботитесь о переносимости своего кода, лучше инкремент осуществлять отдельным оператором:

```
int I1 = *I;
I++;
int I2 = *I;
```

Во всяком случае, все шаблоны STL используют такой подход.

Можно читать значения непосредственно в контейнер. Например:

```
#include<vector>
#include<iterator>
#include <iostream>
#include <conio.h>
using namespace std;
int main(void)
{
 istream_iterator<int> I (cin);
 V.push_back(*I++);
 V.push_back(*I);
 copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
 getch();
}
```

В этом примере для вывода прочитанных значений на экран используется функция **copy:**

```
#include <algorithm>
```

```
template <class InputIterator, class OutputIterator>
 OutputIterator copy(InputIterator first, InputIterator last,
                     OutputIterator result);
```

Параметры **first и last** указывают начало и конец копируемых данных, а параметр **result** — это выходной итератор записи. В приведенном примере в качестве него использован итератор записи в выходной поток **ostream_iterator**, о котором будет сказано немного позднее.

В заключение рассказа об итераторе **istream_iterator** приведу пример, в котором используется вторая форма его конструктора:

```
#include<vector>
#include<iterator>
#include <iostream>
#include<conio.h>
using namespace std;
int main(void)
{
 copy(istream_iterator<int>(cin), istream_iterator<int>(),
      back_inserter(V));
 copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
 getch();
}
```

В этом коде для чтения используется только что описанная функция **copy.** В качестве ее второго аргумента указан итератор **istream_iterator<int>()**. Поскольку в его конструктор не переданы параметры, то этот итератор указывает позицию после конца входного потока. Таким образом, в вектор V будут читаться все вводимые пользователем данные. Чтение закончится после очередного нажатия пользователем клавиши Enter, если во введенном тексте окажется символ конца файла (клавиши Ctrl-Z), или окажутся какие-то символы (например, буквы), которые не могут встретиться в целом числе.

В качестве итератора, записывающего прочитанные данные в вектор V, использован адаптер итератора **back_inserter**. Он вставляет данные в конец последовательности соответствующего контейнера, указанного как его параметр (в приведенном примере контейнер — вектор V).

Итератор чтения, в частности, итератор чтения из потока, может повторно прочитать последний из прочитанных элементов, если не сдвигать его позицию. Правда, в стандарте это не оговорено, но в версии C++Builder 6 работает. Однако итератор чтения не может вернуться назад и прочитать какой-то из ранее прочитанных элементов. Поэтому итераторы чтения неприменимы в многопроходных алгоритмах, требующих многократного просмотра последовательности.

5.5.3 Итераторы записи

Теперь коротко рассмотрим итераторы записи. Они по набору операций близки к итераторам чтения. Но операторы записи не могут участвовать в операциях сравнения и с их помощью можно только записывать данные, но не читать их. Разыменование итератора чтения давало значение очередного читаемого элемента. А разыменование итератора записи определяет место, в которое должна производиться запись, а не значение элемента на этом месте. Например, если I — итератор записи, то оператор

```
*-I = t;
```

заносит в указываемую итератором позицию значение объекта t. Операция разыменования итератора записи может встречаться только в левом операнде оператора присваивания, как в приведенном примере.

Итераторы записи построены так, что после очередной записи ожидается инкремент итератора. А после инкремента ожидается запись. Так что последовательность записей предполагается такой:

```
*I = t1;
++I;
*I = t2;
```

или такой:

```
*I++ = t1;
*I = t2;
```

Иное чередование операций с итератором записи может привести к непредсказуемым последствиям. Например, неизвестно, как сработают следующие операторы:

```
*I = t1;
++I;
++I;
*I = t2;
```

Это зависит от конкретной реализации STL и компилятора. Так что надо рекомендовать не нарушать стандартизованную последовательность операций.

К числу итераторов записи относятся: **insert_iterator** — итератор вставки, **front_insert_iterator** — итератор вставки в начало последовательности, **back_insert_iterator** — итератор вставки в конец последовательности, **ostream_iterator** — итератор записи в выходной поток. Первые три из них — итераторы контейнеров.

Начнем их рассмотрение с итератора вставки **insert_iterator**. В его шаблон передается один тип — тип контейнера. Конструктор итератора имеет вид:

```
insert_iterator(Container& C, Container::iterator i)
```

Параметр C — контейнер, для которого создается итератор, а параметр i — итератор этого контейнера, указывающий позицию вставки.

Вставка в контейнер, содержащий некоторую последовательность, осуществляется следующим образом. Размер контейнера увеличивается на 1, все элементы, расположенные после позиции i, сдвигаются к концу на одну позицию, а в позицию i заносится записываемое значение. Этим запись через итератор вставки отличается от записи через итератор контейнера iterator (см. примеры в разд. 5.5.1), который просто заменяет элемент последовательности новым значением.

Приведем пример. Если V — вектор целых чисел, неоднократно фигурировавший в предыдущих разделах, то операторы

```
insert_iterator<vector<int> > I1(V, V.begin() + 3);
*I1++ = Num1;
*I1 = Num2;
```

вставят в четвертую и пятую позиции соответственно значения целых переменных Num1 и **Num2**, а позиции всех элементов, которые располагались ранее начиная с четвертой позиции увеличатся на 2. Соответственно на 2 увеличится и размер вектора.

В версии STL, используемой в C++Builder 6, можно для итератора вставки не делать инкремент между операциями записи. Очередная запись автоматически будет осуществлена в следующую позицию. А если между записями сделать несколько инкрементов, это тоже ни на что не повлияет. Так что в приведенном выше примере можно вообще убрать инкремент. Но если вы рассчитываете, что ваш код, может быть, будет компилироваться и в какой-то другой системе, отличной от **C++Builder**, то лучше не нарушать стандарт и чередовать запись с инкрементом.

Мы рассмотрели вставку в контейнер, содержащий последовательность. Имеются и другие виды контейнеров, в частности, ассоциативные контейнеры (см. разд. 5.4.6), в которых расположение элементов осуществляется в зависимости от их значений. В этих случаях текущая позиция итератора вставки рассматривается

как начальное приближение, а после вставки проводится поиск той позиции, в которой должен размещаться новый элемент.

Итератор **front_insert_iterator** аналогичен итератору **insert_iterator,** но всегда осуществляет вставку в первую позицию последовательности. Поэтому в конструктор итератора **front_insert_iterator** передается только один параметр — контейнер. Но нельзя отождествлять итератор **front_insert_iterator** с итератором **insert_iterator,** установленным на первую позицию последовательности. Если записывать в контейнер несколько чисел итератором **insert_iterator,** они расположатся в той последовательности, в которой записывались. Причина в том, что после каждой записи позиция итератора явно или неявно увеличивается на **1.** А запись с помощью итератора **front_insert_iterator** приведет к тому, что числа расположатся в обратной последовательности, так как позиция итератора фиксирована — всегда первая.

Рассмотрим пример. Правда, воспользоваться вектором, который фигурировал во всех предыдущих примерах, нам не удастся, так как для него итератор **front_insert_iterator** не определен. Так что возьмем для примера другой контейнер — список **list:**

```
#include <list>
#include <iterator>
...
list<int> L;
front_insert_iterator<list<int> >  I2 (L);
for(int i = 1; i <= 5; i++)
  *I2++ = i;
list<int>::iteratorI = L.begin();
while(I != L.end())
  ShowMessage(*I++);
```

В приведенном коде запись в список L осуществляется итератором **I2** типа **front_insert_iterator.** В результате последовательность чисел в списке будет такая: 5, 4, 3, 2, 1, так как каждое очередное число вставляется перед записанными ранее.

Итератор **back_insert_iterator** отличается от **front_insert_iterator** только тем, что осуществляет вставку всегда в конец последовательности. Так что если в предыдущем примере заменить объявление итератора на:

```
back_insert_iterator<list<int> >  I2(L);
```

то список заполнится последовательностью: **1,** 2, 3, 4, 5.

Осталось рассмотреть еще один итератор записи — **ostream_iterator.** Он предназначен для записи в выходной поток. В шаблон итератора передается один параметр — тип данных. А его конструктор имеет две формы:

```
ostream_iterator(ostream& s)
ostream_iterator(ostream& s, const char* delim)
```

Параметр s указывает выходной поток. Обычно, это стандартный выходной поток — экран. В первой форме конструктора s — единственный параметр. И запись через создаваемый им итератор некоторого элемента t эквивалентна выражению s >> t. Во вторую форму конструктора передается дополнительно параметр **delim** — символ разделителя, помещаемый после каждой записи. Так что запись элемента t через такой итератор эквивалентна выражению **s >> t >> delim.**

Например, операторы

```
ostream_iterator<int> Iout(cout, " ");
*Iout = Num1;
++Iout;
*Iout = Num2;
```

выводят в стандартный выходной поток значения целых переменных **Numl и Num2,** разделенные пробелом, указанным в конструкторе в качестве разделителя.

5.5.4 Итераторы, допускающие чтение и запись

Итераторы, обеспечивающие и чтение, и запись данных, представлены тремя группами: однонаправленные итераторы, двунаправленные и итераторы с произвольным доступом.

Однонаправленные итераторы не имеют каких-то дополнительных операций или возможностей по сравнению, например, с итераторами чтения. Но в них снимается ряд ограничений, свойственных итераторам чтения. Прежде всего, эти итераторы могут использоваться и для чтения, и для записи. Кроме того, несмотря на их однонаправленность от начала к концу последовательности, они могут использоваться в некоторых многопроходных алгоритмах.

В двунаправленных итераторах ко всем этим возможностям добавляются операции префиксного и постфиксного декремента. Так что эти итераторы могут перемещаться по последовательности в обоих направлениях, обеспечивая чтение и перезапись значений любых элементов. В некоторых контейнерах, например, в списке **list,** именно этот тип итератора используется как элемент **list<T>::iterator.**

Недостатком двунаправленных итераторов является возможность перемещения только на одну позицию вперед или назад за одну операцию инкремента или декремента. Этот недостаток устранен в итераторах с произвольным доступом. В них в дополнение к операциям, поддерживаемым двунаправленными итераторами, введены операции "+", "−", "+=", "− =", в которых могут участвовать итераторы и целочисленное выражения. Если целочисленное выражение имеет значение **n,** то соответственная операция означает сдвиг текущей позиции итератора на **n** позиций вперед (при n > 0) или назад (при n < 0). Введена также операция "[n]" для доступа к элементу последовательности, лежащему в позиции, отстоящей на n от текущей. Определены операции отношения меду двумя итераторами, указывающими на одну и ту же последовательность: "<", "<=", ">", ">=", возвращающие булево значение.

Таким образом, если i и j — итераторы с произвольным доступом к объектам типа X, а n — целочисленное значение со знаком, то возможны следующие выражения:

i += n	сдвиг итератора вперед (при n > 0), или назад (при n < 0) на n позиций
i + n или **n** + i	две идентичные формы, эквивалентные выражению i += n
i -= n	сдвиг итератора назад (при n > 0), или вперед (при n < 0) на n позиций; эквивалент выражения i += **(−n)**
i — **n**	эквивалент выражения i −= n
i − **j**	расстояние между i и j, указывающими на одну и ту же последовательность; возвращает такое n, при котором i = j + n
i[n]	возвращает значение типа X элемента, лежащего в позиции, отстоящей на n от текущей; текущая позиция не изменяется; эквивалент выражения *(i + n)
i[n] = t	копирование значения t типа X в элемент, лежащий в позиции, отстоящей на n от текущей; текущая позиция не изменяется; эквивалент выражения *(i + n) = t
i< j	возвращает булево значение **true,** если позиция i меньше j; выражение допустимо, если i и j указывают на одну и ту же последовательность

Итераторами с произвольным доступом являются элементы **iterator и const_ iterator** классов векторов **vector** и очередей **deque.**

Большинство операций над итераторами с произвольным доступом уже иллюстрировались в примерах разд. 5.4. Осталось, пожалуй, проиллюстрировать только операции "[n]" и "<". Ниже приведен пример работы с вектором целых чисел V, в который уже записано не менее трех элементов. Пусть мы хотим просмотреть эти данные, начиная со второй позиции и кончая предпоследней позицией последовательности, и для каждой позиции найти разность значений элементов, расположенных справа и слева от нее. Это можно сделать следующим кодом:

```
vector<int>::iterator I1;
for(I1 = V.begin() + 1; I1 < V.end() - 1; I1++)
  ShowMessage("Позиция: " + IntToStr(I1 - V.begin() + 1) +
              ", разность соседних элементов: " +
              IntToStr(I1[1] - I1[-1]));
```

Начальная позиция итератора **I1** задается равной **V.begin() + 1,** т.е. это вторая позиция последовательности. Так что это иллюстрирует применение операции "+". Конец цикла определяется операцией отношения "<", причем в записи ее правого операнда использована операция "-". В строке, выводимой функцией **ShowMessage,** номер текущей позиции рассчитывается, исходя из расстояния между **I1** и началом последовательности **V.begin()**. А для определения разности значений соседних элементов использована операция "[n]". При этом **I1[1]** — значение элемента, расположенного справа, а **I1[−1]** — значение элемента, расположенного слева.

5.6 Класс строк string

Класс строк **string** предоставляет пользователю существенно более удобные операции работы со строками, чем основной в C++ тип строк **char *.** Основное достижение, пожалуй, связано с автоматическим выделением памяти при изменении размеров строки **string,** что исключает утомительные и чреватые множеством ошибок манипуляции с памятью, присущие типу **char** *. Впрочем, на мой взгляд, класс **string** все-таки во многих отношениях менее удобен, чем класс **AnsiString,** введенный в C++Builder.

Класс **string** определен следующим образом:

```
typedef basic_string <char> string;
```

Таким образом, **string** является псевдонимом класса **basic_string.** Имеется аналогичный класс **wstring,** работающий с типом символов **wchar_t**. Классы объявлены в заголовочном модуле <string>.

Создание объекта типа **string** может быть выполнено следующим образом:

```
#include <string>.
using namespace std;
...
string S("Привет !");
```

или оператором

```
string S = "Привет !";
```

В последнем случае присваивание вызывает конструктор копии класса.

В string, в отличие от **AnsiString,** не предусмотрено приведение числовых и символьных типов. Так что операторы вида:

```
string S1(2);
string S1('a');
```

вызовут ошибку компилятора. Впрочем, присваивание символа переменной типа все-таки возможно:

```
string S2;
S2 = '!';
```

Строки типа **string,** в отличие от **char** *, не требуют завершающего нулевого символа. Они хранят свою длину вместе со значением. Текущее значение длины строки (число символов) может быть получено функцией-элементом **length.** Так для приведенного выше примера переменной S выражение **S.length()** вернет 8.

К отдельным символам строки можно получить доступ операцией **[ind].** Индексы начинаются с 0. Максимальный индекс равен **length** - 1. Например, выражение **S[0]** вернет символ "П", а выражение **S[7]** вернет "!".

Операция **[ind]** не осуществляет проверки допустимости индекса. Так что, например, результат выражения **S[8]** непредсказуем. Альтернативный способ доступа к символам — использование функции **at(ind).** Она работает так же, как операция **[ind],** но при выходе индекса за допустимые пределы (например, в случае выражения **S.at(8))** генерирует исключение.

Склеивание строк (конкатенация) осуществляется операцией "+", как и в типе **AnsiString.** При этом можно смешивать строки и символы. Например, оператор

```
S = S1 + ' ' + S2;
```

занесет в строку S строки **S1** и S2, разделенные символом пробела. Определена и операция "+=". Например, оператор

```
S += S2;
```

прибавит к содержимому строки S строку S2. Аналогичный результат можно получить с помощью функции **append:**

```
S.append(S2);
```

Для класса определены операции отношения "<", ">", "==" и другие. Так что можно использовать выражения вида:

```
if(S1 < S2) ...;
```

Сравнение проводится с учетом регистра. Латинские буквы считаются меньше букв кириллицы.

Имеется также функция-элемент **compare,** осуществляющая сравнение строк. Например, выражение

```
S1.compare(S2)
```

возвращает 0, если строки S1 и S2 эквивалентны, отрицательное число, если **S1** < S2, и положительное число, если **S1** > S2.

Это простейший вариант функции **compare.** Другая перегруженная форма этой функции позволяет сравнивать фрагмент данной строки с указанной строкой. Например, выражение

```
S1.compare(1, 3, S2)
```

сравнивает символы со второго по четвертый строки S1 со строкой S2. Первый параметр функции **compare** показывает индекс первого сравниваемого символа данной строки. Второй параметр указывает число сравниваемых символов. А третий параметр указывает строку, с которой проводится сравнение.

Имеется еще одна перегруженная форма функции **compare,** в которой фрагмент данной строки сравнивается с фрагментом другой строки. В этой форме добавляются еще четвертый и пятый параметры, указывающие соответственно начальный символ и длину фрагмента другой строки. Например, выражение

```
S1.compare(0, 4, S2, 1, 4)
```

сравнивает первые четыре символа строки S1 с символами со второго по пятый строки S2.

Функция **substr** выделяет из данной строки подстроку. Первый аргумент функции указывает индекс первого символа подстроки, а второй аргумент определяет число символов в подстроке. Например, оператор

```
S1 = S2.substr(1, 4);
```

заносит в строку **S1** подстроку из S2, начинающуюся со второго символа (индекс 1) и содержащую 4 символа.

Имеется функция **swap,** осуществляющая обмен между двумя строками. Например, оператор

```
S1.swap(S2);
```

заносит в S1 содержимое S2, а в S2 — содержимое S1.

Ряд перегруженных вариантов функции **insert** выполняет вставку строк или подстрок в указанную позицию данной строки. Например, оператор

```
S1.insert(4,S2);
```

вставляет строку S2 после четвертого символа строки **S1.** Соответственно все символы исходной строки, начиная с пятого, сдвигаются, освобождая место для вставленной строки. Имеются варианты функции **insert,** вставляющие подстроку, а не целую строку, вставляющие заданное число указанных символов и т.п.

Функция **erase** удаляет из строки заданное число символов, начиная с указанной позиции. Например, оператор

```
S1.erase(2,4);
```

удаляет из строки S1 4 символа, начиная с позиции 2 (т.е. начиная с третьего символа). Если значение второго параметра, указывающего число удаляемых символов, велико, то удалятся все символы, начиная с указанного первым параметром и до конца строки.

Имеется ряд функций, осуществляющих поиск заданной подстроки или символов. Функции **find и rfind** осуществляют поиск соответственно первого и последнего вхождения подстроки в данную строку. При успешном поиске возвращается индекс начала найденной подстроки. Если подстрока не найдена, возвращается константа **string::npos.**

Например, если заданы строки

```
string S1 = "В лесу родилась елочка, в лесу она росла.";
string S2= "лес";
```

то операторы

```
int i = S1.find(S2);
int j= S1..rfind(S2) ;
```

вернут значения i = 2, j = 26.

Пусть, например, мы хотим заменить в приведенной выше строке S первое слово "лесу" на слово "лесочке". Это может быть оформлено следующим образом:

```
string S2= "лесу", S3 = "лесочке";
int i;
if ( (i = S1.find(S2)) != string::npos)
{
 S1.erase(i, S2.length());
 S1.insert(i, S3);
}
else ShowMessage("Подстрока не найдена");
```

Функция **erase** удаляет найденную подстроку S2, а функция **insert** вставляет в ту же позицию строку S3.

Вторым аргументом в функцию **find** может быть передана позиция в данной строке, с которой надо начинать поиск. Это позволяет организовать поиск всех вхождений подстроки:

```
string S2= "лесу", S3 = "лесочке";
int i = 0;
while ( (i = S1.find(S2, i+1)) != string::npos)
{
 S1.erase(i, S2.length());
 S1.insert(i, S3);
}
```

Каждый следующий поиск в цикле начинается с позиции, следующей за найденной ранее (i + 1), т.е. осуществляется в оставшейся части строки. Поэтому приведенный код произведет замену всех вхождений подстроки S2 на **S3.**

Функции **find_first_of** и **find_last_of** осуществляют поиск соответственно первого или последнего вхождения в строку одного из заданных символов. Например, оператор

```
int i = S1.find_first_of(".,:;");
```

вернет индекс первого из символов ".", ",", " ", ":", ";" в строке **S1,** т.е. найдет окончание первого слова строки. Указанные символы перечисляются в строке, передаваемой в функцию как аргумент. Вторым аргументом функции может задаваться индекс, начиная с которого должен производиться поиск. Это позволяет так же, как в функции **find,** осуществлять в цикле поиск всех вхождений указанного множества символов.

Функции **find_first_not_of и find_last_not _of** решают противоположную задачу: находят индекс первого или последнего символа, отличного от заданных в строке аргумента.

Выше были приведены примеры контекстного поиска и замены, демонстрирующие применение функций **erase и insert.** Однако ту же контекстную замену можно сделать проще с помощью функции **replace.** В качестве первого аргумента в нее передается индекс начала заменяемой подстроки, в качестве второго аргумента указывается число символов подстроки, а третьим аргументом указывается строка, которая должна заменить удаляемую подстроку. Например, код

```
int i = 0;
while ((i = S1.find(S2, i+1)) != string::npos)
  S1.replace(i, S2.length(), S3);
```

осуществит замену всех вхождений S2 в **S1** на **S3.**

Мы рассмотрели основные функции класса **string.** Останавливаться на всех этих функциях и их перегруженных вариантах в рамках данной книги невозможно. Отметим также, что строки **string** являются обычными контейнерами STL и могут работать с итераторами так же, как другие ранее рассмотренные контейнеры. В целом, как видим, класс **string** достаточно удобен для работы. Но в реальных приложениях обычно фигурируют строки разных типов. Например, функции API Windows требуют, как правило, строки типа **char *.** А большинство строковых свойств компонентов C++Builder имеют тип **AnsiString.** Так что часто возникает необходимость приведения одних типов строк к другим.

Значение строки типа **char *** можно непосредственно присваивать переменным типа **string.** Например:

```
char * Sch = "Привет";
string S = Sch;
```

Однако значение строки типа **AnsiString** присвоить непосредственно строке типа **string** невозможно. Выходом из положения является функция-элемент **c_str()** класса **AnsiString,** которая преобразует тип **AnsiString в char** *. А строку этого типа можно присвоить переменной класса **string.** Таким образом, если требуется записать в переменные класса **string** какие-то строковые свойства компонентов VCL, использующие тип **AnsiString,** это делается операторами вида:

```
string S = (Edit1->Text).c_str();
string S1 = (Memo1->Text).c_str();
```

Аналогичная функция-элемент **c_str()** имеется и в классе string. Она преобразует строку в тип **char *.** А поскольку строки типа **char *** можно присваивать переменным типа AnsiString, то эта же функция дает возможность преобразования **string в AnsiString.** Приведем пример:

```
string S = "Привет";
ShowMessage(S.c_str());
Edit1->Text = S.c_str();
```

Второй из этих операторов использует приведение типа **string к char** *, который требуется для функции **ShowMessage.** А третий оператор присваивает значение свойству **Edit1->Text,** имеющему тип **AnsiString.**

5.7 Алгоритмы

5.7.1 Общие сведения

В STL включено свыше 100 алгоритмов и их вариантов различного назначения. Естественно, в рамках данной книги невозможно рассмотреть их хоть сколько-нибудь подробно. Надеюсь, что скоро смогу подготовить отдельную небольшую книгу по STL, в которой рассмотрю все эти алгоритмы с соответствующими примерами и методикой создания своих собственных алгоритмов. Надеюсь также включить в ближайшем будущем соответствующую справку в [2]. А пока вынужден ограничиться только перечнем алгоритмов с краткими комментариями. Некоторые примеры применения алгоритмов см. в раз. 5.8.

Алгоритмы построены так, что для них безразличны особенности реализации разных контейнеров. Они работают с итераторами, в качестве которых могут выступать итераторы любых контейнеров и даже указатели на обычные массивы. В этом и заключается универсальность принципов построения алгоритмов, принятая в STL. Впрочем, надо обращать внимание в объявлениях алгоритмов, какие типы итераторов они используют. Это определяет, к каким классам контейнеров можно применять тот или иной алгоритм. Например, если алгоритм использует итератор произвольного доступа, то этот алгоритм можно применять только к контейнерам, поддерживающим такие итераторы.

Для использования любых алгоритмов библиотеки в модуль должна быть включена директива

```
#include <algorithm>
```

В большинстве алгоритмов последовательность, с которой работает алгоритм, задается двумя итераторами: **first и last.** При этом полагается, что последовательность указана на интервале **[first, last),** т.е. рассматриваются элементы, начиная с того, на который указывает **first,** и до элемента, предшествующего позиции **last.** Если в качестве **first** указать **begin(),** а в качестве **last** указать **end(),** то будет рассматриваться все содержимое контейнера.

Многие алгоритмы имеют две модификации: первая использует для сравнения стандартную операцию отношения (обычно это операция <), а вторая указывает стандартную или введенную пользователем функцию сравнения. Применение собственной функции сравнения позволяет определить нестандартное упорядочивание элементов. В частности, это безусловно необходимо, если элементы представляют собой указатели на объекты или, например, структуры. В подобных случаях стандартная операция отношения, естественно, не работает. В разд. 5.8 приводятся сведения о стандартных функциях-объектах и создании собственных функций.

Последующие разделы группируют алгоритмы по кругу решаемых ими задач и дают краткие описания всех алгоритмов библиотеки.

5.7.2 Алгоритмы заполнения контейнеров

Алгоритм	Синтаксис / Описание
fill	**template <class ForwardIterator, class T>** **void fill(ForwardIterator first, ForwardIterator last,** **const T& value);** Заполняет контейнер в интервале **[first, last)** значениями **value**
fill_n	**template <class OutputIterator, class Size, class T>** **void fill_n(OutputIterator first, Size n, const T& value);** Заполняет n элементов контейнера, начиная с **first,** значениями **value**
generate	**template <class ForwardIterator, class Generator>** **void generate(ForwardIterator first, ForwardIterator last,** **Generator gen);** Заполняет контейнер в интервале [first, last) значениями, генерируемыми указанной функцией gen. Это функция пользователя без аргументов, возвращающая значение типа, соответствующего элементам контейнера
generate_n	**template <class OutputIterator, class Size, class Generator>** **void generate_n(OutputIterator first, Size n, Generator gen);** Заполняет n элементов контейнера, начиная с **first,** значениями, генерируемыми указанной функцией gen. Это функция пользователя без аргументов, возвращающая значение типа, соответствующего элементам контейнера

5.7.3 Алгоритмы поиска в несортированных последовательностях

Алгоритм	Синтаксис / Описание
adjacent_find	**template <class ForwardIterator> ForwardIterator** **ForwardIterator adjacent_find(ForwardIterator first,** **ForwardIterator last);** **template <class ForwardIterator, class BinaryPredicate>** **ForwardIterator adjacent_find(ForwardIterator first,** **ForwardIterator last, BinaryPredicate pred);** Возвращает итератор, указывающий на первый из двух последовательно расположенных элементов, удовлетворяющих заданному критерию. В первом варианте критерий — равенство элементов. Во втором варианте критерий определяется заданной функцией **pred.** Поиск ведется в интервале **[first, last).** Если пара элементов, удовлетворяющая критерию, не найдена, возвращается **last**
count	**template<class InputIterator, class T>** **typename iterator_traits<InputIterator>::difference_type** **count(InputIterator first, InputIterator last, const T& value);** **template <class InputIterator, class T, class Size>** **void count(InputIterator first, InputIterator last,** **const T& value, Size& n);** Подсчитывает, сколько раз в интервале **[first, last)** встречается значение **value.** Первый вариант возвращает подсчитанное значение. Второй вариант добавляет подсчитанное значение к n

Алгоритм	Синтаксис / Описание
count_if	template<class InputIterator, class Predicate> typename iterator_traits<InputIterator>::difference_type count_if(InputIterator first, InputIterator last, Predicate pred); template <class InputIterator, class Predicate, class Size> void count_if(InputIterator first, InputIterator last, Predicate pred, Size& n); Подсчитывает, сколько элементов в интервале [first, last) удовлетворяют критерию, заданному указанной функцией **pred.** Первый вариант возвращает подсчитанное значение. Второй вариант добавляет подсчитанное значение к n
find	template <class InputIterator, class T> InputIterator InputIterator find(InputIterator first, InputIterator last, const T& value); Возвращает итератор, указывающий на первый элемент в интервале [first, last), значение которого равно **value.** Если элемент не найден, возвращает **last**
find_end	template <class ForwardIterator1, class ForwardIterator2> ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2); template <class Forward Iterator1, class ForwardIterator2, class BinaryPredicate> ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred); Ищет последнее вхождение какого-то из элементов последовательности **[first2, last2) в** последовательность **[first1, last1).** Возвращает итератор, указывающий на найденное вхождения, или **last1,** если элемент не найден. В первом варианте сравниваются значения элементов последовательностей **[first2, last2)** и **[first1, last1).** Во втором варианте соответствие элементов последовательностей определяется указанной функцией **pred**
find_if	template <class InputIterator, class Predicate> InputIterator find_if(InputIterator first, InputIterator last, Predicate pred); Возвращает итератор, указывающий на первый элемент в интервале **[first, last),** значение которого соответствует критерию, заданному функцией **pred.** Если такой элемент не найден, возвращает **last**

Алгоритм	Синтаксис / Описание
find_first_of	**template <class ForwardIterator1, class ForwardIterator2>** **ForwardIterator1 find_first_of(** **ForwardIterator1 first1, ForwardIterator1 last1,** **ForwardIterator2 first2, ForwardIterator2 last2);** **template <class Forward Iterator1, class ForwardIterator2,** **class BinaryPredicate>** **ForwardIterator1 find_first_of(** **ForwardIterator1 first1, ForwardIterator1 last1,** **ForwardIterator2 first2, ForwardIterator2 last2,** **BinaryPredicate pred);** Ищет первое вхождение какого-то из элементов последовательности [first2, last2) в последовательность **[first1, last1).** Возвращает итератор, указывающий на найденное вхождения, или **last1,** если элемент не найден. В первом варианте сравниваются значения элементов последовательностей [first2, last2) и **[first1, last1).** Во втором варианте соответствие элементов последовательностей определяется указанной функцией **pred**
search	**template <class ForwardIterator1, class ForwardIterator2>** **ForwardIterator1 search(** **ForwardIterator1 first1, ForwardIterator1 last1,** **ForwardIterator2 first2, ForwardIterator2 last2);** **template <class ForwardIterator1, class ForwardIterator2,** **class BinaryPredicate>** **ForwardIterator1 search (** **ForwardIterator1 first1, ForwardIterator1 last1,** **ForwardIterator2 first2, ForwardIterator2 last2,** **BinaryPredicate binary_pred);** Ищет вхождение подпоследовательности [first2, last2) в последовательность **[first1, last1).** Возвращает итератор, указывающий на первый элемент найденного вхождения, или **last1,** если подпоследовательность не найдена. В первом варианте сравниваются значения элементов последовательностей [first2, last2) **и [first1, last1).** Во втором варианте соответствие элементов последовательностей определяется указанной функцией **pred**
search_n	**template <class ForwardIterator, class Size, class T>** **ForwardIterator search_n (ForwardIterator first,** **ForwardIterator last, Size count, const T& value);** **template <class ForwardIterator, class Size, class T,** **class BinaryPredicate>** **ForwardIterator search_n (ForwardIterator first,** **ForwardIterator last, Size count, const T& value,** **BinaryPredicate pred);** Ищет в последовательности **[first, last)** подпоследовательность, состоящую из **count** элементов с значениями, равными **value** (первый вариант) или со значениями, удовлетворяющими критерию, заданному функцией **pred** и параметром **value** (второй вариант). Возвращает итератор, указывающий на первый элемент найденной подпоследовательности, или **last1,** если подпоследовательность не найдена

5.7.4 Алгоритмы бинарного поиска в сортированных последовательностях

Алгоритм	Синтаксис / Описание
binary_search	**template <class ForwardIterator, class T>** **bool binary_search(ForwardIterator first,** **ForwardIterator last, const T& value);** **template <class ForwardIterator, class T, class Compare>** **bool binary_search(ForwardIterator first, ForwardIterator last,** **const T& value, Compare comp);** Возвращает **true,** если в интервале **[first, last)** найден элемент, равный **value** (первый вариант) или со значением, удовлетворяющим критерию, заданному функцией-объектом **comp** и параметром **value** (второй вариант)
equal_range	**template <class ForwardIterator, class T>** **pair<ForwardIterator, ForwardIterator>** **equal_range(ForwardIterator first, ForwardIterator last,** **const T& value);** **template <class ForwardIterator, class T, class Compare>** **pair<ForwardIterator, ForwardIterator>** **equal_range(ForwardIterator first, ForwardIterator last,** **const T& value, Compare comp);** Возвращает пару типа **pair** итераторов из диапазона [first, last), первый из которых указывает на первый элемент, не меньший чем **value,** а второй — на первый элемент, больший чем **value.** Иначе говоря, это диапазон, в который можно вставить новый элемент со значением **value,** не нарушая упорядоченности последовательности. В первом варианте при сравнении элемента и значения **value** используется операция <. Во втором варианте для сравнения используется критерий, заданный указанной функцией **comp**
lower_bound	**template <class ForwardIterator, class T>** **ForwardIterator lower_bound(ForwardIterator first,** **ForwardIterator last, const T& value);** **template <class ForwardIterator, class T, class Compare>** **ForwardIterator lower_bound(ForwardIterator first,** **ForwardIterator last, const T& value,** **Compare comp);** Возвращает итератор из диапазона [first, last), который указывает на первый элемент, не меньший чем **value.** Это первый из итераторов, возвращаемых алгоритмом **equal_range.** В первом варианте при сравнении элемента и значения **value** используется операция <. Во втором варианте для сравнения используется критерий, заданный указанной функцией **comp**

Алгоритм	Синтаксис / Описание
upper_bound	template <class ForwardIterator, class T> ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const T& value); template <class ForwardIterator, class T, class Compare> ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp); Возвращает итератор из диапазона [first, last), который указывает на первый элемент, больший чем **value.** Это второй из итераторов, возвращаемых алгоритмом **equal_range.** В первом варианте при сравнении элемента и значения **value** используется операция <. Во втором варианте для сравнения используется критерий, заданный указанной функцией **comp**

Комментарий

Все описанные функции имеют по два перегруженных варианта. В первом для сравнения используется операция <, причем предполагается, что с помощью этой же операции упорядочены элементы контейнера. Во втором перегруженном варианте каждой функции для сравнения используется функция-объект (см. разд. 5.8) **comp,** являющаяся бинарным предикатом. Первым ее параметром является элемент последовательности, а вторым — значение **value.** Предполагается, что последовательность в контейнере упорядочена с помощью той же функции **сотр.**

Алгоритм **lower_bound** возвращает первый из пары итераторов, возвращаемых алгоритмом **equal_range.** А алгоритм **upper_bound** возвращает второй из пары итераторов, возвращаемых алгоритмом **equal_range.** Эти алгоритмы могут использоваться для определения диапазона, в который можно вставить новый элемент со значением **value,** не нарушая упорядоченности последовательности.

5.7.5 Алгоритмы сравнения

Алгоритм	Синтаксис / Описание
equal	template <class InputIterator1, class InputIterator2> bool equal(InputIterator1 first1, InputIterator1 last 1, InputIterator2 first2); template <class InputIterator1, class InputIterator2, class BinaryPredicate> bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate binary_pred); Возвращает **true,** если все элементы в интервале **[first1, last1)** одной последовательности совпадают с соответствующими элементами другой последовательности, начинающейся с итератора **first2.** Предполагается, что во второй последовательности, начинающейся с first2, по крайней мере столько элементов, сколько в интервале **[first1, last1).** Первый вариант алгоритма использует при сравнении операцию ==. Второй вариант сравнивает элементы с помощью указанной функции binary_pred

Алгоритм	Синтаксис / Описание
lexicographical_compare	**template <class InputIterator1, class InputIterator2>** **bool lexicographical_compare(** **InputIterator1 first, InputIterator2 last1,** **InputIterator2 first2, InputIterator last2);** **template <class InputIterator1, class InputIterator2,** **class Compare>** **bool lexicographical_compare(** **InputIterator1 first, InputIterator2 last1,** **InputIterator2 first2, InputIterator last2,** **Compare comp);** Возвращает **true**, если последовательность **[first1, last1)** лексикографически меньше или равна другой последовательности **[first2, last2)**. Каждый элемент первой последовательности сравнивается с соответствующим элементом второй последовательности. Функция возвращает **true**, как только обнаруживается пара элементов, в которой элемент первой последовательности меньше, чем во второй, или если все элементы первой последовательности совпали с соответствующими элементами второй последовательности, но вторая последовательность длиннее. Возвращает **false**, как только обнаруживается пара элементов, в которой элемент второй последовательности меньше, чем в первой. В первом варианте алгоритма сравнение осуществляется операцией <. Во втором критерий сравнения определяется указанной функцией **comp**
mismatch	**template <class InputIterator1, class InputIterator2>** **pair<InputIterator1,InputIterator2> mismatch(** **InputIterator1 first1, InputIterator1 last1,** **InputIterator2 first2);** **template <class InputIterator1, class InputIterator2,** **class BinaryPredicate>** **pair<InputIterator1, Inputiterator2> mismatch(** **InputIterator first1, InputIterator1 last1,** **InputIterator2 first2,** **BinaryPredicate binary_pred);** Сравнивает элементы последовательность **[first1, last1)** с соответствующими элементами последовательности, начало которой задано итератором **first2**. Предполагается, что вторая последовательность содержит по крайней мере столько элементов, сколько имеется в первой. Функция возвращает пару итераторов, указывающих на первые несовпадающие элементы последовательностей. Если последовательности идентичны в пределах числа элементов первой последовательности, то возвращается итератор **last1** и соответствующий ему итератор второй последовательности. Первый вариант алгоритма использует при сравнении операцию эквивалентности ==. Второй вариант сравнивает элементы с помощью указанной функции **binary_pred**

5.7.6 Алгоритмы копирования

Алгоритм	Синтаксис / Описание
copy	**template <class** InputIterator, **class** OutputIterator> OutputIterator copy(InputIterator **first, InputIterator last,** OutputIterator **result);** Копирует элементы в интервале [firstl, last1) в последовательность, начинающуюся с итератора **result.** Этот итератор может относиться к другому контейнеру или к тому же контейнеру. В первом случае осуществляется копирование элементов одного контейнера в другой. Во втором часть элементов контейнера копируется в другое место того же контейнера. В этом случае полагается, что итератор **result** не лежит в интервале **[firstl, lastl).** Но он может указывать на позицию, предшествующую first, и в этом случае копия может перекрывать часть интервала **[firstl, lastl).** Ничего страшного в этом случае не произойдет, поскольку копирование начинается с первого элемента последовательности
copy_ backward	**template <class** BidirectionalIterator1, **class** BidirectionalIterator2> BidirectionalIterator2 copy_backward(BidirectionalIterator1 **first, Bidirectionallteratorl last,** BidirectionalIterator2 **result);** Копирует элементы в интервале **[firstl, lastl)** в последовательность, заканчивающуюся итератором **result.** Копирование начинается с конца, т.е. с элемента, указанного итератором **last** - 1. Итератор **result** может относиться к другому контейнеру или к тому же контейнеру. В первом случае осуществляется копирование элементов одного контейнера в другой. Во втором часть элементов контейнера копируется в другое место того же контейнера. В этом случае полагается, что итератор **result** не лежит в интервале **[firstl, lastl).** Но он может указывать на позицию, превышающую **last,** и в этом случае копия может перекрывать часть интервала **[firstl, lastl).** Ничего страшного в этом случае не произойдет, поскольку копирование начинается с последнего элемента последовательности

5.7.7 Алгоритмы преобразования последовательностей

Алгоритм	Синтаксис / Описание
partition	**template <class** BidirectionalIterator, **class Predicate>** BidirectionalIterator **partition (Bidirectionallterator first, Bidirectionallterator last, Predicate pred);** В интервале [first, last) алгоритм размещает все элементы, удовлетворяющие критерию, заданному функцией **pred,** перед элементами, не удовлетворяющими этому критерию. Возвращает итератор, указывающий на первый элемент второй группы, т.е. группы элементов, не удовлетворяющих критерию **pred.** Алгоритм не гарантирует сохранение относительного расположения элементов каждой группы. При необходимость сохранять последовательность элементов надо использовать алгоритм stable_partition

Алгоритм	Синтаксис / Описание
random_shuffle"	template <class RandomAccessIterator> void random_shuffle(RandomAccessIterator first, RandomAccessIterator last); template <class RandomAccessIterator, class RandomNumberGenerator> void random_shuffle (RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& rand); Перетасовывает элементы в интервале **[first, last),** используя для выбора позиций элементов равномерно распределенные целые случайные числа. Второй вариант использует функцию-объект генератора случайных чисел, указанную аргументом **rand**
replace	template <class ForwardIterator, class T> void replace (ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value); В интервале [first, last) замещает элементы, имеющие значение **old_value,** элементами со значением **new_value**
replace_copy	template <class InputIterator, class OutputIterator, class T> OutputIterator replace_copy(InputIterator first, InputIterator last, OutputIterator result, const T& old_value, const T& new_value); Копирует элементы в интервале **[first, last)** в последовательность, начинающуюся с **result,** присваивая при этом значение **new_value** копиям элементов, имеющих в исходной последовательности значение **old_value.** Возвращает итератор **result+(last–first),** т.е. указывающий на позицию, следующую за последней копией
replace_copy_if	template <class InputIterator, class OutputIterator, class Predicate, class T> OutputIterator replace_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred, const T& new_value); Копирует элементы в интервале **[first, last)** в последовательность, начинающуюся с **result,** присваивая при этом значение **new_value** копиям элементов, удовлетворяющих критерию **pred.** Возвращает итератор **result+(last–first),** т.е. указывающий на позицию, следующую за последней копией
replace_if	template <class ForwardIterator, class Predicate, class T> void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred const T& new_value); Замещает в интервале **[first, last)** значения элементов, удовлетворяющих критерию **pred,** значением **new_value**
reverse	template <class BidirectionalIterator> void reverse(BidirectionalIterator first, BidirectionalIterator last); Изменяет последовательность элементов в интервале **[first, last) на обратную**

Алгоритм	Синтаксис / Описание
reverse_ copy	**template <class BidirectionalIterator, class OutputIterator> OutputIterator reverse_copy (Bidirectionallterator first, Bidirectionallterator last, Outputlterator result);** Копирует элементы в интервале [first, last) в последовательность, начинающуюся с **result,** изменяя при этом последовательность элементов на обратную
rotate	**template <class ForwardIterator> void rotate (ForwardIterator first, ForwardIterator middle, ForwardIterator last);** Циклически сдвигает последовательность **[first, last)** на число позиций **middle - first.** В итоге обменяются местами элементы последовательности, расположенные в интервале [first, **middle),** с элементами в интервале **[middle, last).** Например, если исходная последовательность {1, 2, 3, 4, 5} и **middle = first** + 2 (указывает на элемент 3), то результирующая последовательность: {3, 4, 5, 1, 2}. Предполагается, что **middle** указывает на внутреннюю точку интервала **[first, last)**
rotate_ copy	**template <class Forwardlterator, class OutputIterator> Outputlterator rotate_copy (Forwardlterator first, Forwardlterator middle, Forwardlterator last, Outputlterator result);** Копирует в последовательность, начинающуюся с **result,** сначала элементы в интервале **[middle, last),** а затем элементы в интервале **[first, middle).** Это эквивалентно циклическому сдвигу копии последовательности **[first, last) на middle - first** позиций. Например, если исходная последовательность {1, 2, 3, 4, 5} и **middle = first** + 2 (указывает на элемент 3), то копия: {3, 4, 5, 1, 2}. Предполагается, что **middle** указывает на внутреннюю точку интервала [first, last)
stable_ partition	**template <class Bidirectionallterator, class Predicate> Bidirectionallterator stable_partition(BidirectionalIterator first, Bidirectionallterator last, Predicate pred);** В интервале **[first, last)** размещает все элементы, удовлетворяющие критерию, заданному функцией **pred,** перед элементами, не удовлетворяющими этому критерию. Возвращает итератор, указывающий на первый элемент второй группы, т.е. группы элементов, не удовлетворяющих критерию **pred.** Алгоритм сохраняет относительное расположение элементов внутри каждой группы, в отличие от более быстрого алгоритма **partition**
swap	**template <class T> void swap (T& a, T& b);** Обменивает друг с другом значения элементов a и b
swap_ ranges	**template <class ForwardIterator1, class ForwardIterator2> ForwardIterator2 swap_ranges (Forwardlteratorl firstl, Forwardlteratorl last1, ForwardIterator2 first2);** Обменивает друг с другом значения соответствующих элементов последовательности **[firstl, lastl)** и последовательности, начинающейся с **first2**

Алгоритм	Синтаксис / Описание
transform	**template <class InputIterator, class OutputIterator, class UnaryOperation> OutputIterator transform (InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op);** **template <class InputIterator1, class InputIterator2, class OutputIterator, class BinaryOperation> OutputIterator transform (InputIteratorl firstl, InputIteratorl last1, InputIterator2 first2, OutputIterator result, BinaryOperation binary_op);** Первый вариант алгоритма применяет ко всем элементам в интервале **[first,** last) унарную операцию op (например, вычисляющую квадрат значения, или умножающую значение на константу) и помещает результирующее значение каждого элемента в выходную последовательность, начинающуюся с **result.** Если **result** совпадает со входным итератором, то производится замена элементов исходной последовательности. Второй вариант алгоритма применяет ко всем элементам в интервале **[firstl, lastl)** и соответствующим элементам последовательности, начинающейся с **first2** бинарную операцию **binary_op** (например, складывает или перемножает значения элементов) и помещает результирующее значение каждого элемента в выходную последовательность, начинающуюся с **result.** Если **result** совпадает с итератором первой или второй последовательности, то производится замена элементов соответствующей последовательности

5.7.8 Алгоритмы сканирования

Алгоритм	Синтаксис / Описание
accumulate	**template <class InputIterator, class T> T accumulate (InputIterator first, InputIterator last, T init);** **template <class InputIterator, class T, class BinaryOperation> T accumulate (InputIterator first, InputIterator last, T init, BinaryOperation binary_op);** Первый вариант алгоритма просматривает поочередно каждый элемент в интервале **[first, last)** и возвращает начальное значение **init** плюс сумму значений элементов. Так что если задать **init** = 0, то результат равен сумме значений всех элементов. Второй вариант алгоритма применяет к каждому элементу и текущему значению **init** бинарную операцию binary_op. Например, если задать **init** = 1 и применить операцию умножения **multiplies,** то результат равен произведению значений всех элементов
for_each	**template <class InputIterator, class Function> void for_each(InputIterator first, InputIterator last, Function f);** В интервале **[first, last)** алгоритм поочередно применяет функцию f к каждому элементу. Поскольку итераторы последовательности не допускают изменение значений элементов, то функция может только как-то обрабатывать значения аргументов: накапливать их сумму, сумму квадратов, распечатывать значения и т.п.

Комментарий

Для использования алгоритма **accumulate** в модуль надо включить директиву:

```
#include <numeric>
```

Пример использования вы найдете в разд. 5.8.

5.7.9 Алгоритмы удаления элементов

Алгоритм	Синтаксис / Описание
remove	**template <class ForwardIterator, class T>** **ForwardIterator remove(ForwardIterator first,** **ForwardIterator last, const T& value);** Удаляет в интервале **[first,** last) элементы, значения которых равны **value.** Последующие элементы перемещаются, заполняя позиции удаленных. Но размер контейнера не изменяется. Последние позиции остаются пустыми. Алгоритм возвращает итератор, указывающий на позицию, следующую за последним оставшимся элементом. Это можно использовать, если требуется устранить пустые позиции следующим образом: container.erase(remove(first,last,value),container.end());
remove_copy	**template <class InputIterator, class OutputIterator, class T>** **OutputIterator remove_copy(InputIterator first,** **InputIterator last, OutputIterator result, const T& value);** Копирует в последовательность **result** все элементы в интервале **[first, last),** кроме тех, значения которых равны **value.** Возвращает итератор, указывающий конец результирующей последовательности
remove_ copy_if	**template <class InputIterator, class OutputIterator,** **class Predicate>** **OutputIterator remove_copy_if(InputIterator first,** **InputIterator last, OutputIterator result, Predicate pred);** Копирует в последовательность **result** все элементы в интервале **[first, last),** кроме тех, для которых выполняется критерий **pred.** Возвращает итератор, указывающий конец результирующей последовательности
remove_if	**template <class ForwardIterator, class Predicate>** **ForwardIterator remove_if(ForwardIterator first,** **ForwardIterator last, Predicate pred);** Удаляет в интервале **[first, last)** элементы, значения которых удовлетворяют критерию **pred.** Последующие элементы перемещаются, заполняя позиции удаленных. Но размер контейнера не изменяется. Последние позиции остаются пустыми. Алгоритм возвращает итератор, указывающий на позицию, следующую за последним оставшимся элементом. Это можно использовать, если требуется устранить пустые позиции следующим образом: container.erase(remove_if(first,last,pred),container.end());

Алгоритм	Синтаксис / Описание
unique	**template <class ForwardIterator>** **ForwardIterator unique(ForwardIterator first,** **ForwardIterator last);** **template <class ForwardIterator, class BinaryPredicate>** **ForwardIterator unique(ForwardIterator first,** **ForwardIterator last, BinaryPredicate binary_pred);** Просматривает в интервале [first, last) все элементы, и если встречаются два расположенных подряд элемента со значениями, равными (в первом варианте) или удовлетворяющими критерию **binary_pred** (во втором варианте), то второй элемент удаляется. Таким образом, из группы расположенных подряд эквивалентных элементов остается только первый. Алгоритм возвращает итератор, указывающий на позицию, следующую за последним оставшимся элементом
unique_copy	**template <class InputIterator, class OutputIterator>** **OutputIterator unique_copy(InputIterator first,** **InputIterator last, OutputIterator result);** **template <class InputIterator, class OutputIterator,** **class BinaryPredicate>** **OutputIterator unique_copy(InputIterator first,** **InputIterator last, OutputIterator result,** **BinaryPredicate binary_pred);** Просматривает в интервале **[first, last)** все элементы и копирует их в выходную последовательность **result.** Если встречается группа расположенных подряд элементов со значениями, равными (в пер- , вом варианте) или удовлетворяющими критерию **binary_pred** (во втором варианте), то копируется только первый элемент. Алгоритм возвращает итератор, указывающий на позицию, следующую за последним элементом выходной последовательности

5.7.10 Алгоритмы сортировки

Алгоритм	Синтаксис / Описание
inplace_ merge	**template <class BidirectionalIterator>** **void inplace_merge(BidirectionalIterator first,** **BidirectionalIteratormiddle, •** **BidirectionalIterator last);** **template <class Bidirectionallterator, class Compare>** **void inplace_merge(BidirectionalIterator first,** **Bidirectionallterator middle,** **Bidirectionallterator last, Compare comp);** Объединяет две сортированные последовательности **[first, middle) и [middle, last)** и помещает результат в **[first, last).** Если в первой и второй последовательностях есть одинаковые элементы, то в результирующей последовательности элементы первой последовательности будут предшествовать соответствующим элементам второй последовательности. В первом варианте алгоритма при объединении используется операция <, а во втором — функция сравнения **comp**

Алгоритм	Синтаксис / Описание
merge	**template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator merge(InputIterator firstl, InputIteratorl last1, InputIterator2 first2, InputIterator last2, OutputIterator result);** **template <class InputIteratorl, class InputIterator2, class OutputIterator, class Compare> OutputIterator merge(InputIterator1 firstl, InputIteratorl lastl, InputIterator2 first 2, InputIterator last2, OutputIterator result, Compare comp);** Объединяет две сортированные последовательности **[firstl, lastl) и [first2, last2)** и помещает результат в последовательность, начинающуюся с **result.** Если в первой и второй последовательностях есть одинаковые элементы, то в результирующей последовательности элементы первой последовательности будут предшествовать соответствующим элементам второй последовательности. В первом варианте алгоритма при объединении используется операция <, а во втором — функция сравнения **comp**
nth_element	**template <class RandomAccessIterator> void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);** **template <class RandomAccessIterator, class Compare> void nth_element (RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);** Разделяет все элементы в интервале **[first, last)** на две группы: сначала располагаются элементы, значения которых меньше того, на который указывает **nth,** затем располагается этот граничный элемент, а затем располагаются элементы, значения которых больше того, на который указывает nth. Внутри каждой из групп элементы не упорядочиваются. В первом варианте алгоритма при сортировке используется операция >, а во втором — функция сравнения **comp**
partial_sort	**template <class RandomAccessIterator> void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);** **template <class RandomAccessIterator, class Compare> void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp) ;** Проводит частичную сортировку: только **middle - first** элементов из всех элементов в интервале **[first, last).** В итоге первые **[middle, last)** элементов оказываются сортированными так, как если бы сортировалась вся последовательность, а остальные элементы остаются несортированными. Если задать **middle = last,** будет отсортирована вся последовательность. В первом варианте алгоритма при сравнении используется операция <, а во втором — функция сравнения **comp .**

Алгоритм	Синтаксис / Описание
partial_sort_copy	**template <class InputIterator, class RandomAccessIterator>** **void partial_sort_copy (InputIterator first, InputIterator last,** **RandomAccessIterator result_first,** **RandomAccessIterator result_last);** **template <class InputIterator, class RandomAccessIterator,** **class Compare>** **void partial_sort_copy (InputIterator first, InputIterator last,** **RandomAccessIterator result_first,** **RandomAccessIterator result_last,** **Compare comp);** Проводит частичную сортировку, эквивалентную следующей процедуре: элементы из интервала [first, last) помещаются в буфер, там сортируются и затем столько первых элементов, сколько может поместиться в интервале [result_first, result_last), помещаются в выходную последовательность, начинающуюся с **result_first**. В действительности, конечно, все делается не так, но итог тот же: в выходной последовательности оказываются n первых элементов сортированной исходной последовательности, где n = min{ **last - first, result_last - result_first** }. В первом варианте алгоритма при сравнении используется операция <, а во втором — функция сравнения **comp**
sort	**template <class RandomAccessIterator>** **void sort (RandomAccessIterator first,** **RandomAccessIterator last);** **template <class RandomAccessIterator, class Compare>** **void sort (RandomAccessIterator first,** **RandomAccessIterator last,** **Compare comp);** Проводит сортировку элементов из интервала [first, last). В первом варианте алгоритма при сортировке используется операция < и последовательность упорядочивается в порядке увеличения значений элементов, а во втором сортировка проводится с помощью функции сравнения **сотр.** При сортировке не гарантируется сохранение последовательности элементов с равными значениями. Эффективность может оказаться ниже, чем в алгоритме **stable_sort**
stable_sort	**template <class RandomAccessIterator>** **void stable_sort(RandomAccessIterator first,** **RandomAccessIterator last);** **template <class RandomAccessIterator, class Compare>** **void stable_sort (RandomAccessIterator first,** **RandomAccessIterator last, Compare comp);** Проводит сортировку элементов из интервала [first, last). В первом варианте алгоритма при сортировке используется операция <, а во втором — функция сравнения **сотр.** Гарантируется сохранение последовательности элементов с равными значениями. Эффективность при достаточном объеме памяти может оказаться выше, чем в алгоритме **sort**

5.7.11 Операции с множествами

Алгоритм	Синтаксис / Описание
includes	template <class InputIterator1, class InputIterator2> bool includes (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2); template <class InputIterator1, class InputIterator2, class Compare> bool includes (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp); Возвращает **true**, если каждый элемент сортированного множества **[first2,** last2) содержится в сортированном множестве **[first1, last1).** В первом варианте алгоритма при предварительной сортировке и сравнении множеств используется операция <, а во втором — функция сравнения **comp**
set_difference	template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result); template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp); Формирует в последовательности, начинающейся с **result,** сортированную разность двух сортированных множеств **[first1, last1)** и **[first2,** last2). Она содержит элементы, входящие в **[first1, last1),** но не входящие в [first2, last2). Алгоритм возвращает итератор, указывающий конец сформированной последовательности. В первом варианте алгоритма при предварительной сортировке и сравнении множеств используется операция <, а во втором — функция сравнения **comp**
set_intersection	template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator last2, OutputIterator result); template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp); Формирует в последовательности, начинающейся с **result,** сортированное пересечение двух сортированных множеств **[first1, last1)** и **[first2,** last2). Оно содержит элементы, входящие и в **[first1, last1),** и **в [first2,** last2), причем в результат копируются элементы первого множества. Алгоритм возвращает итератор, указывающий конец сформированной последовательности. В первом варианте алгоритма при предварительной сортировке и сравнении множеств используется операция <, а во втором — функция сравнения **comp**

Алгоритм	Синтаксис / Описание
set_ symmetric_ difference	**template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_symmetric_difference (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);** **template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator set_symmetric_difference (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);** Формирует в последовательности, начинающейся с **result,** сортированное множество, в которое входят элементы множества **[first1, last1),** отсутствующие в множестве [first2, **last2),** плюс элементы множества [first2, **last2),** отсутствующие в **[first1, last1).** Алгоритм возвращает итератор, указывающий конец сформированной последовательности. В первом варианте алгоритма при предварительной сортировке и сравнении множеств используется операция <, а во втором — функция сравнения **comp**
set_union	**template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_union (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);** **template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator set_union (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);** Формирует сортированное объединение двух множеств **[first1, last1) и [first2, last2).** Результат заносится в **result.** В первом варианте алгоритма при сравнении используется операция <, а во втором — функция сравнения **comp.** Результатом объединения является множество, содержащее все элементы первого, плюс элементы второго, отсутствующие в первом. В этом отличие от объединения двух множеств алгоритмом **merge,** который добавляет к элементам первого множества все элементы второго. Отличие от классического определения объединения множеств состоит в том, что если какой-то элемент входит в первое множество n раз, а во второе **m** раз, то в результат этот элемент войдет max(n, **m)** раз

5.7.12 Операции с кучей (heap)

Алгоритм	Синтаксис / Описание
makc_heap	**template <class RandomAccessIterator>** **void make_heap(RandomAccessIterator first,** **RandomAccessIterator last);** **template <class RandomAccessIterator, class Compare>** **void make_heap(RandomAccessIterator first,** **RandomAccessIterator last, Compare comp);** Создает кучу из последовательности **[first, last),** перемещая элемент с максимальным значением на первую позицию. Расположение остальных элементов неопределенно. В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — функция сравнения **comp**
pop_heap	**template <class RandomAccessIterator>** **void pop_heap(RandomAccessIterator first,** **RandomAccessIterator last);** **template <class RandomAccessIterator, class Compare>** **void pop_heap(RandomAccessIterator first,** **RandomAccessIterator last,** **Compare comp);** Выталкивает из кучи **[first,** last) первый элемент с наибольшим значением, перемещая его в позицию **last** - 1 и создавая кучу из элементов в интервале **[first, last - 1).** В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — функция сравнения **comp**
push_heap	**template <class RandomAccessIterator>** **void push_heap(RandomAccessIterator first,** **RandomAccessIterator last);** **template <class RandomAccessIterator, class Compare>** **void push_heap(RandomAccessIterator first,** **RandomAccessIterator last,** **Compare comp);** Добавляет элемент, расположенный в позиции **last - 1, в** кучу, сформированную в интервале **[first, last** - 1), превращая таким образом всю последовательность **[first, last)** в кучу. Добавление элемента в кучу означает, что если он больше первого элемента кучи, то перемещается в первую позицию, а в противном случае остается на последней позиции. В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — функция сравнения **comp**

Алгоритм	Синтаксис / Описание
sort_heap	template <class RandomAccessIterator> void sort_heap(RandomAccessIterator first, RandomAccessIterator last); template <class RandomAccessIterator, class Compare> void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp); Сортирует кучу [first, last), превращая ее в последовательность, упорядоченную в порядке убывания элементов. В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — функция сравнения comp

Комментарий

Кучей (heap) в данном контексте называется такая организация последовательности, в которой первым расположен элемент с наибольшим значением. Алгоритм **make_heap** создает кучу из указанной последовательности. Алгоритм **pop_heap** выталкивает из кучи первый элемент (с наибольшим значением), перемещая его на последнее место и превращая последовательность остальных элементов (их уже на 1 меньше) опять в кучу. Последовательное применение **pop_heap** ко все уменьшающейся куче приводит, в конце концов, к упорядочиванию всей последовательности в порядке убывания элементов. Алгоритм **push_heap** включает новый элемент в кучу. Перед применением этого алгоритма надо добавить новый элемент в последовательность, например, функцией-элементом контейнера **pop_back.** Алгоритм **sort_heap** сортирует кучу, превращая ее в последовательность, упорядоченную в порядке убывания элементов.

5.7.13 Алгоритмы определения минимума и максимума

Алгоритм	Синтаксис / Описание

max	template <class T>
	const T& max(const T&, const T&);
	
	template <class T, class Compare>
	const T& max(const T&, const T&, Compare);
	Возвращает максимальное из двух значений. В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — указанная функция сравнения

max_element	template <class ForwardIterator> ForwardIterator max_element(ForwardIterator first, ForwardIterator last); template <class Forwardlterator, class Compare> Forwardlterator max_element(ForwardIterator first, Forwardlterator last, Compare comp); Возвращает итератор, указывающий на элемент с наибольшим значением в последовательности [first, last). Если несколько элементов имеют одинаковое наибольшее значение, возвращается итератор, указывающий на первый из них. В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — указанная функция сравнения

Алгоритм	Синтаксис / Описание
min	**template <class T>** **const T& min(const T&, const T&);** **template <class T, class Compare>** **const T& min(const T& a, const T&, Compare);** Возвращает минимальное из двух значений. В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — указанная функция сравнения
min_element	**template <class ForwardIterator>** **ForwardIterator min_element(ForwardIterator first, ForwardIterator last);** **template <class ForwardIterator, class Compare>** **ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Compare comp);** Возвращает итератор, указывающий на элемент с наименьшим значением в последовательности **[first,** last). Если несколько элементов имеют одинаковое наименьшее значение, возвращается итератор, указывающий на первый из них. В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — указанная функция сравнения

5.7.14 Генераторы перестановок

Алгоритм	Синтаксис / Описание
next_ permutation	**template <class BidirectionalIterator>** **bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);** **template <class Bidirectionallterator, class Compare>** **bool next_permutation(BidirectionalIterator first, Bidirectionallterator last, Compare comp);** Пытается сделать следующую (см. комментарий) перестановку последовательности [first, last). Если следующая перестановка существует, она осуществляется и алгоритм возвращает **true.** Если следующей перестановки нет, делается первая перестановка и возвращается **false.** В первом варианте алгоритма для сравнения элементов используется операция <, а во втором — указанная функция сравнения
prev_ permutation	**template <class BidirectionalIterator>** **bool prev_permutation(BidirectionalIterator first, Bidirectionallterator last);** **template <class Bidirectionallterator, class Compare>** **bool prev_permutation(BidirectionalIterator first, Bidirectionallterator last, Compare comp);** Пытается сделать предыдущую (см. комментарий) перестановку последовательности **[first, last).** Если предыдущая перестановка существует, она осуществляется и алгоритм возвращает **true.** Если предыдущей перестановки нет, делается последняя перестановка и возвращается **false. В** первом варианте алгоритма для сравнения элементов используется операция <, а во втором — указанная функция сравнения

Комментарий

Генераторы перестановок предполагают наличие последовательности без дублей, например, {1 2 3}. Множество всех перестановок упорядочивается в лексикографической последовательности, т.е. результат каждой следующей перестановки больше предыдущей. Для трех элементов это дает следующую последовательность перестановок: {1 2 3}, {1 3 2}, {2 1 3}, {2 3 1}, {3 1 2}, {3 2 1}. Таким образом, первая перестановка соответствует упорядочиванию элементов в порядке нарастания, а последняя — упорядочиванию элементов в порядке убывания. Алгоритм **next_permutation** анализирует текущее расположение элементов и делает следующую перестановку. А если следующей перестановки нет (элементы расположены в убывающей последовательности, т.е. соответствуют последней перестановке), то осуществляется первая перестановка — элементы располагаются в порядке возрастания. Аналогично работает алгоритм **prev_permutation**, осуществляя предыдущую перестановку или, если предыдущей нет, осуществляя последнюю перестановку — элементы располагаются в порядке убывания.

5.8 Функции-объекты

В разд. *5.7* вы можете увидеть, что многие алгоритмы допускают указание функций, используемых для сравнения каких-то значений или каких-то вычислений. В качестве таких функций в STL используются *функции-объекты*.

Функция-объект — это объект, содержащий операцию **operator ().** К такой функции можно получить доступ и через указатель на функцию, и как к объекту с операцией **operator ().** Структура функций-объектов разработана так, чтобы обеспечить высокую эффективность использующих их алгоритмов.

Унарные функции-объекты принимают один аргумент, а бинарные функции-объекты принимают два аргумента. Базовые классы шаблонов унарных и бинарных функций-объектов объявлены следующим образом:

```
template <class Arg, class Result>
struct unary_function {
  typedef Arg argument type;
  typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};
```

Как видно, шаблоны унарных функций включают объявление двух типов: **argument_typ** — тип аргумента, и **result_type** — тип результата. Шаблоны бинарных функций включают объявление трех типов: **first_argument_type и second_argument_type** — типы аргументов, **result_type** — тип результата.

Ниже приведен список стандартных функций-объектов, включенных в библиотеку. В их описании x — первый аргумент, y — второй аргумент (для бинарных функций). Функции объявлены в заголовочном файле <functional>.

Арифметические функции

Функция	Результат
plus	**сложение x + y**
minus	**вычитание x - y**

Функция	Результат
multiplies	умножение x * y
divides	деление x / y
modulus	остаток целочисленного деления x % y
negate	изменение знака — x

Аргументы во всех функциях имеют одинаковый тип. Тип результата во всех функциях совпадает с типом аргументов. Так что во все шаблоны требуется передавать всего один тип. Например, объявление функции-объекта **multiplies** имеет вид:

```
template<class T>
struct multiplies: binary_function<T, T, T> {
        T operator() (const T&, const T&) const;
};
```

По такому же принципу строятся объявления и других функций. В качестве примера рассмотрим Применение функции **multiplies** для вычисления произведения элементов массива целых чисел с помощью алгоритма **accumulate** (см. разд. 5.7.8):

```
#include <numeric>
#include<functional>
using namespace std;

int a[5] = {1, 2, 3, 4, 5};
int mult = accumulate(a, a+5, 1, multiplies<int>());
```

В результате выполнения алгоритма **accumulate** переменная **mult** получит значение 120.

Функции сравнения

Функция	Результат равен true, если ...
equal_to	x == y
not_equal_to	x != y
greater	x > y
less	x < y
ht greater_equal	x >= y
less_equal	x <= y

Аргументы во всех функциях имеют одинаковый тип. Тип результата во всех функциях bool. Так что во все шаблоны требуется передавать всего один тип — тип аргументов. Например, объявление функции-объекта **greater** имеет вид:

```
template <class T>
 struct greater : binary_function<T, T, bool> {
          bool operator() (const T&, const T&) const;
};
```

По такому же принципу строятся объявления и других функций.

В качестве примера рассмотрим применение функции **greater** для упорядочивания массива целых чисел в порядке убывания значений элементов с помощью алгоритма **sort** (см. разд. 5.7.10). По умолчанию этот алгоритм использует функцию **less** и упорядочивает последовательность в порядке нарастания значений элементов. Но следующий код изменяет функцию сравнения и обеспечивает упорядочивание в порядке уменьшения значений элементов:

```
#include <numeric>
#include<functional>
using namespace std;

int a[5] = {3, 2, 1, 5, 4};
sort(a, a+5, greater<int>());
```

В результате выполнения алгоритма **sort** с функцией greater элементы в массиве располагаются следующим образом: {5, 4, 3, 2, 1}.

Логические функции

Функция	Результат
logical_and	логическое И x && y
logical_or	логическое ИЛИ x ‖ y
logical_not	логическое отрицание ! x

Аргументы во всех функциях имеют одинаковый тип. Тип результата во всех функциях **bool**. Так что во все шаблоны требуется передавать всего один тип — тип аргументов. Например, объявление функции-объекта logical_and имеет вид:

```
template <class T>
 struct logical_and : binary_function<T, T, bool> {
        bool operator() (const T&, const T&) const;
};
```

По такому же принципу строятся объявления и других функций. Поскольку функции производят логические операции над передаваемыми в них аргументами, то практически всегда в шаблоны функций передается тип bool (иногда int).

Рассмотрим следующий пример:

```
bool l1[4] = {true, false, false, true);
bool l2[4] = {false, false, true, true};
transform(l1, l1 + 3, l2, l1, logical_and<bool>());
```

Имеется два массива булевых значений 11 и 12. 6 результате применения алгоритма transform (см. разд. 5.7.7) в массив И заносятся значения, соответствующие применению логической операции И к начальным значениям элементов этих массивов. В данном примере элементы массива 11 получат значения {false, false, false, true}.

В заключение данного раздела остановимся коротко на создании собственных функций-объектов. Они строятся по тому же принципу, который вы видели выше в объявлениях стандартных функций. Один пример создания собственной функции приведен в разд. 5.4.6.2. Там эта функция была ориентирована только на целые числа и поэтому реализовывалась не в виде шаблона. Теперь рассмотрим похожий пример шаблонной реализации функции.

Пусть мы хотим создать функцию сравнения двух числовых значений, которая позволяла бы реализовать следующее упорядочивание последовательности: сначала должны располагаться положительные числа, за ними отрицательные, а внутри отрицательных и положительных чисел элементы должны располагаться в порядке нарастания. Для упрощения кода предположим, что элементы не могут иметь нулевое значение.

Описание такой функции сравнения может иметь вид:

```
#include <math.hpp>

template <class T>
 struct mycomp: binary_function<T, T, bool> {
  bool operator() (const T& x, const T& y) const
```

```
  {
   if(Sign(x) == Sign(y)) return (x < y);
   else return (x >= 0);
  }
};
```

Рассмотрим это описание. Функция сравнения должна принимать два аргумента, которыми являются значения двух элементов последовательности. Функция должна возвращать **true**, если первый элемент надо разместить в последовательности ранее второго, т.е. если он должен считаться меньше второго. Так что вашу функцию сравнения надо объявить как бинарную с некоторым типом T для обоих аргументов и типом **bool** результата. Тип T должен передаваться как параметр шаблона, чтобы пользователь мог использовать эту функцию как для целых, так и для действительных чисел.

Заголовок шаблона

```
template <class T>
```

обеспечивает передачу в функцию типа T, а заголовок структуры

```
struct mycomp: binary_function<T, T, bool>
```

определяет, что эта функция-объект (ей дано имя mycomp) является бинарной с параметрами типа T и результатом типа **bool**.

Реализация функции, вероятно, особых пояснений не требует. В ней используется библиотечная функция определения знака Sign, для которой необходимо подключить к модулю заголовочный файл *math.hpp*. А далее все просто. Если знаки аргументов совпадают, то возвращается **true**, если первый аргумент меньше второго. А если знаки различны, то возвращается true, если первый аргумент положительный.

Описав в своем модуле подобный шаблон, вы можете, например, выполнить следующий код:

```
int a[5] = {3, -10, -2, 5, 4};
double b[5] = {-4.5, 6, 5.1, 0.5, -10};
sort(a, a+5, mycomp<int>());
sort(b, b+5, mycomp<double>());
```

В результате его выполнения массивы a и b приобретут вид: a = {3, 4, 5, -10, −2}, b = {0.5, 5.1, 6, -10, −4.5}.

Предметный указатель

Ниже приведены ссылки на те разделы книги, в которых обсуждаются те или иные понятия. Из списка исключены функции, описания которых приведены в гл. 4, так как в этой главе разделы расположены в алфавитном порядке и требуемые функции легко найти, не обращаясь к данному предметному указателю. Включены только те функции, описание которых затерялось где-то внутри текста. Не включены в список также многие понятия, вынесенные в заголовки разделов, так как их тоже нетрудно найти в тексте книги.

Дополнительные источники информации о C++ и C++Builder 6

Ниже приведены сведения о некоторых книгах и иных информационных материалах автора по C++ и C++Builder. Оперативную информацию о готовящихся к выпуску и вышедших книгах вы можете найти на сайте автора **http://delci.h1.ru** и сайте издательства **www.binom-press.ru.**

1. Архангельский А. Я. Программирование в C++BuiIder 6 — М: ЗАО «Издательство БИНОМ», 2002

Книга содержит методические и, частично, справочные материалы по C++Builder 6 и предшествующим версиям C++Builder 5 и 4. Рассмотрены такие возможности C++Builder, как построение кросс-платформенных приложений, технологии доступа к данным ADO, InterBase Express, dbExpress, компоненты-серверы COM, технологии распределенных приложений COM, CORBA, MIDAS, новая методика диспетчеризации действий. Дается методика построения прикладных программ, реализующих текстовые и графические редакторы, мультипликацию и мультимедиа, работу с базами данных, создание отчетов, приложений для Интернет, распределенных приложений, клиентов и серверов.

Справочная часть книги содержит некоторые материалы по языку, функциям, типам и классам C++Builder, но, конечно, сведения по языку и функциям приводятся в значительно меньшем объеме, чем в данной книге. Зато, там дается методика построения приложений самого разного назначения, рассказывается о техники связи с базами данных, о построении распределенных приложений и т.п. Конечно, та и данная книги частично перекликаются, поскольку каждая из них должна быть самодостаточной. Но, мне кажется, что они в значительной степени дополняют друг друга. Конечно, хорошо бы было объединить их в одну и совсем избежать повторов. Но книгу такого объема (порядка полутора тысяч страниц) технически невозможно издать, и пользоваться ею тоже было бы невозможно — *уж* очень она была бы увесистой.

2. Серия справочных файлов «Русские справки по C++Builder»

Серия справок — это программный продукт, призванный оказать вам поддержку в процессе проектирования. Справки встраиваются в среду C++Builder командой Help | Customize в дополнение к англоязычной справке и в процессе проектирования при нажатии клавиши F1 вам предлагаются на выбор темы английских или русских справок. Русские справки — это не перевод с английского, а, скорее, расширенный электронный вариант материалов данной книги и книги [1]. Так что они могут быть полезны не только тем, кто испытывает определенные сложности

с английским, но и всем пользователям C++Builder, поскольку содержат иначе построенное и скомпонованное изложение справочных данных, иные примеры, в них устранен ряд ошибок англоязычных справок (надеюсь, не добавлено собственных ошибок). Честно говоря, в них значительно больше справочных сведений, чем в данной книге.

В настоящее время серия включает в себя три справки: по C++ в C++Builder, по компонентам и классам C++Builder, по графикам и диаграммам TeeChart. Число входов предметного указателя справок около 3000, а число страниц текста в три раза превышает объем данной книги. Намечен также выпуск дополнительных справок по стандартной библиотеке STL, по Интернет, по методике проектирования, по развернутым и прокомментированным примерам и ряд других.

Достоинство справок по сравнению с книгами в том, что они обеспечивают оперативную помощь в среде разработки C++Builder, облегчают поиск нужной информации (в книгах это делать значительно сложнее), позволяют легко переносить примеры в свой проект. Да и стоят справки намного дешевле, чем книги. Но, конечно, справки не заменяют книг, хотя и содержат много материала, не поместившегося в книги.

Справки распространяются через Интернет по адресу: **http://lab18.ipu.rssi.ru/help2/.** Там вы найдете условия распространения, включающие бесплатную поддержку — каждые 3-4 месяца выходят дополнения к справкам, которые распространяются бесплатно тем, кто приобрел начальную версию.

Распространение справок через Интернет не означает, что вы обязательно должны иметь доступ в Интернет с домашнего компьютера и иметь собственный адрес e-mail. Достаточно, если доступ в Интернет и e-mail есть у вас на работе или у кого-то из ваших друзей и знакомых. Вы можете воспользоваться этими возможностями, а затем на дискетах перенести файлы на свой компьютер.

3. Архангельский А. Я., Тагин М. А. Стандартная библиотека функций С. — М: ЗАО «Издательство БИНОМ», 2002

Эта книга (надеюсь, подготовить ее осенью) будет содержать полное и подробное описание и примеры применения всех функций стандартной библиотеки языка С. Форма описания будет подобной главам 3 и 4 данной книги. Но число подробно описанных функций будет намного больше — по видимому, свыше 500. Поскольку стандартная библиотека С используется в самых разных системах, то планируемая книга будет рассчитана не только на пользователей C++Builder. В ней будут содержаться примеры применения функций в Turbo C и Turbo C++, C++Builder, Visual C++. Планируемый объем книги — 300 стр.

4. Архангельский А. Я., Тагин М. А. Стандартная библиотека STL языка C++. — М: ЗАО «Издательство БИНОМ», 2002

Эта книга (она будет подготовлена после предыдущей) будет содержать полное и подробное описание библиотеки STL, коротко рассмотренной в главе 5 данной книги. Будут описаны шаблоны, функции, алгоритмы, классы библиотеки, методика разработки собственных шаблонов функций и алгоритмов. Описание будет рассчитано на пользователей и содержать много примеров применения всех функций и алгоритмов библиотеки. Поскольку стандартная библиотека C++ используется в самых разных системах, то планируемая книга будет рассчитана не только на пользователей C++Builder. В ней будут содержаться примеры ее применения в Turbo C++, C++Builder, Visual C++. Планируемый объем книги — 300 стр.

5. **Архангельский А. Я. Решение типовых задач в C++Builder 6. — М: ЗАО «Издательство БИНОМ», 2003**

Эта книга будет готова не раньше следующего года (в этом году, надеюсь, выйдет аналогичная книга по Delphi 6). Она рассчитана на подготовленных читателей, ознакомившихся с C++Builder, например, по книге [1] или данной. В ней рассматриваются вопросы, совершенно не затронутые в данной книге и в «Программировании в C++Builder 6», или только в них упомянутые: решение типовых вычислительных задач (решение систем линейных и нелинейных уравнений, операции с матрицами, векторами и т.п.), ориентированное на особенности C++Builder 6; графики и диаграммы в C++Builder 6; разработка распределенных приложений; множество задач, связанных с работой в Интернет и интранет и многое другое. Книга содержит множество примеров прикладных программ, многие из которых могут рассматриваться как законченные программные продукты. Так что вы просто можете использовать их в своей текущей работе. В книге содержатся указания по доработке этих приложений для ваших целей, так что вы можете на их основе создавать свои собственные программные средства.

Запланированы в книге также главы «Обо всем понемногу», в которых будет рассмотрено множество частных задач. С подобными задачами приходится сталкиваться при разработке приложений и по ним задается множество вопросов на различных форумах в Интернет.

Научно-техническое издание

Архангельский Алексей Яковлевич

C++Builder 6. Справочное пособие.
Книга 1. Язык C++

Оформление обложки *И.Ю. Буровой*
Компьютерная верстка *С.В. Лычагина, К.А. Свиридова*

Подписано в печать 26.08.02. Формат $70\times100/_{16}$.
Гарнитура «Школьная». Бумага газетная. Печать офсетная.
Усл. печ. л. 44,2. Тираж 4000 экз. Заказ № 3145.

Издательство «БИНОМ-ПРЕСС», 2002 г.
170026, г. Тверь, Комсомольский пр., 12.

Отпечатано в полном соответствии
с качеством предоставленных диапозитивов
в издательско-полиграфическом комплексе «Звезда».
614990, г. Пермь, ГСП-131, ул. Дружбы, 34.