
Прасид Пай, Питер Абрахам

Реактивное программирование на C++



Praseed Pai, Peter Abraham

C++ Reactive Programming

*Design concurrent and asynchronous applications using
the RxCpp library and Modern C++17*



Прасид Пай, Питер Абрахам

Реактивное программирование на C++



*Проектирование параллельных и асинхронных приложений
с использованием библиотеки RxCpp и современного C++17*



Москва, 2019

УДК 004.4
ББК 32.973.202-018.2
П12

Пай П., Абрахам П.

П12 Реактивное программирование на C++ / пер. с англ. В. Ю. Винника. – М.: ДМК Пресс, 2019. – 324 с.: ил.

ISBN 978-5-97060-778-7

В книге изложены понятия и принципы функционального реактивного программирования, помогающие строить параллельные, асинхронные приложения с наименьшими усилиями и минимумом ошибок. Реактивное программирование – парадигма программирования, ориентированная на потоки данных и распространение изменений, это путь для лёгкого создания пользовательских интерфейсов, анимации или моделирования систем, изменяющихся во времени.

Всесторонне рассмотрена библиотека RxCpp, описана разработка реактивных микросервисов на C++, а также использование библиотеки Qt/C++ в реактивном стиле. Изучив эту книгу, вы будете хорошо разбираться в тонкостях реактивной модели программирования и методах её реализации на новейшей версии стандарта C++17.

Издание предназначено для разработчиков C++, желающих получить максимум эффективности от своих приложений.

УДК 004.4
ББК 32.973.202-018.2

Authorized Russian translation of the English edition of C++ Reactive Programming ISBN 9781788629775 © 2018 Packt Publishing.

This translation is published and sold by permission of Packt Publishing, which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-78862-977-5 (англ.)
ISBN 978-5-97060-778-7 (рус.)

© 2018 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2019

Содержание

Над книгой работали	11
Предисловие	12
 Глава 1. Модель реактивного программирования – обзор и история	 17
Событийно-ориентированная модель программирования	18
Событийно-ориентированное программирование в системе X Window.....	19
Событийно-ориентированное программирование в среде Microsoft Windows	20
Событийно-ориентированное программирование в каркасе Qt	22
Событийно-ориентированное программирование средствами библиотеки MFC	23
Прочие модели событийно-управляемого программирования	24
Ограничения классических моделей обработки событий.....	24
Реактивная модель программирования	25
Ключевые интерфейсы реактивной программы.....	26
Методы вталкивания и втягивания данных	28
Дуальность интерфейсов IEnumerable и IObservable	28
Превращение событий в наблюдаемый источник	31
Методологические замечания	36
Итоги	37
 Глава 2. Современный язык C++ и его ключевые идиомы	 39
Принципы проектирования языка C++.....	40
Абстракция нулевой стоимости	40
Выразительность	40
Взаимозаменяемость	43
Усовершенствования языка, повышающие качество кода	44
Автоматический вывод типов	44
Единообразный синтаксис инициализации.....	46
Вариадические шаблоны	46
Ссылки rvalue	48
Семантика перемещения	50
Умные указатели	52
Лямбда-функции	54
Функциональные объекты и лямбда-функции	55

Композиция, карринг и частичное применение функций.....	57
Обёртки над функциями.....	60
Операция композиции функций.....	61
Прочие возможности языка.....	63
Выражения-свёртки	63
Сумма типов: тип variant	64
Прочее	65
Циклы по диапазонам и наблюдатели	65
Итоги	69



Глава 3. Параллельное и многопоточное программирование на языке C++

Что такое параллельное программирование.....	71
Здравствуй, мир потоков!	72
Управление потоками	74
Запуск потока.....	74
Присоединение к потоку.....	75
Передача аргументов в поток	77
Использование лямбда-функций	79
Управление владением	80
Совместный доступ потоков к данным.....	82
Двоичные семафоры	84
Предотвращение тупиков	87
Условные переменные	91
Потокобезопасный стек	93
Итоги	96

Глава 4. Асинхронное программирование

и неблокирующая синхронизация в языке C++

Асинхронные задачи в языке C++	99
Фьючерсы и обещания	100
Класс std::packaged_task	102
Функция std::async	104
Модель памяти в языке C++	106
Параллельный доступ к памяти	106
Соглашение о порядке модификации памяти	107
Атомарные операции и типы в языке C++	108
Атомарные типы.....	108
Тип std::atomic_flag.....	111
Тип std::atomic<bool>	113
Тип std::atomic<T*> и арифметика указателей	116
Общий случай шаблона std::atomic<>	117
Порядок доступа к памяти	118
Последовательно согласованный порядок доступа	119



Результат : последовательная согласованность.....	120
Семантика захвата и освобождения	120
Ослабленный порядок доступа к памяти	122
Неблокирующая очередь.....	124
Итоги	126

Глава 5. Знакомство с наблюдаемыми источниками 127

Шаблон «Наблюдатель»	128
Ограниченность классического шаблона «Наблюдатель»	131
Обобщённый взгляд на шаблоны проектирования	133
Объектно-ориентированная модель программирования и иерархии	135
Обработка выражений с помощью шаблонов «Композит» и «Посетитель»	136
Разглаживание многоуровневых композитов для итеративного доступа	142
Операции отображения и фильтрации списков.....	146
От наблюдателей к наблюдаемым источникам.....	149
Итоги	153

Глава 6. Введение в программирование потоков

событий на языке C++ 155

Что такое программирование потоков данных.....	156
Преимущества модели программирования потоков данных	157
Прикладное программирование с использованием библиотеки Streams	157
Ленивые вычисления	158
Пример программы для обработки потока данных.....	159
Агрегирование значений в парадигме потоков данных	160
Погружение стандартных контейнеров в парадигму потоков данных	160
Несколько слов о библиотеке Streams	161
Программирование потоков событий	162
Преимущества программирования на основе потоков событий.....	162
Библиотека Streamulus и её программная модель	162
Библиотека Spreadsheet для оповещения об изменениях данных	168
Библиотека RaftLib – ещё один инструмент обработки потоков данных	170
Потоки данных и реактивное программирование	172
Итоги	173

Глава 7. Знакомство с моделью маршрутов данных и библиотекой RxCpp 174

Парадигма маршрутов данных.....	175
Знакомство с библиотекой RxCpp	176
Библиотека RxCpp и её модель программирования	177
Простой пример взаимодействия источника с наблюдателем	178
Фильтрация и преобразование потоков данных.....	178

Создание потока из контейнера.....	179
Создание собственных наблюдаемых источников	179
Конкатенация потоков	180
Отписка от потока данных.....	180
Визуальное представление потоков данных	181
Операции над потоками данных.....	181
Операция average.....	182
Операция scan.....	182
Соединение операций в конвейер	183
Работа с планировщиками.....	183
Сага о двух операциях: как разглаживать потоки потоков	186
Прочие важные операции	191
Беглый взгляд на ещё не изученное.....	192
Итоги	193

Глава 8. Ключевые элементы библиотеки RxCpp

Наблюдаемые источники данных	194
Что такое объект-производитель	195
Горячие и холодные источники данных	195
Горячие источники данных	196
Горячие источники данных и механизм повтора	198
Наблюдатели и подписчики.....	199
Единство наблюдаемого и наблюдателя.....	200
Планировщики.....	203
Методы observe_on и subscribe_on.....	206
Планировщик с циклом выполнения run_loop.....	208
Операции над потоками данных.....	209
Операции создания потоков.....	210
Операции преобразования данных	210
Операции фильтрации	211
Операции комбинирования данных	212
Операции обработки ошибок	212
Вспомогательные операции	212
Логические операции.....	213
Математические операции и агрегирование потоков.....	213
Операции для управления подключениями	213
Итоги	214

Глава 9. Реактивное программирование графических интерфейсов на основе каркаса Qt

Введение в программирование интерфейсов пользователя на основе каркаса Qt.....	216
--	-----

Объектная модель библиотеки Qt	217
Сигналы и слоты	218
Подсистема событий	220
Обработчики событий	221
Отправка событий	221
Система метаобъектов	222
Программа «Здравствуй, мир» на основе библиотеки Qt	222
События, сигналы и слоты на примере	225
Создание собственного визуального объекта	225
Создание главного диалогового окна приложения	227
Запуск приложения	231
Интеграция библиотек RxCpp и Qt	232
Реактивная фильтрация событий из каркаса Qt	233
Создание окна и размещение его элементов	235
Наблюдатели для различных типов событий	236
Знакомство с библиотекой RxQt	238
Итоги	241

Глава 10. Шаблоны и идиомы реактивного программирования на языке C++

Объектно-ориентированное программирование и шаблоны проектирования	242
Основные каталоги шаблонов	244
Шаблоны «Банды четырёх»	244
Каталог POSA	245
Ещё раз о шаблонах проектирования	246
От шаблонов проектирования к реактивному программированию	248
Разглаживание иерархии и линейный проход	254
От итераторов к наблюдаемым источникам	256
Шаблон «Ячейка»	257
Шаблон «Активный объект»	260
Шаблон «Ресурс взаимны»	262
Шаблон «Шина событий»	263
Итоги	267

Глава 11. Реактивные микросервисы на языке C++

Язык C++ и веб-программирование	269
Модель программирования REST	269
Библиотека REST SDK для языка C++	270
Программирование HTTP-клиента с использованием библиотеки C++ REST SDK	270
Программирование HTTP-сервера	272
Тестирование HTTP-сервера с помощью утилит curl и postman	275

Создание HTTP-клиента с помощью библиотеки libcurl.....	276
Реактивная библиотека-обёртка RxCurl.....	277
Использование формата JSON с протоколом HTTP.....	278
Использование библиотеки C++ REST SDK для создания сервера.....	282
Обращение к REST-сервисам с помощью библиотеки RxCurl.....	290
Несколько слов об архитектуре реактивных микросервисов.....	292
Мелкоблочные сервисы.....	293
Разнородное хранение данных.....	294
Независимое развёртывание сервисов.....	294
Оркестровка и хореография сервисов.....	295
Реактивный стиль запросов к веб-сервисам.....	295
Итоги.....	295

Глава 12. Особые возможности потоков и обработка

ошибок.....	297
Средства обработки ошибок в библиотеке RxCpp.....	300
Выполнение действия в ответ на ошибку.....	300
Восстановление после ошибки.....	302
Обработка ошибки путём перезапуска источника данных.....	305
Автоматическое выполнение завершающих действий в случае ошибки.....	307
Обработка ошибок и планировщики.....	308
Примеры обработки потоков событий.....	313
Агрегирование потоков данных.....	313
Событийно-управляемое приложение.....	315
Итоги.....	319



Над книгой работали

Авторы

Прасид Пай работает в индустрии программного обеспечения на протяжении 25 лет. Начиная с системного программирования в среде MS DOS на языке ANSI C. Принимал активное участие в разработке крупных кроссплатформенных систем на языке C++ для систем Windows, GNU Linux и macOS. Обладает опытом применения технологий COM+ и CORBA на языке C++. В последнее десятилетие работает с языком Java и платформой .Net.

Выступил основным разработчиком компилятора с языка SLANG4.net, первоначально написанного на языке C#, а затем портированного на язык C++ с использованием системы LLVM. Соавтор книги «.NET Design Patterns» («Шаблоны проектирования для платформы .NET»), вышедшей в издательстве Packt Publishing.

Питер Абрахам стал приверженцем языка программирования C++ и пылким борцом за производительность кода со времён своей учёбы в колледже, где он достиг высот в программировании для операционных систем Windows и GNU Linux. Он обогатил свой опыт программированием для архитектуры CUDA, обработкой изображений, компьютерной графикой, работая в таких компаниях, как Quest Global, Siemens и Tektronics.

В своей профессиональной деятельности Питер постоянно использует последние нововведения языка C++ и библиотеку RxCpp. Он обладает большим опытом работы с инструментами разработки кроссплатформенных графических приложений, такими как Qt, WxWidgets и FOX toolkit.

РЕЦЕНЗЕНТ

Сумант Тамбе – разработчик программного обеспечения, исследователь, участник разработки с открытым кодом, блогер, докладчик на различных конференциях, автор многочисленных статей и любитель компьютерных игр. Опытен в применении современного языка C++, брокера сообщений Kafka, различных служб распространения данных, методологии реактивного программирования и потоков данных для решения новых задач, возникающих в области больших данных и интернета вещей.

Автор блога «C++ Truths» («Истины о языке C++») и вики-книги «More C++ Idioms» («Ещё идиомы программирования на языке C++»). Делится своими знаниями в блоге, на конференциях, семинарах и встречах профессионального сообщества. Удостаивался звания Microsoft MVP в области технологий разработки программ на протяжении пяти лет. Имеет докторскую степень по компьютерным наукам в университете Вандербилта.

Предисловие

Эта книга поможет читателю овладеть парадигмой реактивного программирования на языке C++ и создавать асинхронные и многопоточные приложения. Книга включает в себя задачи, взятые из реальной практики, которые читателю предстоит решать с помощью реактивной модели программирования. Здесь освещен долгий путь становления средств обработки событий. Читатель узнает о поддержке параллельного программирования в языке C++ и о функциональном реактивном программировании. Описанные в книге конструкции на базе объектно-ориентированного и функционального программирования позволят читателю создавать эффективные программы. Наконец, читатель узнает о программировании микросервисов на языке C++ и научится создавать собственные операции для библиотеки RxCpp.

Для кого предназначена эта книга

Разработчик, программирующий на языке C++ и интересующийся применением реактивного программирования для создания асинхронных и параллельных приложений, найдёт эту книгу чрезвычайно интересной. От читателя не требуется наличие каких-либо знаний реактивного программирования.

Что охватывает эта книга

В главе 1 «Модель реактивного программирования – обзор и история» вводятся некоторые структуры данных, ключевые для реактивной модели программирования (для краткости – Rx). В ней речь идёт также об обработке событий в графических интерфейсах пользователя, дан общий обзор реактивного программирования, рассказано о реализации графических версий различных видов интерфейса на основе библиотеки классов MFC.

В главе 2 «Современный язык C++ и его ключевые идиомы» рассказано о тонкостях языка C++, о правилах вывода типов, шаблонах с переменным числом аргументов, ссылках rvalue и семантике перемещения, лямбда-функциях, основах функционального программирования, соединении операций в конвейер, а также о том, как реализовать итератор и наблюдателя.

Глава 3 «Параллельное и многопоточное программирование на языке C++» содержит сведения о средствах многопоточного программирования, включённых в стандарт языка C++. Читатель узнает, как запустить поток и управлять им, а также о различных тонкостях стандартной библиотеки, связанных с этим. Эта глава представляет собой хорошее введение в средства поддержки многопоточности, появившиеся в новом стандарте языка C++.

В главе 4 «Асинхронное программирование и неблокирующая синхронизация в языке C++» рассказано о средствах, предоставляемых стандартной библиотекой для организации параллельных вычислений на основе задач. Также в ней говорится о появившейся в современном языке C++ модели памяти для многопоточного программирования.

В главе 5 «Знакомство с наблюдаемыми источниками» говорится об одном из шаблонов проектирования «Банды четырёх» – шаблоне «Наблюдатель» и о его недостатках. Читатель узнает о шаблонах проектирования «Компоновщик» и «Посетитель» в контексте моделирования дерева синтаксического разбора выражения.

В главе 6 «Введение в программирование потоков событий на языке C++» внимание сосредоточено на программировании потоков событий. Также рассматривается библиотека Streamulus, в которой реализован подход к обработке потоков событий на основе встраиваемых предметно-ориентированных языков (англ. Domain-Specific Embedded Language, DSEL). Изложение сопровождается рядом примеров программ.

Глава 7 «Знакомство с моделью потоков данных и библиотекой RxCpp» открывается общим обзором вычислительной парадигмы на основе потоков данных, затем показаны основы создания программ с помощью библиотеки RxCpp. Читатель изучит набор операций, предоставляемых этой библиотекой.

Глава 8 «Ключевые элементы библиотеки RxCpp» даёт представление о том, как средства библиотеки RxCpp взаимодействуют между собой. Глава открывается разбором объектов наблюдения (Observable), далее описаны механизм подписки и реализация расписания.

Глава 9 «Реактивное программирование графических интерфейсов с помощью библиотеки Qt» посвящена применению парадигмы реактивного программирования для создания графических интерфейсов пользователя на основе библиотеки Qt. Читатель узнает о концептуальной основе библиотеки Qt, об иерархии её классов, системе метаобъектов, а также о механизме сигналов и слотов. В качестве примера создаётся приложение, обрабатывающее события от мыши и фильтрующее их. Затем читателю предстоит знакомство с более сложной темой – как создавать собственные реактивные операции средствами библиотеки RxCpp, если имеющихся в ней операций не хватает для той или иной задачи. Знание этих аспектов также поможет понять, как создавать композитные операции, состыковывая уже имеющиеся. Последняя тема в данной книге не освещается, подробную информацию можно найти по адресу https://www.packtpub.com/sites/default/files/downloads/Creating_Custom_Operators_in_RxCpp.pdf.

Глава 10 «Шаблоны и идиомы для реактивного программирования на языке C++» погружает читателя в чудесный мир шаблонов проектирования и идиом языка программирования. Изложение начинается с шаблонов «Банды четырёх», затем разобраны шаблоны, специфические для реактивного программирования.

В главе 11 «Реактивные микросервисы на языке C++» показано, как реактивная модель программирования может использоваться для создания реактив-

ных микросервисов на языке C++. Читатель узнает о наборе средств разработки Microsoft C++ REST SDK и связанной с ним модели программирования.

Глава 12 «Особые возможности потоков и обработка ошибок» посвящена средствам обработки ошибок из библиотеки RxCpp, а также некоторым сложным конструкциям и операциям для обработки потоков событий. Рассказано, как продолжать поток событий после возникновения ошибки, как ожидать, пока источник потока исправляет ошибочное состояние, и возобновлять обработку и как работают обобщённые операции, способные обрабатывать как ошибочные, так и нормальные ситуации.

КАК ИЗВЛЕЧЬ ИЗ ЭТОЙ КНИГИ МАКСИМУМ ПОЛЬЗЫ

Для понимания большинства вопросов, рассматриваемых в этой книге, читателю необходимо владеть программированием на языке C++.

ЗАГРУЗКА ИСХОДНОГО КОДА ПРИМЕРОВ

Файлы с исходным кодом примеров программ можно найти на сайте www.packtpub.com. Покупатель этой книги может посетить страницу www.packtpub.com/support, зарегистрироваться и получить файлы по электронной почте на свой адрес.

Для загрузки файлов нужно выполнить следующие шаги.

1. Войти или зарегистрироваться на сайте www.packtpub.com.
1. Перейти по ссылке **Support**.
2. Щёлкнуть по ссылке **Code Downloads & Errata**.
3. Ввести английское название книги в поле **Search** и выполнить дальнейшие инструкции на сайте.

Когда файлы получены, нужно распаковать их с помощью последних версий архиваторов:

- WinRAR или 7-Zip для ОС Windows;
- Zipreg, iZip или UnRarX для системы Mac;
- 7-Zip или PeaZip для системы Linux.

Исходный код всех примеров к этой книге также можно найти в системе GitHub по адресу <https://github.com/PacktPublishing/Cpp-Reactive-Programming>. Возможные обновления этих примеров будут публиковаться в этом же репозитории.

По адресу <https://github.com/PacktPublishing/> можно также найти обширные репозитории с примерами и к другим книгам и видеолекциям. Рекомендуем читателю ознакомиться с ними.

КАК ЗАГРУЗИТЬ ЦВЕТНЫЕ ИЛЛЮСТРАЦИИ

Издательство также предоставляет файл в формате PDF, в котором собраны цветные рисунки, снимки экрана, диаграммы и другие иллюстрации к этой

книге. Их можно загрузить по адресу https://www.packtpub.com/sites/default/files/downloads/CPPReactiveProgramming_ColorImages.pdf.

ПРИНЯТЫЕ В ТЕКСТЕ СОГЛАШЕНИЯ

На протяжении всей книги текст оформлен с использованием следующих соглашений.

Код_в_тексте: таким шрифтом набраны размещённые в основном тексте элементы исходного кода, имена таблиц баз данных, имена директорий и файлов, расширения имён файлов, пути к файлам, адреса сетевых ресурсов, пользовательский ввод, идентификаторы пользователей в сети Twitter и т. д. Пример использования: «Приведённый выше фрагмент кода инициализирует объект структурного типа `WNDCLASS` (или `WNDCLASSEX` на новых системах) нужным шаблоном окна».

Самостоятельные фрагменты кода, вынесенные в отдельный абзац, оформлены следующим образом:

```
/* закрыть соединение с сервером */
XCloseDisplay(display);
```

```
return 0;
}
```

Если нужно обратить внимание читателя на определённую часть кода, она набирается полужирным шрифтом.

```
/* закрыть соединение с сервером */
XCloseDisplay(display);
```

```
return 0;
}
```

Всё, что вводится или выводится в консоли, передано в книге следующим образом:

```
$ mkdir css
$ cd css
```

Полужирным начертанием отмечены новые термины, важные слова и текст, отображаемый программой на экране. Например, таким способом показаны пункты меню и сообщения в диалоговых окнах. Пример использования: «На жаргоне оконного программирования это называется циклом обработки **сообщений**».



Так будут оформляться предупреждения и важные примечания.



Так будут оформляться советы или рекомендации.

ОТЗЫВЫ И ПОЖЕЛАНИЯ

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу www.dmkpress.com.

СПИСОК ОПЕЧАТОК

Хотя мы приняли все возможные меры, для того чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу www.dmkpress.com, и мы исправим это в следующих тиражах.

НАРУШЕНИЕ АВТОРСКИХ ПРАВ

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты www.dmkpress.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Модель реактивного программирования – обзор и история

Появление систем X Window System, Microsoft Windows и IBM OS/2 Presentation Manager сделало популярным программирование графических интерфейсов для платформы PC. Это стало большим шагом вперёд по сравнению с преобладавшим ранее интерфейсом командной строки и подходом к программированию, ориентированным на пакетную обработку. Реагирование на событие оказалось в центре внимания программистов по всему миру, тогда как разработчики платформ принялись за создание прикладных программных интерфейсов, с помощью которых программисты могли бы обрабатывать события, – низкоуровневых, в духе языка C, использующих указатели на функции обратного вызова. Модели программирования при этом основывались в основном на модели кооперирующихся потоков, а по мере появления более совершенных микропроцессоров большинство платформ стало поддерживать также и вытесняющую многопоточность. Обработка событий (как и другие асинхронные задачи) становилась всё сложнее, и традиционные подходы к реагированию программ на события всё менее поддавались масштабированию. Несмотря на то что появлялись превосходные инструменты для создания графических интерфейсов, основанные на языке C++, для обработки событий по-прежнему использовались числовые коды сообщений, диспетчеризация через указатели на функции и другие низкоуровневые технологии. Ведущие разработчики компиляторов даже пытались добавлять свои расширения в язык C++, чтобы облегчить программирование в среде Windows. Обработка событий, асинхронные вычисления и другие подобные задачи требовали новых подходов. К счастью, современный стандарт языка C++ поддерживает функциональную парадигму программирования, содержит средства управления потоками вместе с подходящей моделью памяти и улучшенные средства управления памятью, что позволяет программистам работать с асинхронными потоками данных, в частности трактовать события как потоки. Всё это достигается благодаря модели

программирования, называемой реактивным программированием. Чтобы показать общую картину явления, осветим в этой главе следующие вопросы:

- событийно-ориентированная модель программирования и её реализации на различных платформах;
- что собой представляет реактивное программирование;
- различные модели реактивного программирования;
- разбор нескольких простых программ для закрепления понимания основных понятий;
- методология, принятая в данной книге.

СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ МОДЕЛЬ ПРОГРАММИРОВАНИЯ

Событийно-ориентированное программирование – это такая модель программирования, в которой ход выполнения программы определяется событиями. Примерами событий могут быть нажатия на кнопку мыши, нажатия клавиш на клавиатуре, жесты на сенсорном экране, сигналы от датчиков, сообщения от других программ и т. д. Событийно-ориентированное приложение основано на механизмах, позволяющих обнаружить события в реальном масштабе времени (или близко к тому) и отвечать, реагировать на них, вызывая подходящие процедуры – обработчики событий. Поскольку большинство ранних программ, обрабатывающих события, было написано на языках C и C++, для организации обработчиков в них применялись низкоуровневые технологии наподобие указателей на функции обратного вызова. Более поздние системы, такие как Visual Basic, Delphi и другие среды быстрой разработки приложений, содержали уже встроенные средства событийно-ориентированного программирования. Чтобы отчетливее показать предмет, сделаем краткий обзор механизмов обработки событий в нескольких различных платформах. Это поможет читателю лучше понять круг проблем, для решения которых предназначены реактивные модели программирования (в контексте разработки графических интерфейсов пользователя).

i В реактивном программировании данные рассматриваются как потоки, а события в системах оконного интерфейса могут рассматриваться как потоки, разнородные элементы которых должны обрабатываться единообразно. Реактивная модель программирования предоставляет средства для сбора событий из разных источников в поток, фильтрации потоков, различных преобразований над потоками, выполнения тех или иных действий над элементами потоков и т. д. Эта модель программирования содержит в своей основе средства асинхронной обработки и управление расписанием асинхронных действий. В этой главе в основном рассматриваются ключевые структуры данных, характерные для реактивного программирования, и то, как создавать простейшие реактивные программы. Реальным реактивным программам внутренне присущ асинхронный принцип работы, тогда как примеры из этой главы работают синхронно. Прежде чем переходить к асинхронному порядку выполнения и управлению расписанием, предстоит сперва объяснить ряд теоретических принципов и соответствующих языковых конструкций, это будет сделано в следующих главах. Примеры из данной главы служат только для первоначального знакомства с предметом и представляют лишь учебный интерес.

Событийно-ориентированное программирование в системе X Window

Система X Window представляет собой кроссплатформенный интерфейс прикладного программирования, поддерживается главным образом в системах, отвечающих стандарту POSIX, а также перенесена в систему Microsoft Windows. Фактически программный интерфейс X представляет собой сетевой протокол оконного графического ввода-вывода, которому требуется оконный менеджер, управляющий совокупностью окон. Клиентское приложение формирует графические образы, а X-сервер отвечает за их отображение на экране конкретной машины. В персональных настольных системах, как правило, графический клиент и сервер работают локально, на одной и той же машине. Следующая программа поможет читателю понять дух модели программирования, лежащей в основе библиотеки XLib, и присущий ей способ обработки событий.

```
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(void)
{
    Display *display;
    Window window;
    XEvent event;
    char *msg = "Hello, World!";
    int s;
```

В этом фрагменте кода подключены необходимые заголовочные файлы, в которых содержатся прототипы функций, предоставляемых библиотекой XLib для языка C. Создавая программы на основе одной лишь библиотеки XLib, программисту нужно иметь дело с некоторыми специфическими структурами данных. В наше время, впрочем, для создания коммерческих программных продуктов чаще всего используют такие высокоуровневые библиотеки-обёртки, как Qt, WxWidgets, Gtk+ или Fox.

```
/* open connection with the server */
display = XOpenDisplay(NULL);
if (display == NULL){
    fprintf(stderr, "Cannot open display\n");
    exit(1);
}
s = DefaultScreen(display);
/* create window */
window = XCreateSimpleWindow(display,
    RootWindow(display, s), 10, 10, 200, 200, 1,
    BlackPixel(display, s), WhitePixel(display, s));

/* select kind of events we are interested in */
```



```
XSelectInput(display, window, ExposureMask | KeyPressMask);
```

```
/* map (show) the window */
XMapWindow(display, window);
```

В этом фрагменте кода инициализируется соединение с графическим сервером, затем создаётся окно с заданными параметрами. Как правило, программы в среде X Window работают под управлением оконного менеджера, который управляет взаимным расположением окон. Мы выбрали интересные нашей программе типы сообщений, вызвав функцию `XSelectInput`, перед тем как отобразить окно.

```
/* event loop */
for (;;)
{
    XNextEvent(display, &event);

    /* draw or redraw the window */
    if (event.type == Expose)
    {
        XFillRectangle(display, window,
            DefaultGC(display, s), 20, 20, 10, 10);
        XDrawString(display, window,
            DefaultGC(display, s), 50, 50, msg, strlen(msg));
    }
    /* exit on key press */
    if (event.type == KeyPress)
        break;
}
```

Затем программа входит в бесконечный цикл, в котором запрашивает очередное событие, а соответствующая функция из библиотеки `XLib` отображает в окне текстовую строку. На жаргоне оконного программирования это называется циклом обработки **сообщений**. Для получения очередного события используется функция `XNextEvent`.

```
/* close connection to server */

XCloseDisplay(display);
return 0;
}
```

Покинув «бесконечный» цикл обработки сообщений, программа закрывает соединение с графическим сервером.

Событийно-ориентированное программирование в среде Microsoft Windows

Корпорация Microsoft разработала модель программирования графических интерфейсов пользователя, которую можно считать наиболее успешной оконной системой в мире. Третья версия системы Windows имела ошеломительный

успех в 1990 г., и фирма Microsoft продолжила его развивать в версиях Windows NT, Windows 95, 98, ME. Рассмотрим в общих чертах модель событийно-ориентированного программирования в системе Microsoft Windows (за подробной информацией о том, как работает эта модель, можно обратиться к документации фирмы Microsoft). Следующий пример поможет понять суть программирования в среде Windows на языках С и С++.

```
#include <windows.h>
//----- Prtotype for the Event Handler Function
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam, LPARAM lParam);
//----- Entry point for a Idiomatic Windows API function
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg = { 0 };
    WNDCLASS wc = { 0 };
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.hbrBackground = (HBRUSH)(COLOR_BACKGROUND);
    wc.lpszClassName = "minwindowsapp";
    if (!RegisterClass(&wc))
        return 1;
```

Приведённый выше фрагмент кода инициализирует объект структурного типа WNDCLASS (или WNDCLASSEX на новых системах) нужным шаблоном окна. Самое важное в этой структуре – это поле lpfnWndProc, в нём находится адрес функции, посредством которой экземпляр окна отвечает на события.

```
if (!CreateWindow(wc.lpszClassName,
    "Minimal Windows Application",
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    0, 0, 640, 480, 0, 0, hInstance, NULL))
    return 2;
```

Вызов функции CreateWindow (или CreateWindowEx на новых версиях ОС Windows) создаёт окно на основе оконного класса с именем, взятым из параметра WNDCLASS.lpszClassname.

```
while (GetMessage(&msg, NULL, 0, 0) > 0)
    DispatchMessage(&msg);
return 0;
}
```

Этот блок кода запускает цикл, в который берёт из очереди новые сообщения до тех пор, пока не будет получено сообщение WM_QUIT. Сообщение WM_QUIT завершает цикл. В некоторых случаях их необходимо также подвергнуть некоторой предварительной обработке, перед тем как передавать их функции DispatchMessage. Наконец, системная функция DispatchMessage вызывает оконную функцию обратного вызова, адрес которой был ранее передан через поле lpfnWndProc.

```

LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam, LPARAM lParam) {
    switch (message) {
        case WM_CLOSE:
            PostQuitMessage(0); break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

```

Показанный выше фрагмент кода представляет собой минимальную функцию обратного вызова (в англоязычной литературе – callback). Обратившись к документации фирмы Microsoft, можно узнать больше о прикладном интерфейсе программирования в среде Windows и о том, как события обрабатываются в программах.

Событийно-ориентированное программирование в каркасе Qt

Каркас Qt представляет собой кроссплатформенный и многоплатформенный инструмент профессиональной разработки программ, включая разработку графических интерфейсов пользователя, который работает в средах Windows, GNU Linux, macOS X и в других системах семейства Mac. Этот инструмент поддерживает также встроенные системы и мобильные устройства. Модель программирования на языке C++ в этом каркасе основана на использовании специального инструмента, называемого **метаобъектным компилятором** (англ. Meta Object Compiler, MOC); он просматривает исходный код в поисках директив (особых макросов и расширений языка), помещённых в исходный код, и определённым образом генерирует вспомогательный исходный код, отвечающий за обработку событий. Таким образом, перед тем как компилятор языка C++ получит исходный код, этот код должен пройти сквозь инструмент MOC, который сгенерирует код, отвечающий стандарту ANSI C++ и не содержащий языковых конструкций, специфических для системы Qt. Более подробные сведения можно почерпнуть из документации по каркасу Qt. Следующая простая программа демонстрирует ключевые моменты программирования в каркасе Qt и присущую ему логику обработки событий.

```

#include <qapplication.h>
#include <qdialog.h>
#include <qmessagebox.h>
#include <qobject.h>
#include <qpushbutton.h>

class MyApp : public QDialog {
    Q_OBJECT
public:
    MyApp(QObject* /*parent*/ = 0):
        button(this)
    {

```



```

button.setText("Hello world!");
button.resize(100, 30);
// When the button is clicked, run button_clicked
connect(&button,
        &QPushButton::clicked, this, &MyApp::button_clicked);
}

```

Макрос `Q_OBJECT` указывает метаобъектному компилятору сгенерировать таблицу диспетчеризации события (Event Dispatch). Когда источник событий подключается к приёмнику, в эту таблицу вставляется новая строка. Сгенерированный код обрабатывается компилятором наряду с исходным кодом, написанным программистом, в результате чего строится исполняемая программа.

```

public slots:
    void button_clicked() {
        QMessageBox box;
        box.setWindowTitle("Howdy");
        box.setText("You clicked the button");
        box.show();
        box.exec();
    }
protected:
    QPushButton button;
};

```

Слово `slots` как расширение языка особым образом обрабатывается метаобъектным компилятором при генерации вспомогательного кода, но для компилятора языка C++ прозрачно, так как представляет собой обычный макрос.

```

int main(int argc, char** argv) {
    QApplication app(argc, argv);
    MyApp myapp;
    myapp.show();
    return app.exec();
}

```

Этот последний фрагмент кода инициализирует объект, играющий роль обёртки над приложением, и отображает главное окно. В целом Qt можно назвать наилучшим из всех каркасов для разработки приложений, разработанных для языка C++. Кроме того, в нём имеются хорошие средства интеграции с популярным языком программирования Python.

Событийно-ориентированное программирование средствами библиотеки MFC

Библиотека классов MFC (Microsoft Foundation Classes) по сей день остаётся довольно популярным средством для создания приложений в среде Microsoft Windows. Если прибавить к ней ещё и библиотеку ATL (**A**ctiveX **T**emplate **L**ibrary), можно получить также некоторую поддержку веб-программирования. Для обработки событий в библиотеке MFC используется механизм, называемый схемой сообщений, или таблицей сообщений (message map). Таблица

сообщений, оформленная с помощью специальных макросов, как показано в следующем примере, присутствует в каждой программе, созданной на основе библиотеки MFC.

```
BEGIN_MESSAGE_MAP(CClockFrame, CFrameWnd)
    ON_WM_CREATE()
    ON_WM_PAINT()
    ON_WM_TIMER()
END_MESSAGE_MAP()
```



Эта таблица определяет реакцию на стандартные сообщения системы Windows: сообщения WM_CREATE, WM_PAINT и WM_TIMER должны обрабатываться, соответственно, функциями OnCreate, OnPaint и OnTimer. На уровне внутренних механизмов реализации, глубоко скрытых от пользователя, эти таблицы представляют собой массивы, в которых целочисленный код сообщения используется для поиска нужной строки, содержащей указатель на функцию-обработчик. Таким образом, отличие данного подхода от модели обработки сообщений на основе системных вызовов Windows оказывается при внимательном рассмотрении не слишком значительным.



Мы не приводим здесь пример исходного кода, поскольку среди доступных для скачивания примеров программ к этой книге имеется полная реализация одного характерного графического приложения в духе модели реактивного программирования на основе библиотеки MFC. Читатель может изучить исходный код и комментарии к нему, чтобы разобраться в неочевидных аспектах обработки событий в библиотеке MFC.

Прочие модели событийно-управляемого программирования

Системы распределённой обработки объектов, такие как COM+ или CORBA, обладают собственными подсистемами обработки событий. Модель событий, принятая в технологии COM+, основана на понятии «точки соединения» (англ. connection point), представленном интерфейсами IConnectionPointContainer и IConnectionPoint, а в технологии CORBA реализована модель на основе так называемого сервиса событий¹ (event service). Спецификация CORBA поддерживает оба механизма оповещения о событиях: «втягивание» и «вталкивание». Разбор технологий COM+ и CORBA выходит за рамки этой книги, читатель может обратиться к соответствующей документации.

Ограничения классических моделей обработки событий

Цель этого краткого обзора моделей обработки событий, присущих различным платформам, состоит в том, чтобы показать предмет с нужной точки зрения.

¹ Следует отметить, что сервис событий в технологии CORBA подвергается критике из-за ряда серьёзных недостатков (схема доставки сообщений, при которой каждый клиент получает сообщения обо всех событиях от сервера, перегружает сеть; отсутствует фильтрация сообщений по каким-либо критериям, синхронный режим отправки сообщений) – см, например, http://www.k-press.ru/cs/2000/3/corba/corba_callback.asp. – Прим. перев.

Логика реагирования на события в этих моделях обычно тесно связана с платформой, для которой создан код. Хотя с появлением многоядерных процессоров создание многопоточного кода на основе низкоуровневых средств стало чересчур сложным, в стандартной библиотеке языка C++ стала доступна высокоуровневая модель параллельного программирования на основе задач. Но ведь источники событий чаще всего написаны отнюдь не на основе стандартной библиотеки! Так, в стандарт языка C++ не включена библиотека для создания графических интерфейсов пользователя, единый интерфейс для доступа к внешним устройствам и т. д. Как преодолеть это противоречие? К счастью, события и данные от внешних источников можно организовывать в потоки или последовательности, которые затем можно весьма эффективно обрабатывать, используя средства функционального программирования, такие как лямбда-функции. При этом нетрудно получить и дополнительную выгоду: если наложить некоторые ограничения на изменение значений объектов, параллельная обработка окажется неотъемлемой частью модели обработки потоков.

РЕАКТИВНАЯ МОДЕЛЬ ПРОГРАММИРОВАНИЯ

Говоря упрощённо, реактивное программирование – это не что иное, как программирование с асинхронными потоками данных. Применяя к потокам те или иные операции, можно решать различные вычислительные задачи. Первая задача реактивной программы – превратить свои данные в поток, из какого бы источника эти данные ни были получены. Так, создавая современное приложение с графическим интерфейсом, нужно обрабатывать события перемещения и нажатия кнопок мыши. Сейчас в большинстве приложений функция обратного вызова (callback) вызывается при наступлении такого события и сразу обрабатывает его. При этом большую часть времени обработчик события занят фильтрацией событий по тем или иным критериям, а затем вызывает функцию для обработки конкретной разновидности события. В контексте этой задачи применить реактивную модель программирования – значит собрать события от мыши (типа перемещения и нажатия кнопки) в коллекцию, затем установить на эту коллекцию фильтр, наконец оповещать обработчики. В таком случае логика взаимодействия приложения с обработчиком не будет вызываться без необходимости.

Модель, основанная на обработке потоков, широко известна и довольно проста в реализации. Почти всё, что угодно, можно превратить в поток. Примерами потоков могут служить сообщения, логи, каналы в сети Twitter, блоги, ленты новостей и т. д. Методы функционального программирования очень хорошо подходят для обработки потоков. Такой язык, как C++ современного стандарта, включающий превосходную поддержку как объектно-ориентированного, так и функционального стиля программирования, становится очевидным выбором для написания реактивных программ. Главная идея, лежащая в основе реактивного программирования, состоит в том, что некоторые типы данных

могут представлять значения, протяжённые во времени. Такие типы данных (конкретно, последовательности) в этой парадигме программирования играют роль наблюдаемых источников (observable). Результаты вычислений, зависящих от изменяющихся со временем значений, в свою очередь, представляют собой значения, изменяющиеся со временем. Изменяемое значение должно получать асинхронные оповещения всякий раз, когда изменяется другое значение, от которого оно зависит.

i Несмотря на то что главный предмет этой книги составляет реактивное программирование, в этой главе мы будем заниматься в основном объектно-ориентированным подходом. Это необходимо для того, чтобы определить ряд ключевых интерфейсов (для чего в языке C++ используется аппарат виртуальных функций), которые в дальнейшем понадобятся для собственно реактивного программирования. Позднее, когда будут изучены конструкции языка C++ для поддержки функционального программирования, читатель сможет самостоятельно построить отображение объектно-ориентированных конструкций на функциональные. Кроме того, в этой главе мы намеренно устранимся от обсуждения вопросов параллельной обработки, чтобы сконцентрировать внимание на программных интерфейсах. В главах 2 «Современный язык C++ и его ключевые идиомы», 3 «Параллельные вычисления и потоки в языке C++» и 4 «Асинхронное программирование и неблокирующая синхронизация в языке C++» разъясняются основы реактивного программирования с использованием средств функционального программирования.

КЛЮЧЕВЫЕ ИНТЕРФЕЙСЫ РЕАКТИВНОЙ ПРОГРАММЫ

Чтобы прояснить, что на самом деле происходит внутри реактивной программы, создадим несколько простых программ, демонстрирующих основные моменты. С точки зрения проектирования, если отвлечься от вопросов параллельной обработки и сфокусировать внимание лишь на программных интерфейсах, реактивная программа должна состоять из следующих частей:

- источник событий, реализующий интерфейс `IObservable<T>`;
- приёмник событий (также называемый наблюдателем или подписчиком), реализующий интерфейс `IObserver<T>`;
- механизм подписки наблюдателя на события от некоторого подписчика;
- механизм оповещения подписчиков о данных, поступающих из источника.

i В этой главе, и только в ней, примеры написаны с использованием классической версии языка C++, так как необходимые элементы новейшего стандарта языка будут описаны лишь в последующих главах. В частности, в приведённых ниже примерах используются «сырые» указатели, чего практически всегда можно избежать, если создавать код на новой версии языка C++. Код примеров в этой главе написан так, чтобы в целом соответствовать требованиям, описанным в документации к системе ReactiveX. При создании кода на языке C++ мы не будем пользоваться подходами на основе наследования, которые обычно применяются при программировании на языках Java и C#.

Для начала определим интерфейсы для наблюдателя (IObserver) и наблюдаемого источника (IObservable), а также класс исключения CustomException.

```
#pragma once
//Common2.h

struct CustomException /*: public std::exception */ {
    const char * what() const throw () {
        return "C++ Exception";
    }
};
```

Класс CustomException – лишь заглушка, нужная для полноты интерфейса. Поскольку мы договорились придерживаться в этой главе стандартных средств языка C++, сохраним совместимость с классом std::exception.

```
template<class T> class IEnumerator
{
public:
    virtual bool HasMore() = 0;
    virtual T next() = 0;
};

template <class T> class IEnumerableable
{
public:
    virtual IEnumerator<T> *GetEnumerator() = 0;
};
```

Интерфейс IEnumerableable составляет основу для источников данных; он предоставляет клиенту возможность перебирать свои элементы посредством интерфейса IEnumerator<T>.

i Обращаем внимание читателя, что принцип, положенный в основу пары интерфейсов «перечисляемое-перечислитель» (IEnumerableable<T> и IEnumerator<T>), зеркально отражает идею, на которой основана пара интерфейсов «наблюдаемое-наблюдатель» (IObserver<T> и IObservable<T>). Эту последнюю пару определим следующим образом:

```
template<class T> class IObserver
{
public:
    virtual void OnCompleted() = 0;
    virtual void OnError(CustomException *exception) = 0;
    virtual void OnNext(T value) = 0;
};

template<typename T>
class IObservable
{
public:
    virtual bool Subscribe(IObserver<T>& observer) = 0;
};
```

Интерфейс `IObserver<T>` – это интерфейс для приёмника данных, через который он получает оповещения от источника данных. Источник же, в свою очередь, должен реализовывать интерфейс `IObservable<T>`.



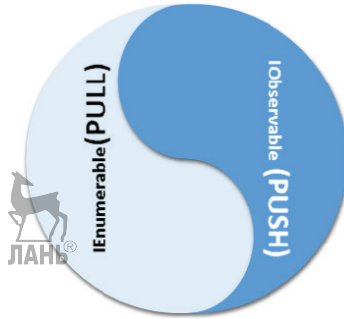
Мы определили интерфейс наблюдателя `IObserver<T>` с тремя методами. Это методы `OnNext`, с помощью которого наблюдатель получает оповещение об очередном элементе данных, `OnCompleted`, которым источник извещает, что у него более нет данных, и метод `OnError`, которым источник извещает об исключении. Интерфейс `IObservable<T>` воплощается источником данных, а приёмники регистрируют в нём свои объекты, обладающие интерфейсом `IObserver<T>`, чтобы получать оповещения.

Методы вталкивания и втягивания данных

В реактивном программировании выделяются два подхода: на основе вталкивания и на основе втягивания. В системе, основанной на принципе втягивания, источник данных ждёт запроса от приёмника (в нашей терминологии – подписчика), чтобы отправить ему очередные элементы потока данных. Таков классический подход, при котором источник данных играет пассивную роль, а приёмник активно запрашивает у него информацию. Для этого удобно применять шаблон итератора; интерфейсы `IEnumerable<T>` и `IEnumerator<T>` предназначены именно для этой модели, синхронной по своей природе (поток-приёмник данных блокируется, пока источник выполняет его запрос). В системе, основанной на вталкивании, напротив, источник данных рассылает события по сети приёмников, инициируя их обработку. В этом случае, в отличие от систем с втягиванием, новые элементы данных передаются подписчикам самим источником, который тем самым моделирует последовательность элементов. Асинхронная природа этой модели обеспечивается тем, что подписчики не блокируются в ожидании данных, а вместо этого реагируют на поступившие изменения. Легко видеть, что использование этого подхода предпочтительнее при создании сложных интерфейсов пользователя, в которых нежелательно блокировать главный поток графического интерфейса в ожидании событий. Вталкивание даёт отличный механизм для обеспечения отзывчивости приложения.

Дуальность интерфейсов `IEnumerable` и `IObservable`

Если присмотреться внимательно, можно обнаружить, что различие между двумя описанными выше подходами не столь значительно. Интерфейс `IEnumerable<T>` можно считать втягивающим эквивалентом вталкивающего интерфейса `IObservable<T>`. Эти интерфейсы фактически дуальны. Когда две сущности обмениваются информацией, втягивание данных со стороны первой из них соответствует вталкиванию со стороны второй. Эта дуальность иллюстрируется следующей диаграммой:



Чтобы лучше понять дуализм вытягивания и вталкивания, рассмотрим пример кода.

i В этой главе при написании кода мы придерживаемся классического языка C++, так как новые средства языка и стандартной библиотеки для поддержки потоков, неблокирующего программирования и другие темы, существенные для реализации реактивной модели программирования на современном языке C++, обсуждаются в других главах.

```
#include <iostream>
#include <vector>
#include <iterator>
#include <memory>
#include "Common2.h"
using namespace std;

class ConcreteEnumerable : public IEnumerable<int>
{
    int *numberlist;
    int _count;
    friend class Enumerator;
public:
    ConcreteEnumerable(int numbers[], int count):
        numberlist(numbers), _count(count) {}
    ~ConcreteEnumerable() {}
    class Enumerator : public IEnumerable<int> {
        int *innumbers, icount, index;
    public:
        Enumerator(int *numbers, int count):
            innumbers(numbers), icount(count), index(0) {}
        bool HasMore() { return index < icount; }
        // строго говоря, следующий метод должен выбрасывать
        // исключение при выходе индекса за допустимые границы;
        // пускай он пока возвращает -1
        int next() { return (index < icount) ?
            innumbers[index++] : -1; }
        ~Enumerator() {}
    };
};
```

```

IEnumerator<int> *GetEnumerator()
    { return new Enumerator(numberlist, _count); }
};

```



Показанный выше класс хранит указатель на массив целых чисел вместе с его размером и позволяет перебирать один за другим его элементы, воплощая интерфейс `IEnumerable<T>`. Логика перебора элементов реализована вложенным классом, который воплощает интерфейс `IEnumerator<T>`.

```

int main()
{
    int x[] = { 1,2,3,4,5 };
    // здесь используются "сырые" указатели, так как умные
    // указатели unique_ptr и shared_ptr излагаются позднее
    ConcreteEnumerable *t = new ConcreteEnumerable(x, 5);
    IEnumerator<int> * numbers = t->GetEnumerator();
    while (numbers->HasMore())
        cout << numbers->next() << endl;
    delete numbers;
    delete t;
    return 0;
}

```

В главной функции программы создаётся экземпляр класса `ConcreteEnumerable`, играющий роль обёртки над обычным массивом, затем происходит перебор элементов через посредство этой обёртки.

Теперь напишем генератор чётных чисел, чтобы показать, как видоизменяется взаимодействие типов данных при преобразовании втягивающей программы во вталкивающую. При этом надёжность программы принесём в жертву краткости листинга.

```

#include <iostream>
#include <vector>
#include <iterator>
#include <memory>
#include "Common2.h"
using namespace std;

class EvenNumberObservable : IObservable<int>
{
    int *_numbers;
    int _count;
public:
    EvenNumberObservable(int numbers[], int count):
        _numbers(numbers), _count(count){}
    bool Subscribe(IObserver<int>& observer) {
        for (int i = 0; i < _count; ++i)
            if (_numbers[i] % 2 == 0)
                observer.OnNext(_numbers[i]);
        observer.OnCompleted();
        return true;
    }
};

```



Этот класс принимает массив целых чисел, отбрасывает нечётные значения, а о каждом найденном в массиве чётном значении информирует наблюдателя, т. е. объект, воплощающий интерфейс `IObservable<T>`. Иными словами, источник данных вталкивает свои данные наблюдателю. Реализация наблюдателя показана ниже:

```
class SimpleObserver : public IObservable<int>
{
public:
    void OnNext(int value) { cout << value << endl; }
    void OnCompleted() { cout << "hello completed" << endl; }
    void OnError( CustomException * ex) {}
};
```

Тем самым класс `SimpleObserver` реализует интерфейс `IObservable<int>` и, следовательно, способен получать оповещения и реагировать на них.

```
int main()
{
    int x[] = { 1,2,3,4,5 };

    EvenNumberObservable * t = new EvenNumberObservable(x, 5);
    IObservable<int> *xy = new SimpleObserver();

    t->Subscribe(*xy);

    delete xy;
    delete t;

    return 0;
}
```

Из этого примера видно, как это просто – подписаться на получение от наблюдаемого источника одних лишь чётных чисел, имея последовательность натуральных чисел. Построенная нами система автоматически проталкивает (иначе говоря, публикует) сообщение для наблюдателя всякий раз, когда обнаруживает в исходных данных чётное число. В этом коде представлены такие образцы реализации ключевых интерфейсов, чтобы из них стало очевидно, что на самом деле происходит «под капотом» системы.

ПРЕВРАЩЕНИЕ СОБЫТИЙ В НАБЛЮДАЕМЫЙ ИСТОЧНИК

Выше мы разобрали, как преобразовать втягивающую программу, основанную на интерфейсе `IEnumerable<T>`, в программу вталкивающую, построенную на паре интерфейсов `IObservable<T>`–`IObserver<T>`. В реальных приложениях, однако, источник данных не столь прост, как показанный выше поток целых чисел. Рассмотрим, как преобразовать события `MouseMove` в поток, для чего создадим небольшую программу на основе библиотеки MFC.

i Библиотека MFC выбрана для этого примера потому, что реактивному программированию на основе более современной библиотеки Qt посвящена отдельная глава. В ней мы рассмотрим создание реактивных программ на основе идиоматичных асинхронных потоков, работающих по принципу вталкивания. В этой же программе, основанной на библиотеке MFC, мы всего лишь будем фильтровать события перемещения мыши, выделяя из них те, что попадают в заданный прямоугольник, и оповещая о таких событиях наблюдателя. В этом примере события обрабатываются синхронно.

```
#include "stdafx.h"
#include <afxwin.h>
#include <afxext.h>
#include <math.h>
#include <vector>
#include "../Common2.h"

using namespace std;

class CMouseFrame :public CFrameWnd, IObservable<CPoint>
{
private:
    RECT _rect;
    POINT _curr_pos;
    vector<IObserver<CPoint>*> _event_src;
public:
    CMouseFrame()
    {
        HBRUSH brush = (HBRUSH)::CreateSolidBrush(
            RGB(175, 238, 238));
        CString mywindow = AfxRegisterWndClass(
            CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS, 0, brush, 0);
        Create(mywindow, _T("MFC Clock By Praseed Pai"));
    }
}
```

В данной части кода объявлен класс для окна приложения. Этот класс порождён от класса CFrameWnd из библиотеки MFC и, помимо того, воплощает интерфейс IObservable<T>, тем самым вынуждая программиста реализовать в этом классе метод Subscribe. Вектор указателей на объекты, обладающие интерфейсом IObserver<T>, используется для хранения всех подписчиков (иначе говоря, наблюдателей), подключённых к этому источнику событий. Хотя в этом примере будет только один наблюдатель, код допускает наличие любого числа наблюдателей.

```
virtual bool Subscribe(IObserver<CPoint>& observer) {
    _event_src.push_back(&observer);
    return true;
}
```

Метод Subscribe просто сохраняет указатель на наблюдателя в векторе и возвращает значение true, означающее успех. Всякий раз, когда мышь изменит своё положение, окно получит оповещение через внутренние механизмы библиотеки MFC, затем, если координаты мыши попадают в заданную прямо-

угольную область, наблюдатели будут об этом извещены. За это отвечает представленный ниже код.

```
bool FireEvent(const CPoint& pt) {
    vector<IObserver<CPoint> *>::iterator it =
        _event_src.begin();
    while (it != _event_src.end()){
        IObserver<CPoint> *observer = *it;
        observer->OnNext(pt);
        // в настоящих реактивных программах установлен
        // строгий порядок вызова методов: метод OnCompleted
        // должен вызываться только после того, как все
        // данные обработаны; этот код лишь демонстрирует
        // общую идею.
        observer->OnCompleted();
        it++;
    }
    return true;
}
```



Метод `FireEvent` проходит по всем наблюдателям и в каждом из них вызывает метод `OnNext`, также вызывает он и метод `OnCompleted`. Механизмы обработки событий в реактивном программировании предполагают ряд правил вызова методов у наблюдателей. В частности, если вызван метод `OnCompleted`, то метод `OnNext` для данного наблюдателя более вызываться не должен. Подобным же образом, если у наблюдателя вызван метод `OnError`, дальнейшие сообщения ему не отправляются. Если бы мы стремились в точности соблюдать все соглашения, принятые в модели реактивного программирования, текст программы вышел бы слишком сложным. Предназначение показанного здесь кода состоит в том, чтобы схематически показать реактивную модель программирования в действии.

```
int OnCreate(LPCREATESTRUCT l){
    return CFrameWnd::OnCreate(l);
}

void SetCurrentPoint(CPoint pt) {
    this->_curr_pos = pt;
    Invalidate(0);
}
```

Метод `SetCurrentPoint` вызывается наблюдателем и устанавливает координаты для последующего отображения текста. Вызов метода `Invalidate` приводит к генерации сообщения `WM_PAINT`, по которому механизмы библиотеки MFC, согласно таблице обработчиков сообщений, вызовут метод `OnPaint`.

```
void OnPaint()
{
    CPaintDC d(this);
    CBrush b(RGB(100, 149, 237));
    int x1 = -200, y1 = -220, x2 = 210, y2 = 200;
```

```

Transform(&x1, &y1); Transform(&x2, &y2);
CRect rect(x1, y1, x2, y2);
d.FillRect(&rect, &b);
CPen p2(PS_SOLID, 2, RGB(153, 0, 0));
d.SelectObject(&p2);

char *str = "Hello Reactive C++";
CFont f;
f.CreatePointFont(240, _T("Times New Roman"));
d.SelectObject(&f);
d.SetTextColor(RGB(204, 0, 0));
d.SetBkMode(TRANSPARENT);
CRgn crgn;
crgn.CreateRectRgn(
    rect.left, rect.top,
    rect.right, rect.bottom);
d.SelectClipRgn(&crgn);
d.TextOut(
    _curr_pos.x, _curr_pos.y,
    CString(str), strlen(str));
}

```

Метод `OnPaint` вызывается самой библиотекой MFC в ответ на вызов метода `Invalidate`. Этот метод отображает строку текста в том месте, которое ранее было установлено методом `SetCurrentPoint`.

```

void Transform(int *px, int *py)
{
    ::GetClientRect(m_hWnd, &_rect);
    int width = (_rect.right - _rect.left) / 2;
    int height = (_rect.bottom - _rect.top) / 2;
    *px = *px + width;
    *py = height - *py;
}

```

Метод `Transform` вычисляет границы внутренней области окна и переводит абстрактные декартовы координаты точки в координаты относительно данного окна. Это вычисление можно было бы ещё лучше выполнить с помощью преобразователей координат, поддерживаемых системой Windows.

```

void OnMouseMove(UINT nFlags, CPoint point)
{
    int x1 = -200, y1 = -220, x2 = 210, y2 = 200;
    Transform(&x1, &y1); Transform(&x2, &y2);
    CRect rect(x1, y1, x2, y2);
    POINT pts;
    pts.x = point.x;
    pts.y = point.y;
    rect.NormalizeRect();
    // В реальной программе эти точки накапливались бы
    // в списке
    if (rect.PtInRect(point)) {

```

```

        // лучше отправлять оповещения, не блокируя поток
        FireEvent(point);
    }
}

```

Метод `OnMouseMove` проверяет, находится ли текущее положение мыши в прямоугольнике, центрированном относительно экрана, и если это так, извещает об этом наблюдателей.

```

    DECLARE_MESSAGE_MAP();
};

BEGIN_MESSAGE_MAP(CMouseFrame, CFrameWnd)
    ON_WM_CREATE()
    ON_WM_PAINT()
    ON_WM_MOUSEMOVE()
END_MESSAGE_MAP()

class WindowHandler : public IObserver<CPoint>
{
private:
    CMouseFrame *window;
public:
    WindowHandler(CMouseFrame *win) : window(win) { }
    virtual ~WindowHandler() { window = 0; }
    virtual void OnCompleted() {}
    virtual void OnError(CustomException *exception) {}
    virtual void OnNext(CPoint value) {
        if (window) window->SetCurrentPoint(value);
    }
};

```



Показанный выше класс `WindowHandler` реализует интерфейс `IObserver<CPoint>` и обрабатывает события, полученные от объекта `CMouseFrame`, который воплощает интерфейс `IObservable<CPoint>`. В этом небольшом примере наблюдатель передаёт окну координаты для отображения текста.

```

class CMouseApp :public CWinApp
{
    WindowHandler *reactive_handler;
public:
    int InitInstance()
    {
        CMouseFrame *p = new CMouseFrame();
        p->ShowWindow(1);
        reactive_handler = new WindowHandler(p);
        p->Subscribe(*reactive_handler);
        m_pMainWnd = p;
        return 1;
    }

    virtual ~CMouseApp() {

```



```

        if (reactive_handler) {
            delete reactive_handler;
            reactive_handler = 0;
        }
    };

    CMouseApp a;

```

Последний фрагмент кода отвечает за инициализацию и запуск приложения. Сначала создаётся и отображается главное окно, затем создаётся объект-наблюдатель, далее наблюдатель подключается к источнику данных. Деструктор отвечает за удаление объекта-наблюдателя. В последней строке создаётся объект, инкапсулирующий приложение в целом, что приводит к запуску всей системы.

МЕТОДОЛОГИЧЕСКИЕ ЗАМЕЧАНИЯ



Цель данной главы состоит в том, чтобы познакомить читателя с ключевыми интерфейсами, лежащими в основе реактивного подхода к программированию, а именно с интерфейсами `IObservable<T>` и `IObserver<T>`. Эти два интерфейса, по существу, дуальны интерфейсам `IEnumerable<T>` и `IEnumerator<T>`. Мы разобрали, как смоделировать эти пары интерфейсов на классической версии языка C++, и написали их упрощённые реализации. Наконец, мы создали программу с графическим интерфейсом, которая перехватывает события от мыши и извещает о них множество наблюдателей. Эти игрушечные реализации позволили немного прикоснуться к идеям и принципам реактивной модели программирования. Показанные здесь реализации можно считать примером объектно-ориентированного реактивного программирования.

Чтобы стать профессионалом в реактивном программировании на языке C++, программисту нужно уверенно овладеть следующими темами:

- новые языковые конструкции в новом стандарте языка C++;
- средства поддержки функционального программирования, поддерживаемые в новом стандарте языка C++;
- модель асинхронного программирования (библиотека RxCpp берёт эту заботу на себя!);
- обработка потоков событий;
- мощные, пригодные для реальных задач библиотеки для реактивного программирования, такие как библиотека RxCpp;
- применение парадигмы реактивного программирования для разработки графических интерфейсов и веб-программирования;
- усложнённые конструкции реактивного программирования;
- обработка ошибок и исключений.

В этой главе речь шла главным образом о ключевых идиомах и о том, зачем вообще нужна стройная модель асинхронной обработки данных. В следующих трёх главах будут разобраны новые средства, появившиеся в языке C++, в особенности средства многопоточного и параллельного программирования, а также средства неблокирующей синхронизации, ставшие возможными благодаря появлению моделей памяти с гарантиями. Всё это заложит прочный фундамент для овладения функциональным реактивным программированием.

В главе 5 «Знакомство с наблюдаемыми источниками» мы вернёмся к теме наблюдателей и воплотим знакомые интерфейсы в духе функционального программирования, по-новому осветив некоторые понятия. В главе 6 «Введение в программирование потоков событий на языке C++» мы займёмся более сложными вопросами обработки потоков событий с помощью двух промышленных библиотек, использующих подход на основе встроенных предметно-ориентированных языков (англ. Domain-Specific Embedded Language, DSEL).

Тем самым будет подготовлена основа для знакомства с промышленной библиотекой RxCpp и её тонкостями, позволяющими создавать программные продукты современного уровня и профессионального качества. Этой замечательной библиотеке посвящены глава 7 «Введение в модель потоков данных и библиотеку RxCpp» и глава 8 «Ключевые элементы библиотеки RxCpp». В последующих главах рассматривается реактивное программирование графических интерфейсов с использованием библиотеки Qt и некоторых изощрённых операций библиотеки RxCpp.

Последние три главы посвящены вопросам повышенной сложности: шаблонам проектирования реактивных программ, созданию микросервисов на языке C++, обработке ошибок и исключений. Читатель, приступивший к чтению книги, владея лишь классической версией языка C++, к концу книги приобретёт значительный опыт не только в создании реактивных программ, но и в использовании современной версии языка. В силу самой темы этой книги разобрана будет большая часть нововведений из стандарта C++ 17 (доступных на момент написания книги).

Итоги

Из этой главы читатель узнал о некоторых структурах данных, ключевых для реактивной модели программирования. Мы построили их упрощённые реализации, что позволило проиллюстрировать лежащие в их основе понятия. Для начала мы разобрали, как обрабатывать события от графического интерфейса пользователя с помощью функций API системы Windows, с помощью функций библиотеки XLib, а также с использованием библиотек MFC и Qt. Кроме того, мы вкратце рассмотрели, как обрабатываются события в технологиях COM+ и CORBA. Затем был представлен краткий обзор реактивной модели програм-

мирования в целом. Мы определили несколько интерфейсов и сделали набросок их реализации. Наконец, для полноты изложения было показано, как встроить реализации этих интерфейсов в программу с графическим интерфейсом, основанную на библиотеке MFC. Также были изложены основные методологические аспекты данной книги.

В следующей главе будет сделан беглый обзор основных новшеств современного языка C++ (под этим словом будем понимать стандарты C++ 11, 14 и 17) с упором на семантику перемещения, лямбда-выражения, вывод типов, циклы по диапазонам, сочленение функций и умные указатели. Всё это нужно для написания даже простейшего кода в реактивной парадигме.



Глава 2

.....

Современный язык C++ и его ключевые идиомы

Классический язык программирования C++ был стандартизирован в 1998 г., а в 2003 г. вышла новая редакция стандарта, содержащая главным образом небольшие исправления. За поддержкой усложнённых абстракций разработчики обращались к комплексу библиотек Boost (<http://www.boost.org>) и к другим библиотекам с открытой лицензией. Благодаря новой эпохе в развитии стандарта, начавшейся с версии C++ 11, язык получил ряд существенных усовершенствований, и теперь разработчики могут выразить на этом языке множество широко распространённых абстракций, поддерживаемых другими языками, без необходимости использовать сторонние библиотеки. Даже потоки и интерфейсы к файловой системе, которые до сих пор оставались исключительно в ведении библиотек, стали частью стандарта языка. Современный стандарт языка C++ (под этим словосочетанием будем понимать стандарты C++ 11, 14 и 17) добавляет ряд замечательных новых средств и в сам язык, и в его стандартную библиотеку, что фактически делает его наилучшим из имеющихся языков для промышленной разработки программного обеспечения. Разобранные в этой главе новшества составляют минимальный набор средств, которыми программисту необходимо овладеть, чтобы работать с конструкциями реактивного программирования вообще и с библиотекой RxCpp в частности. Основная цель этой главы состоит в том, чтобы осветить лишь наиболее важные нововведения, помогающие реализовывать идеи реактивного программирования, не касаясь при этом особо запутанных тем. Такие конструкции, как лямбда-функции, автоматический вывод типов, ссылки `rvalue`, семантика перемещения и стандартная поддержка параллельного программирования, составляют часть того аппарата, которым, по убеждению авторов данной книги, должен владеть каждый программист на языке C++. В этой главе будут обсуждаться следующие темы:

- ключевые моменты проектирования языка C++ в целом;
- некоторые улучшения языка, помогающие писать более изящный код;
- улучшенное управление памятью на основе ссылок `rvalue` и семантики перемещений;

- улучшенное управление временем жизни объектов на основе различных видов умных указателей;
- параметризация поведения с использованием лямбда-функций;
- обёртка для функций – тип-шаблон `std::function`;
- некоторые прочие нововведения;
- создание итераторов и наблюдателей на основе перечисленных языковых средств.

Принципы проектирования языка C++

В той мере, в которой это напрямую касается разработчиков программного обеспечения, основными принципами, которыми постоянно руководствуются создатели языка C++, были и остаются следующие три:

- абстракция нулевой стоимости – введение абстракций всё более высокого уровня не должно оплачиваться потерей производительности;
- выразительность – для типов данных, определённых пользователем, в первую очередь классов, должен поддерживаться столь же богатый набор выразительных средств, как и для встроенных типов;
- взаимозаменяемость – должно быть возможно подставлять пользовательские типы данных в любые контексты, где ожидается встроенный тип, например в обобщённые структуры данных и алгоритмы.

Разберём вкратце каждый из этих принципов отдельно.

Абстракция нулевой стоимости

Язык программирования C++ с момента своего возникновения помогал программистам создавать код, весьма эффективно использующий возможности процессора, для которого компилируется программа, и в то же время работать на высоком уровне абстракции там, где это необходимо. Разрабатывая средства абстрагирования, разработчики языка всегда старались минимизировать или даже полностью исключить связанные с ними потери производительности. Эту характеристику языка называют «абстракцией нулевой стоимости» или «абстракцией с нулевыми потерями». Единственная заметная плата – это затраты на косвенные (т. е. осуществляемые через указатель) вызовы виртуальных функций. Несмотря на многообразие добавленных в язык возможностей, его создатели строго соблюдали гарантию нулевой стоимости абстракций, присущую языку с момента его возникновения.

Выразительность

Язык C++ помогает разработчикам делать собственные типы или классы столь же выразительными, как и встроенные в язык базовые типы. Это позволяет, например, создавать арифметические типы с произвольной разрядностью и точностью (известные в некоторых языках программирования как `BigInteger` и `BigFloat`), с тем же поведением, что и у встроенных целых и вещественных

типов. В иллюстративных целях ниже показан класс `SmartFloat`, представляющий собой обёртку над действительным числом двойной точности согласно стандарту IEEE и переопределяющий большинство операций, присущих типу `double`. Этот пример кода демонстрирует, как создавать типы данных, подражающие семантике встроенных в язык типов.

```
//---- SmartFloat.cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
class SmartFloat {
    double _value; // завернутое значение
public:
    SmartFloat(double value) : _value(value) {}
    SmartFloat() : _value(0) {}
    SmartFloat(const SmartFloat& other) {_value = other._value;}
    SmartFloat& operator=(const SmartFloat& other) {
        if ( this != &other ) { _value = other._value; }
        return *this;
    }
    SmartFloat& operator=(double value) {
        _value = value; return *this;
    }
    ~SmartFloat() {}
```

В представленном выше фрагменте объявлен класс `SmartFloat`, в который завернуто значение типа `double`, а также определены конструкции и операции присваивания, что позволяет создавать инициализированные объекты. В следующем фрагменте кода определены операции инкремента и декремента. Обе операции представлены в двух формах: префиксной и постфиксной.

```
SmartFloat& operator++() { _value++; return *this; }
SmartFloat operator++(int) // постфиксная операция
{ SmartFloat nu(*this); ++_value; return nu; }
SmartFloat& operator--() { _value--; return *this; }
SmartFloat operator--(int)
{ SmartFloat nu(*this); --_value; return nu; }
```

Этот фрагмент служит чисто иллюстративным целям. При создании реального кода нам пришлось бы проверять переполнение разрядной сетки, чтобы обеспечить надёжность своего изделия. В конце концов, цель создания обёрток состоит именно в повышении надёжности кода!

```
SmartFloat& operator+=(double x) { _value += x; return *this; }
SmartFloat& operator-=(double x) { _value -= x; return *this; }
SmartFloat& operator*=(double x) { _value *= x; return *this; }
SmartFloat& operator/=(double x) { _value /= x; return *this; }
```

В этом фрагменте показано, как реализовывать комбинированные операции присваивания. Здесь мы снова ради краткости кода опустили проверки на

возможный выход значений за допустимые границы. С этой же целью тут не показана генерация или обработка исключений.

```
bool operator>(const SmartFloat& other)
{ return _value > other._value; }
bool operator<(const SmartFloat& other)
{ return _value < other._value; }
bool operator==(const SmartFloat& other)
{ return _value == other._value; }
bool operator!=(const SmartFloat& other)
{ return _value != other._value; }
bool operator>=(const SmartFloat& other)
{ return _value >= other._value; }
bool operator<=(const SmartFloat& other)
{ return _value <= other._value; }
```

Здесь реализованы операции-отношения для сравнения объектов. Наконец, семантика наших объектов как заменителей стандартного типа с плавающей точкой двойной точности реализована следующим образом:

```
operator int() { return _value; }
operator double() { return _value; }
};
```

Для полноты семантики мы реализовали операции преобразования для двух типов, `int` и `double`. Теперь напомним две функции, вычисляющие сумму элементов массива. Первая из них принимает на вход массив значений встро-
енного типа `double`, тогда как вторая ожидает массив объектов нашего клас-
са `SmartFloat`. Нетрудно видеть, что код этих функций одинаков, различаются
лишь использованные в них типы данных. Показанная ниже тестовая програм-
ма позволяет убедиться, что эти функции выдают одинаковые результаты.

```
double Accumulate(double a[], int count){
    double value = 0;
    for( int i=0; i<count; ++i) { value += a[i]; }
    return value;
}

double Accumulate(SmartFloat a[], int count) {
    SmartFloat value = 0;
    for( int i=0; i<count; ++i) { value += a[i]; }
    return value;
}

int main() {
    double x[] = { 10.0,20.0,30,40 };
    SmartFloat y[] = { 10,20.0,30,40 };
    double res = Accumulate(x,4); // вызов с типом double
    cout << res << endl;
    res = Accumulate(y,4); // вызов с типом SmartFloat
    cout << res << endl;
}
```

Таким образом, мы убедились, что язык C++ позволяет создавать выразительные типы данных, полностью копирующие семантику встроенных типов. Выразительность языка делает возможным создание как типов, моделирующих значения, так и типов, подражающих семантике указателей, для чего служат многочисленные методики, поддерживаемые языком. Благодаря перегруженным операциям, операциям преобразования типов, операциям создания объекта по заданному адресу (известной в англоязычной литературе под названием «placement new»), семантике перемещения и иным подобным средствам работа с классами в языке C++ поднята на более высокий уровень, сопоставимый с высоким уровнем современных управляемых, динамических и декларативных языков. Впрочем, с властью приходит ответственность: язык C++ теперь предоставляет ещё больше возможностей «выстрелить себе в ногу».



Взаимозаменяемость

В предыдущем примере показано, как для собственного типа данных определить все операции, характерные для встроенного типа. Ещё одна цель, на которую ориентировались создатели языка C++, – это возможность программировать в обобщённом стиле, то есть писать такой код, в который можно было бы подставлять созданные программистом типы данных наряду с такими встроенными типами, как float, double, int и др.

```
template <class T>
T Accumulate(T a[], int count){
    T value = 0;
    for( int i=0; i<count; ++i) { value += a[i]; }
    return value;
}

int main() {
    // шаблонизированная версия типа SmartFloat
    SmartValue<double> y[] = { 10.0,20.0,30,40 };
    double res = Accumulate(y,4);
    cout << res << endl;
}
```



И Язык C++ поддерживает несколько парадигм, и три описанных выше принципа – лишь часть его концепции. Язык предоставляет средства для создания надёжных предметно-ориентированных типов, позволяющих писать элегантный код. Данная триада принципов, безусловно, сделала классический язык C++ мощным и эффективным. В новых версиях стандарта в язык добавлено множество новых абстракций, облегчающих труд программиста. Однако при этом создатели языка нигде не поступились тремя указанными выше принципами ради прочих возможных выгод. Это оказалось возможным благодаря поддержке языком метапрограммирования, что, в свою очередь, стало следствием полноты механизма шаблонов по Тьюрингу, неожиданной для самих создателей языка. За более подробной информацией о метапрограммировании на шаблонах (англ. template meta programming, TMP) и о полноте по Тьюрингу отсылаем читателя к любимой поисковой системе.

Усовершенствования языка, повышающие качество кода

За последнее десятилетие мир языков программирования претерпел значительные изменения, и эти изменения не могли не отразиться и на новом обличье языка C++. Большинство нововведений в современном стандарте этого языка направлено на поддержку усложнённых абстракций, элементов функционального программирования и параллельной обработки. Многие современные языки обладают сборщиками мусора, возлагая сложность управления памятью на среду выполнения программ. Стандарт языка C++, однако, не предполагает автоматической сборки мусора. Положенная в основу языка C++ гарантия нулевой стоимости абстракций (означающая в том числе и нулевые накладные расходы на неиспользуемую функциональность) и стремление максимизировать эффективность исполняемого кода вынуждают для достижения того же уровня абстракции, что и в языках наподобие C#, Java или Scala, прибегать к особым ухищрениям на этапе компиляции и к приёмам метапрограммирования. Некоторые из этих приёмов описаны в следующих разделах, читатель может также изучить эти темы самостоятельно. Сайт <http://en.cppreference.com> может послужить хорошим источником для всех, кто желает углубить свои знания о языке C++.

Автоматический вывод типов

Компиляторы современного языка C++ способны проделать чрезвычайно полезную работу, самостоятельно выводя типы заданных программистом выражений. Большинство современных языков программирования поддерживает вывод типов, не является исключением и обновлённый язык C++. Данная идея позаимствована из языков функционального программирования, таких как Haskell и ML. Так, механизмы вывода типов стали доступны в языках C# и Scala. Напишем небольшую программу для первого знакомства с выводом типов в языке C++.

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<string> vt = {"first", "second", "third", "fourth"};
    // указать тип явно
    for (vector<string>::iterator it = vt.begin(); it != vt.end(); it++)
        cout << *it << " ";
    // поручить вывод типа компилятору
    for (auto it2 = vt.begin(); it2 != vt.end(); it2++)
        cout << *it2 << " ";
    return 0;
}
```



Ключевое слово `auto` в объявлении переменной означает, что её тип должен быть вычислен компилятором, исходя из выражения, использованного для её инициализации, – в частности, на основе типов функций, входящих в это вы-

ражение. В показанном примере выигрыш от автоматического вывода типа невелик. Однако, по мере того как усложняются объявления переменных, всё существеннее становится выгода от перекладывания вывода типов на компилятор. Всюду далее в этой книге мы будем широко использовать автоматический вывод типов для упрощения кода. Напишем теперь ещё одну программу, чтобы отчетливее продемонстрировать вывод типов в действии.

```
#include <iostream>
#include <vector>
#include <initializer_list>
using namespace std;
int main() {
    vector<double> vtdbl = {0, 3.14, 2.718, 10.00};
    auto vt_dbl2 = vtdbl; // тип будет выведен
    auto size = vt_dbl2.size(); // size_t
    auto &rvec = vtdbl; // ссылка на объект выведенного типа
    cout << size << endl;
    // тип итератора выведет компилятор
    for (auto it = vtdbl.begin(); it != vtdbl.end(); ++it)
        cout << *it << " ";
    // тип переменной it2 выводится как итератор по вектору
    for (auto it2 = vt_dbl2.begin(); it2 != vt_dbl2.end(); ++it2)
        cout << *it2 << " ";
    // это присваивание изменит первый элемент вектора vtdbl
    rvec[0] = 100;
    // пройти по вектору и убедиться, что присваивание сработало
    for (auto it3 = vtdbl.begin(); it3 != vtdbl.end(); ++it3)
        cout << *it3 << " ";
    return 0;
}
```

Этот фрагмент кода демонстрирует, как использовать аппарат вывода типов при разработке на современном языке C++. В язык также добавлено новое ключевое слово, позволяющее запросить тип произвольного выражения. В общем случае эта конструкция имеет вид `decltype(<выражение>)`. Следующий пример программы покажет, как пользоваться данным ключевым словом.

```
#include <iostream>
using namespace std;
int foo() { return 10; }
char bar() { return 'g'; }
auto fancy() -> decltype(1.0f) { return 1; } // тип возврата - float
int main() {
    // тип переменной x такой, как тип возврата функции foo(),
    // а тип переменной y - как тип возврата функции bar()
    decltype(foo()) x;
    decltype(bar()) y;
    cout << typeid(x).name() << endl;
    cout << typeid(y).name() << endl;
    struct A { double x; };
    const A* a = new A();
}
```

```

decltype(a->x) z; // тип переменной z - double
decltype((a->x)) t= z; // тип - double&
cout << typeid(z).name() << endl;
cout << typeid(t).name() << endl;
cout << typeid(decltype(fancy())).name() << endl;
return 0;
}

```

Конструкция `decltype` обрабатывается во время компиляции, она позволяет задать тип переменной по образцу типа какого-либо выражения, также она может использоваться для указания возвращаемого типа функции, как явствует из примера с функцией `fancy()`.

Единообразный синтаксис инициализации

В классической версии языка C++ имелось несколько синтаксических форм инициализации объектов, каждая для своих целей. В современном языке C++ появился единый синтаксис инициализации – в коде из раздела о выводе типов можно увидеть примеры его применения. Язык предоставляет вспомогательные классы, позволяющие разработчикам использовать преимущества единообразной инициализации в своих классах.

```

#include <iostream>
#include <vector>
#include <initializer_list>
using namespace std;
template <class T>
struct Vector_Wrapper {
    std::vector<T> vctr;
    Vector_Wrapper(std::initializer_list<T> l) : vctr(l) {}
    void Append(std::initializer_list<T> l)
    { vctr.insert(vctr.end(), l.begin(), l.end()); }
};
int main() {
    Vector_Wrapper<int> vcw = {1, 2, 3, 4, 5}; // инициализация списком
    vcw.Append({6, 7, 8}); // инициализация аргумента списком
    for (auto n : vcw.vctr) { std::cout << n << ' '; }
    std::cout << '\n';
}

```

В этом примере показано, как применить список инициализации в классе, созданном самим программистом.

Вариадические шаблоны

Начиная с версии стандарта C++ 11 язык поддерживает вариадические шаблоны. Вариадическим называется шаблон класса или функции, имеющий произвольное число параметров. В более ранних стандартах языка C++ шаблон всегда принимал фиксированное число параметров. Хотя вариадичность шаблонов поддерживается как на уровне классов, так и на уровне функций, в этом разделе будем заниматься исключительно вариадическими функциями, поскольку

они широко используются при создании программ в функциональном стиле, для метапрограммирования, то есть для генерации кода во время компиляции, а также при определении операции композиции функций.

```
//Variadic.cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;
// база для рекурсивной компиляции
int add() { return 0; } // условие завершения рекурсии
// вариадический шаблон функции:
// компилятор сгенерирует конкретную функцию,
// исходя из количества и типов аргументов,
// с которыми она вызвана
template<class T0, class ... Ts>
decltype(auto) add(T0 first, Ts ... rest) {
    return first + add(rest ...);
}
int main() {
    int n = add(0,2,3,4);
    cout << n << endl;
    return 0;
}
```

В этом примере компилятор генерирует функции, исходя из числа и типов аргументов, переданных при вызове. Компилятор понимает, что под именем `add` скрывается шаблон функции с переменным числом типов-параметров, и генерирует исполняемый код, рекурсивно разбирая список фактических аргументов. Эта рекурсия, осуществляемая во время компиляции, обрывается, когда обработан будет весь список аргументов, переданных функции при вызове. Для этого служит базовый случай, то есть функция без аргументов. Следующий пример иллюстрирует, как использовать вариадические шаблоны вместе с т. н. «совершенной передачей» (англ. *perfect forwarding*) для создания функции с переменным числом аргументов произвольных типов, сохраняя при этом преимущества строгой типизации.

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;
// База рекурсии: печатать значения простых типов
void EmitConsole(int value)
{ cout << "Integer: " << value << endl; }
void EmitConsole(double value)
{ cout << "Double: " << value << endl; }
void EmitConsole(const string& value)
{cout << "String: "<< value << endl; }
```

Три перегруженных варианта функции `EmitConsole` печатают свой аргумент на консоль. Таким образом, в программе определены функции для печати значений типа `int`, `double` и `std::string`. Используя эти функции в качестве базы рекурсии, создадим функцию, принимающую произвольное число аргументов различных типов по универсальным ссылкам и использующую совершенную передачу этих аргументов в последующие функции¹.

```
template<typename T>
void EmitValues(T&& arg)
{ EmitConsole(std::forward<T>(arg)); }
template<typename T1, typename... Tn>
void EmitValues(T1&& arg1, Tn&&... args) {
    EmitConsole(std::forward<T1>(arg1));
    EmitValues(std::forward<Tn>(args)...);
}
int main() {
    EmitValues(0,2.0,"Hello World",4);
    return 0;
}
```

Ссылки `rvalue`

Программисты, имеющие достаточно большой опыт работы с языком C++, хорошо знают, что ссылки играют роль псевдонимов для переменных и что присваивание нового значения или иная модифицирующая операция над ссылкой вызывает изменение и той переменной, на которую она ссылается. Эта разновидность ссылок, поддерживаемая начиная с первых версий языка C++, относится к категории `lvalue` (от англ. left value), так как эти ссылки потенциально могут стоять в левой части операции присваивания². Следующий пример кода демонстрирует использование ссылок `lvalue`.

```
//---- Lvalue.cpp
#include <iostream>
using namespace std;
int main() {
    int i=0;
    cout << i << endl; // выводит 0
```

¹ Этот пример не вполне иллюстрирует преимущества совершенной передачи, так как функции, использующиеся в качестве базы рекурсии, принимают свои аргументы либо по значению, либо по константной ссылке. Для полноты примера стоило бы пополнить базу рекурсии функциями с аргументом типа ссылки `rvalue`. – *Прим. перев.*

² Для полноты картины следует также упомянуть о константных ссылках. Если параметр функции объявлен с типом `t const&`, где `t` – произвольный тип, то при вызове в него в качестве аргумента может передаваться временный объект (скажем, результат, возвращаемый некоторой функцией), являющийся, очевидно, `rvalue`. Кроме того, ссылка типа `t const&`, хотя и не может находиться в левой части присваивания в силу константности, всё равно считается `lvalue`: в современном языке C++ определяющим признаком для категории выражения является уже не его роль в присваивании, а его способность индивидуализировать объект. – *Прим. перев.*


```

int& ri = i;
ri = 20;
cout << i << endl; // выводит 20
return 0;
}

```

Здесь переменная `ri`, объявленная с типом `int&`, является ссылкой `lvalue`. В современном языке C++ появилось также понятие ссылки `rvalue`. К категории `rvalue`, по определению, относятся те и только те выражения, что не являются `lvalue`; говоря упрощённо, это то, что может употребляться лишь в правой части операции присваивания¹. В современном языке C++, в отличие от классической версии, стало возможным связывать ссылки со значениями `rvalue`.

```

// Rvaluref.cpp
#include <iostream>
using namespace std;
int main() {
    int&& j = 42; int x = 3, y = 5; int&& z = x + y; cout << z << endl;
    z = 10; cout << z << endl; j = 20; cout << j << endl;
}

```

Ссылка `rvalue` обозначается двумя знаками ссылки: `&&`. Следующая программа должна продемонстрировать использование ссылок `rvalue` в качестве аргументов функции.

```

// RvaluerefCall.cpp
#include <iostream>
using namespace std;
void TestFunction( int & a ) {cout << a << endl;}
void TestFunction( int && a ) {
    cout << "rvalue references" << endl;
    cout << a << endl;
}
int main() {
    int&& j = 42;
    int x = 3, y = 5;
    int&& z = x + y;
    TestFunction(x + y ); // вызовется функция с аргументом rvalue
}

```

¹ Такое изложение слишком упрощённо и без дополнительных уточнений может создать у читателя искажённое представление о предмете. Так, константная ссылка, очевидно, не может использоваться в левой части присваивания и тем не менее представляет собой `lvalue`, так как индивидуализирует объект. С другой стороны, если функция `f()` возвращает объект класса, для которого перегружена операция присваивания, то операторы вида `f()=x` (где `x` – объект подходящего типа) вполне законны. Роль выражения в операторе присваивания могла считаться критерием его принадлежности к категориям `rvalue` и `lvalue` лишь в языке C, тогда как в языке C++ трактовка этих понятий изменена. Кроме того, следует напомнить, что в современном языке C++, помимо `rvalue` и `lvalue`, есть ещё три категории: `glvalue`, `xvalue` и `prvalue`. Для профессионального программирования необходимо основательное знакомство с данной темой. – Прим. перев.

```
TestFunction(j); // вызовется функция с аргументом lvalue
return 0;
}
```

Главная выгода от использования ссылок `rvalue` проявляется в связи с управлением памятью. В языке C++ с самого начала поддерживались конструкторы копирования и перегрузка операции присваивания. Обычно они копируют данные из объекта в объект. Однако с помощью ссылок `rvalue` можно избежать трудоёмкого полного копирования из временного объекта, возникшего как промежуточный результат вычисления. Этой теме посвящён следующий раздел.

Семантика перемещения



В языке C++ определяемые программистом классы могут содержать конструкторы копирования, операции присваивания и деструкторы (в том числе виртуальные). Это бывает нужно для корректного управления ресурсами, которыми владеет объект, при его клонировании и при присваивании его существующему объекту¹. Однако копирование завернутых в объект данных часто обходится слишком дорого, и тогда перенос этих данных во владение другого объекта (как правило, с помощью указателей) может значительно ускорить работу программы. Современный язык C++ позволяет разработчикам определить для своих классов конструктор перемещения и перемещающую операцию присваивания, избегая тем самым копирования данных из больших объектов. Ссылки `rvalue` в качестве аргументов методов играют роль подсказок компилятору: следует использовать именно эти, перемещающие версии конструктора копирования или операции присваивания, если указанный при вызове аргумент представляет собой временный объект.

```
//----- FloatBuffer.cpp
#include <iostream>
#include <vector>
using namespace std;
class FloatBuffer {
    double *bfr; int count;
public:
    FloatBuffer():bfr(nullptr),count(0){}
    FloatBuffer(int pcount):
        bfr(new double[pcount]), count(pcount)
    {}
    // Конструктор копирования
```



¹ Для старых версий стандарта языка C++ сформулировано т. н. «правило трёх»: если в классе присутствует хотя бы один из трёх следующих методов (конструктор копирования, перегруженная операция присваивания и деструктор), должны быть определены и остальные. С появлением ссылок `rvalue` в стандарте C++ 11 оно расширилось до «правила пяти»: к перечню методов добавлены конструктор перемещения и перемещающая операция присваивания. – *Прим. перев.*

```

FloatBuffer(const FloatBuffer& other):
    count(other.count), bfr(new double[other.count])
{ std::copy(other.bfr, other.bfr + count, bfr); }
// Копирующая операция присваивания
FloatBuffer& operator=(const FloatBuffer& other) {
    if (this != &other) {
        delete[] bfr; // освободить имеющийся буфер
        count = other.count;
        bfr = new double[count]; // создать новый буфер
        std::copy(other.bfr, other.bfr + count, bfr);
    }
    return *this;
}

// перемещающий конструктор
FloatBuffer(FloatBuffer&& other): bfr(nullptr), count(0) {
    cout << "перемещающий конструктор" << endl;
    // при перемещении не нужно копировать данные из объекта,
    // можно просто забрать себе указатель на буфер
    bfr = other.bfr;
    count = other.count;
    // объект-источник более не владеет этими данными
    other.bfr = nullptr;
    other.count = 0;
}

// перемещающая операция присваивания
FloatBuffer& operator=(FloatBuffer&& other) {
    if (this != &other) {
        // освободить существующий буфер
        delete[] bfr;
        // забрать себе данные из объекта-источника
        bfr = other.bfr;
        count = other.count;
        // объект-источник более не владеет этими данными
        other.bfr = nullptr;
        other.count = 0;
    }
    return *this;
}
};

int main() {
    // Создать вектор объектов и добавить в него элементы.
    // Поскольку библиотека STL поддерживает семантику
    // перемещения, будут вызваны перемещающие операции.
    vector<FloatBuffer> v;
    v.push_back(FloatBuffer(25));
    v.push_back(FloatBuffer(75));
    return 0;
}

```



Функция `std::move`, используемая при передаче аргументов в другие функции, подсказывает компилятору, что объект-аргумент можно перемещать, тогда компилятор подставит вызов подходящего метода (например, перемещающего присваивания или перемещающего конструктора), помогая избежать накладных расходов на копирование данных. В сущности, функция `std::move` представляет собой статическое преобразование операцией `static_cast` к типу ссылки `rvalue`.

Умные указатели

Управление временем жизни объектов в языке C++ всегда было хлопотным делом. Если разработчик окажется недостаточно внимательным, может произойти утечка памяти (т. е. ситуация, когда объект продолжает существовать в памяти, хотя все указатели на него утеряны) или, того хуже, обращение к уже уничтоженному объекту (т. е. обратная ситуация, когда объект уже уничтожен, а программа ещё пытается пользоваться указателем на него). Умные указатели – это классы-обёртки над «сырыми» указателями, обладающие перегруженными операциями разыменования (*) и косвенного обращения (->). Умные указатели могут брать на себя управление временем жизни объекта, вести подсчёт ссылок на него, освобождать память при исчезновении последней ссылки и, таким образом, выполнять роль ограниченного сборщика мусора. В современном языке C++ имеются следующие классы умных указателей:

- `unique_ptr<T>`;
- `shared_ptr<T>`;
- `weak_ptr<T>`.

Класс `unique_ptr<T>` представляет собой такую обёртку над «сырым» указателем, которая реализует исключительное владение объектом. Следующий пример кода демонстрирует использование этого класса.

```
#include <iostream>
#include <deque>
#include <memory>
using namespace std;
int main(int argc , char **argv) {
    // умный указатель на двухстороннюю очередь
    unique_ptr< deque<int> > dq(new deque<int>() );
    // наполнить значениями через перегруженную операцию
    dq->push_front(10);dq->
    dq->push_front(20);
    dq->push_back(23);
    dq->push_front(16);
    dq->push_back(41);

    auto dqiter = dq->begin();
    while ( dqiter != dq->end())
    { cout << *dqiter << "\n"; dqiter++; }
```



```
// выход из области видимости умного указателя влечёт
// вызов его деструктора, который уничтожает объект
return 0;
}
```

Принцип действия второй разновидности умных указателей, `std::shared_ptr`, основан на подсчёте ссылок на каждый объект. Объект уничтожается, когда последний умный указатель на него исчезает или перестаёт указывать на данный объект.

```
#include <iostream>
#include <memory>
#include <stdio.h>
using namespace std;

// объекты типа shared_ptr<T> можно передавать по значению:
// копия этого умного указателя указывает на тот же объект,
// а конструктор копирования увеличивает счётчик ссылок.
void foo_byvalue(std::shared_ptr<int> i) { (*i)++;}


// передача объекта shared_ptr<T> по ссылке не создаёт копию
void foo_byreference(std::shared_ptr<int>& i) { (*i)++; }

int main(int argc, char **argv )
{
    auto sp = std::make_shared<int>(10);
    foo_byvalue(sp);
    foo_byreference(sp);
    // должно быть выведено значение 12
    std::cout << *sp << std::endl;
    return 0;
}
```

Умный указатель `std::weak_ptr` тоже представляет собой обёртку над «сырым» указателем, однако объектом не владеет. Он создаётся как копия указателя `shared_ptr`. Сколько бы ни существовало одновременно копий указателя `weak_ptr`, это никак не влияет на «родительский» указатель `shared_ptr` и его копии. Когда все указатели `shared_ptr` на некоторый объект уничтожаются, уничтожается и сам объект, тогда все указывающие на него указатели `weak_ptr` становятся пустыми. Следующая программа демонстрирует подход, с помощью которого можно распознать пустой указатель.

```
#include <iostream>
#include <deque>
#include <memory>

using namespace std;
int main( int argc , char **argv )
{
    std::shared_ptr<int> ptr_1(new int(500));
    std::weak_ptr<int> wptr_1 = ptr_1;
```



```

{
    std::shared_ptr<int> ptr_2 = wptr_1.lock();
    if(ptr_2)
    {
        cout << *ptr_2 << endl; // будет выполнено
    }
    // выход и области видимости объекта ptr_2
}

ptr_1.reset(); // объект уничтожается

std::shared_ptr<int> ptr_3= wptr_1.lock();
if(ptr_3)
    cout << *ptr_3 << endl;
else
    cout << "Указатель пуст" << endl;

return 0;
}

```

В более старых версиях стандарта языка C++ вместо этих трёх был единственный тип умного указателя под названием `auto_ptr`, из последующих версий стандарта он исключён. Вместо него следует использовать тип `unique_ptr`.

Лямбда-функции

Одно из важнейших нововведений, появившихся в языке C++, – это лямбда-функции или лямбда-выражения. Это анонимные функции, которые можно определять непосредственно в том месте, где они используются, что упрощает устройство программы, код становится заметно яснее и изящнее.

Вместо того чтобы давать строгое определение лямбда-функции, напомним пример кода, который подсчитывает количество положительных чисел в контейнере `vector<int>`. Функция `count_if` из библиотеки STL делает то, что нужно: в заданном диапазоне итераторов подсчитывает элементы, удовлетворяющие некоторому условию.

```

// LambdaFirst.cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    auto num_vect = vector<int>{ 10, 23, -33, 15, -7, 60, 80};
    // лямбда-функция для распознавания положительных чисел
    auto filter = [](int const value) {return value > 0; };
    auto cnt= count_if(
        begin(num_vect), end(num_vect), filter);
    cout << cnt << endl;
    return 0;
}

```

В этом фрагменте кода переменной `filter` в качестве значения присваивается анонимная функция, затем она подаётся как аргумент в функцию `count_if` из стандартной библиотеки. Теперь напишем ещё одну простую лямбда-функцию, которую, однако, определим непосредственно в том месте, где она используется. Воспользуемся функцией `accumulate` из библиотеки STL, чтобы вычислить сумму всех элементов контейнера.

```
//----- LambdaSecond.cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;
int main() {
    auto num_vect =
        vector<int>{ 10, 23, -33, 15, -7, 60, 80};
    // определение бинарной операции поместить в аргумент
    auto accum = std::accumulate(
        std::begin(num_vect), std::end(num_vect), 0,
        [](auto const s, auto const n) {return s + n;});
    cout << accum << endl;
}
```



Функциональные объекты и лямбда-функции

При программировании на классических версиях языка C++, особенно при использовании библиотеки STL (например, для фильтрации контейнеров по условию или свёртки контейнеров по некоторой операции) широко использовались функциональные объекты, то есть объекты классов, обладающих перегруженной операцией функционального применения. Ниже следует пример кода.

```
// LambdaThird.cpp
#include <iostream>
#include <numeric>
using namespace std;

// функциональные объекты для сложения и умножения чисел
template <typename T>
struct addition{
    T operator () (T init, T a) { return init + a; }
};
template <typename T>
struct multiply {
    T operator () (T init, T a) { return init * a; }
};

int main()
{
    double v1[3] = {1.0, 2.0, 4.0};
```



```

auto sum = accumulate(v1, v1 + 3, 0.0, addition<double>());
cout << "сумма = " << sum << endl;
sum = accumulate(v1, v1+3, 0.0,
    [] (double a, double b) { return a +b; });
cout << "сумма = " << sum << endl;
auto mul = accumulate(v1, v1 + 3, 1.0, multiply<double>());
cout << "произведение = " << mul << endl;
mul = accumulate(v1, v1+3, 1,
    [] (double a, double b) { return a *b; });
cout << "произведение = " << mul << endl;
return 0;
}

```

Следующая программа иллюстрирует использование лямбда-функций в простейшем алгоритме сортировки. Сначала покажем, как реализовать сортировку с помощью обычных функций, затем напомним эквивалентный код на основе лямбда-функций. Этот код написан в обобщённом стиле: тип элементов массива сделан параметром шаблона; код опирается на предположение, что это либо встроенный в язык числовой тип, либо пользовательский тип, для которого определена операция сравнения.

```

// LambdaFourth.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// обобщённые функции для сравнения и обмена
template <typename T>
bool Cmp(T const& a , T const& b) { return (a > b); }
template <typename T>
void Swap(T& a, T& b) { T c = a; a = b; b = c; }

```

Шаблоны функций Cmp и Swap понадобятся ниже для сравнения двух элементов массива и их обмена.

```

template <class T>
void SelectionSort(T *arr, int length) {
    for (int i = 0; i < length-1; ++i)
        for (int j = i+1; j < length; ++j)
            if (Cmp(arr[i], arr[j]))
                Swap(arr[i], arr[j]);
}

```

Имея в руках функции Cmp и Swap, можно без труда написать алгоритм сортировки выбором. Нужно всего лишь сравнивать первый элемент ещё не отсортированной части массива с каждым из остающихся элементов и, если функция сравнения Cmp вернёт значение «истина», менять эти элементы местами с помощью функции Swap.

```

template <typename T>
void SelectionSortLambda(T *arr, int length) {
    auto CmpLambda = [] (const auto& a, const auto& b)

```




```

        { return (a > b); };
    auto SwapLambda = [] (auto& a , auto& b)
    { auto c = a; a = b; b = c; };
    for (int i = 0; i < length-1; ++i)
        for (int j = i + 1; j < length; ++j)
            if (CmpLambda(arr[i], arr[j]))
                SwapLambda(arr[i], arr[j]);
}

```



В этом фрагменте сравнение и обмен элементов массива оформлены в виде лямбда-функций. Таким образом, механизм лямбда-функций позволяет разместить небольшой блок исполняемого кода в теле функции непосредственно в том месте, где он нужен, и далее обращаться с ним как с функцией. В определении тела лямбда-функции используется тот же синтаксис, что и в обычных функциях. Лямбда-функцию можно присвоить переменной в качестве значения, передать в другую функцию в качестве аргумента или вернуть из функции в качестве её результата. В этом примере значениями переменных `CmpLambda` и `SwapLambda` становятся анонимные функции, реализация которых почти не отличается от обычных функций `Cmp` и `Swap`, показанных выше. Более подробные сведения о лямбда-функциях читатель может найти на странице <http://en.cppreference.com/w/cpp/language/lambda>.

```

int main( int argc , char **argv ){
    double ar1[4] = { 20, 10, 15, -41 };
    SelectionSort(ar1, 4);
    for (int i = 0; i != 4; ++i)
        cout << ar1[i] << "\n";

    double ar2[4] = { 20, 10, 15, -41 };
    SelectionSortLambda(ar2, 4);
    for (int i = 0; i != 4; ++i)
        cout << ar2[i] << "\n";

    return 0;
}

```



Последний фрагмент кода демонстрирует применение двух алгоритмов сортировки.

Композиция, карринг и частичное применение функций

Одно из преимуществ лямбда-функций состоит в том, что они позволяют соединять две функции воедино, т. е. дают возможность выразить композицию функций в строго математическом смысле (читателю рекомендуется прочесть подробнее о композиции функций в контексте математики и функционального программирования, воспользовавшись любимой поисковой системой). Следующая программа иллюстрирует эту идею. Здесь показана весьма упрощённая реализация, создание полностью универсальной версии выходит за рамки данной главы.

```
//----- Compose.cpp
//----- g++ -std=c++ Compose.cpp
#include <iostream>
using namespace std;

// базовый случай для рекурсивной компиляции
template <typename F, typename G>
auto Compose(F&& f, G&& g)
{ return [=](auto x) { return f(g(x)); }; }

// рекурсия по аргументам во время компиляции
template <typename F, typename... R>
auto Compose(F&& f, R&&... r){
    return [=](auto x) { return f(Compose(r...)(x)); };
}
```

Функция `Compose` представляет собой вариадический шаблон функции. Компилятор будет рекурсивно генерировать код, разбирая последовательность аргументов, пока не обработает их все. В этом коде использована конструкция `[=]`, которая указывает компилятору захватить значения всех переменных, доступных в текущей области видимости, и включить их в замыкание лямбда-функции. Читателю рекомендуется узнать побольше о замыканиях и захвате переменных в приложении к функциональному программированию. В языке C++ поддерживаются способы захвата переменных по значению и по ссылке (обозначается `&`), также можно в явном виде перечислять имена захватываемых переменных (например, `[var1, =var2]`).

Парадигма функционального программирования основывается на математической теории, называемой лямбда-исчислением, начала которой были заложены американским математиком Алонзо Чёрчем. В лямбда-исчислении рассматриваются исключительно унарные функции; для работы с функциями нескольких аргументов применяется приём, называемый каррингом (в честь математика Хаскелла Карри), состоящий в преобразовании их в функции одного аргумента¹. Воспользовавшись лямбда-функциями и чуть поменяв способ записи функций, можно симитировать карринг и на языке C++.

¹ Понятие карринга заслуживает более развёрнутого комментария. Рассмотрим функцию f от двух аргументов. Тогда смысл выражения $f(x, y)$ таков: это значение функции f на конкретных значениях аргументов (первый аргумент равен x , второй – y). Для карринга необходимо посмотреть на функцию f под другим углом. Пусть значение первого аргумента фиксировано и равно x , а значение второго не определено. Тогда функция f от двух аргументов превращается в функцию $f(x)$ от одного оставшегося аргумента. Иными словами, $f(x)$, где x – произвольное, но фиксированное значение, есть результат частичного применения функции f (имеющей два аргумента) к первому аргументу. Зададим теперь некоторым значением y второго аргумента, тогда получим $f(x, y) = (f(x))(y)$, то есть результат применения к единственному аргументу y функции $f(x)$. Тем самым функция f предстаёт как функция типа $X \rightarrow (Y \rightarrow Z)$, значением которой на единственном аргументе является, в свою очередь, функция одного оставшегося аргумента. – *Прим. перев.*

```

auto CurriedAdd3(int x) {
    return [x](int y) { // захват переменной x
        return [x, y](int z){ return x + y + z; };
    };
};

```

Частичное применение функций преобразовывает функцию от некоторого числа аргументов в функцию от меньшего их числа. Если функции подано меньше аргументов, чем она ожидает, результатом частичного применения становится функция, ожидающая оставшихся аргументов. Когда и оставшиеся аргументы получают определённые значения, функция сможет выработать определённое значение. С точки зрения программирования частичное применение функции можно трактовать как некую разновидность кеширования аргументов, пришедших первыми, до тех пор, пока не поступят остальные.

В следующих фрагментах кода используются такие конструкции, как вариадические шаблоны и пакеты параметров. Пакет параметров шаблона – это параметр шаблона, который принимает ноль или более аргументов шаблона (аргументов-значений, аргументов-типов или аргументов-шаблонов). Пакет параметров функции – это параметр функции, который может принимать ноль или более аргументов функции. Шаблон, обладающий хотя бы одним пакетом параметров, называется вариадическим шаблоном. Хорошее понимание пакетов параметров и вариадических шаблонов необходимо для понимания конструкций наподобие `sizeof...` в следующем коде.

```

template <typename... Ts>
auto PartialFunctionAdd3(Ts... xs) {
    static_assert(sizeof...(xs) <= 3);

    if constexpr (sizeof...(xs) == 3){
        // Base case: evaluate and return the sum.
        return (0 + ... + xs);
    }
    else{
        // Recursive case: bind 'xs...' and return another
        return [xs...](auto... ys){
            return PartialFunctionAdd3(xs..., ys...);
        };
    }
}

int main() {
    // композиция двух функций
    auto val = Compose(
        [](int const a) { return std::to_string(a); },
        [](int const a) { return a * a; })(4); // val = "16"
    cout << val << std::endl; // напечатает 16

    // вызов каррированной функции
    auto p = CurriedAdd3(4)(5)(6);
}

```

```

cout << p << endl;

// композиция многих функций
auto func = Compose(
    [](int const n) { return std::to_string(n); },
    [](int const n) { return n * n; },
    [](int const n) { return n + n; },
    [](int const n) { return std::abs(n); });
cout << func(5) << endl;

// частичное применение функции
PartialFunctionAdd3(1, 2, 3);
PartialFunctionAdd3(1, 2)(3);
PartialFunctionAdd3(1)(2)(3);

return 0;
}

```

Обёртки над функциями

Классы-обёртки над функциями позволяют заключать функции, функциональные объекты и лямбда-функции в объекты, допускающие копирование. Тип класса-обёртки зависит от прототипа завернутой в него функции. Шаблон `std::function<прототип>` из заголовочного файла `<functional>` и есть универсальный класс-обёртка.

```

// FuncWrapper.cpp, требуется C++ 17 (-std=c++1z )
#include <functional>
#include <iostream>
using namespace std;

// просто функция
void PrintNumber(int val){ cout << val << endl; }

// класс с перегруженной операцией вызова
struct PrintNumber {
    void operator()(int i) const { std::cout << i << '\n'; }
};

// понадобится для вызова метода
struct FooClass {
    int number;
    FooClass(int pnum) : number(pnum){}
    void PrintNumber(int val) const {
        std::cout << number + val << endl;
    }
};

int main() {
    // обёртка над обычной функцией
    std::function<void(int)> displaynum = PrintNumber;
    displaynum(0xF000);
}

```

```
// вызов посредством std::invoke
std::invoke(displaynum,0xFF00);

// обёртка над лямбда-функцией
std::function<void()> lambdaprint = []()
    { PrintNumber(786); };
lambdaprint();
std::invoke(lambdaprint);

// обёртка над методом класса
std::function<void(const FooClass&, int)>
    classdisplay = &FooClass::PrintNumber;
// создать экземпляр
const FooClass fooinstance(100);
classdisplay (fooinstance,100);
}
```



В дальнейшем мы будем широко пользоваться классом-обёрткой `std::function`, чтобы обращаться с функциями, как с данными.

ОПЕРАЦИЯ КОМПОЗИЦИИ ФУНКЦИЙ

Стандартная оболочка командной строки в операционных системах семейства Unix позволяет перенаправлять вывод одной команды на вход другой, выстраивая таким образом сколь угодно длинные цепочки программ-фильтров. Когда при создании кода в функциональном стиле из относительно простых функций строятся более сложные, код быстро становится сложным для понимания из-за глубокой вложенности. Именно по этой причине название функционального языка Lisp, первоначально означавшее «обработка списков» (list processing), в шутку расшифровывают как Lots of Irritating and Silly Parentheses (множество раздражающих и дурацких скобок). Однако теперь, благодаря новым возможностям языка C++, можно перегрузить операцию `|`, превратив её в композицию функций, т. е. операцию, сочленяющую две функции воедино – подобно тому, как сочленяются команды в командной оболочке системы Unix или в консоли PowerShell системы Windows. Операция композиции `|` широко используется в библиотеке RxCpp для соединения функций между собой. Следующий код поможет разобраться, как функции соединяются в цепочки. Этот код, впрочем, лишь демонстрирует, как это вообще возможно, и годится только для учебных целей.

```
#include <iostream>
using namespace std;

struct AddOne {
    template<class T>
    auto operator()(T x) const { return x + 1; }
};

struct SumFunction {
```



```
// функция двух аргументов
template<class T>
auto operator()(T x, T y) const { return x + y;}
};
```

В показанном выше фрагменте кода объявлен набор классов с перегруженной операцией функционального применения, которые будут использованы для построения цепочек функций. Теперь нужен механизм, позволяющий превращать произвольные функции в замыкания.

```
// унарная функция с замыканием,
// использован пакет параметров вариативного шаблона
template<class F>
struct PipableClosure : F{
    template<class... Xs>
    PipableClosure(Xs&&... xs) : // Xs - универсальная ссылка
    F(std::forward<Xs>(xs)...) // совершенная передача
    {}
};
```



```
// преобразователь функции в замыкание
template<class F>
auto MakePipeClosure(F f) {
    return PipableClosure<F>(std::move(f));
}
```

```
// замыкание для функции двух аргументов
template<class F>
struct PipableClosureBinary {
    template<class... Ts>
    auto operator()(Ts... xs) const {
        return MakePipeClosure(
            [=](auto x) -> decltype(auto)
            { return F()(x, xs...);});
    }
};
```

```
// операция композиции,
// использована совершенная передача
template<class T, class F>
decltype(auto) operator|(T&& x, const PipableClosure<F>& pfn)
{
    return pfn(std::forward<T>(x));
}
```



```
int main() {
    // замыкание унарной функции
    const PipableClosure<AddOne> fnclosure = {};
    int value = 1 | fnclosure | fnclosure;
    std::cout << value << std::endl;

    // замыкание функции двух аргументов
    const PipableClosureBinary<SumFunction> sumfunction = {};
```

```

int value1 = 1 | sumfunction(2) | sumfunction(5) | fnclosure;
std::cout << value1 << std::endl;
}

```

В главной функции создаётся экземпляр шаблонного класса `PipableClosure`, в котором в качестве параметра подставлен класс `AddOne`, обладающий поведением унарной функции; затем строится цепочка из исходного значения и двух применений этой функции. Этот фрагмент кода должен вывести на печать число 3. Далее создаётся экземпляр класса `PipableClosureBinary` и составляется цепочка из функций как двух, так и одного аргумента.



ПРОЧИЕ ВОЗМОЖНОСТИ ЯЗЫКА

В предыдущих разделах разобраны наиболее важные семантические новшества, появившиеся начиная со стандарта C++ 11. Цель всей этой главы состоит в том, чтобы осветить языковые средства, помогающие создавать выразительный и современный программный код. В стандарте C++ 17 появился ещё ряд полезных возможностей. Расскажем здесь о некоторых из них.

Выражения-свёртки

В стандарт C++ 17 добавлена поддержка выражений-свёрток, упрощающих генерацию вариативных функций. Компилятор выполняет сопоставление с образцом и генерирует код, как бы отгадывая намерение программиста. Следующий фрагмент кода демонстрирует эту идею.

```

// Folds.cpp
// Требуется поддержка C++ 17 (-std=c++1z)
#include <functional>
#include <iostream>

using namespace std;
template <typename... Ts>
auto AddFoldLeftUn(Ts... args) { return (... + args); }
template <typename... Ts>
auto AddFoldLeftBin(int n, Ts... args){ return (n + ... + args);}
template <typename... Ts>
auto AddFoldRightUn(Ts... args) { return (args + ...); }
template <typename... Ts>
auto AddFoldRightBin(int n, Ts... args) {
    return (args + ... + n);
}
template <typename T, typename... Ts>
auto AddFoldRightBinPoly(T n, Ts... args) {
    return (args + ... + n);
}
template <typename T, typename... Ts>
auto AddFoldLeftBinPoly(T n, Ts... args) {
    return (n + ... + args);
}

int main() {

```



```

auto a = AddFoldLeftUn(1,2,3,4);
cout << a << endl;
cout << AddFoldRightBin(a,4,5,6) << endl;

// свёртка справа налево
auto b = AddFoldRightBinPoly(
    "C++ "s,"Hello "s,"World "s );
cout << b << endl;

auto c = AddFoldLeftBinPoly(
    "Hello "s,"World "s, "C++ "s);
cout << c << endl;
}

```



Эта программа должна напечатать следующий текст:

```

10
25
Hello World C++
Hello World C++

```

Сумма типов: тип `variant`

Несколько заумное определение типа `std::variant` звучит как «объединение (union), безопасное с точки зрения типов»¹. В качестве параметра этому шаблону можно передать сколь угодно длинный перечень типов. Тогда в каждый момент времени объект будет содержать значение какого-либо одного из этих типов-аргументов. При попытке «достать» из объекта `std::variant` значение не того типа, который в нём в настоящее время содержится, будет выброшено исключение типа `std::bad_variant_access`. В следующем примере кода это исключение не обрабатывается.

```

#include <variant>
#include <string>
#include <cassert>
#include <iostream>
using namespace std;
int main(){
    std::variant<int, float,string> v, w;
    v = 12.0f; // содержит значение типа float
    cout << std::get<1>(v) << endl;
    w = 20; // присвоить значение типа int
    cout << std::get<0>(w) << endl;
    w = "hello"s; // присвоить строку
    cout << std::get<2>(w) << endl;
}

```

¹ Куда более любопытно определение в терминах теории категорий как копроизведения. Интересующийся читатель может легко найти информацию об этом в сети. – *Прим. перев.*

Прочее

В новом стандарте языка C++ появилась также поддержка многопоточного и параллельного программирования, гарантий памяти, асинхронного выполнения – этим вопросам посвящены следующие две главы. Также в языке появился тип `std::optional` (обёртка для значения, которое может отсутствовать) и `std::any` (обёртка для значения любого типа). Одним из важнейших нововведений стало добавление параллельных версий для большинства алгоритмов из библиотеки STL.

Циклы по диапазонам и наблюдатели

В этом разделе будет показано, как создать собственный класс, объекты которого можно подставлять в цикл по диапазону; этот пример проиллюстрирует, как все изученные выше языковые средства применять совместно для создания современного, идиоматичного кода. Реализуем класс, предстающий для клиента как последовательность чисел из определённого интервала, а вместе с ним реализуем ряд вспомогательных приспособлений, поддерживающих итерацию по этим значениям с помощью цикла по диапазону. Сперва напишем версию, основанную на понятии итератора (или перечислителя) и поддерживающую циклы по диапазонам. Затем, путём некоторых ухищрений, преобразуем её в реализацию, основанную на идее наблюдателя, т. е. на основном интерфейсе реактивного программирования. Впрочем, представленная здесь реализация наблюдателя годится лишь для иллюстративных целей и не претендует на уровень промышленного решения.

В приведённом ниже коде класс `iterable` сделан вложенным в класс пере-
числимого интервала.

```
// объекты можно использовать в конструкциях наподобие
// for (auto l : EnumerableRange<5, 25>())
// { std::cout << l << ' '; }
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>
using namespace std;

template<long Start, long End>
class EnumerableRange {
public:
    class iterable : public std::iterator<
        std::input_iterator_tag, // category (категория)
        long,                    // value_type (тип значения)
        long,                    // difference_type (тип разности)
        const long*,             // pointer type (тип указателя)
        long>                    // reference type (тип ссылки)
    {
```



```

    long current_num = Start;
public:
    reference operator*() const { return current_num; }
    explicit iterable(long val = 0): current_num(val) {}
    iterable& operator++() {
        current_num = (End >= Start
            ? current_num + 1
            : current_num - 1;
        return *this;
    }
    iterable operator++(int) {
        iterable retval = *this;
        ++(*this);
        return retval;
    }
    bool operator==(iterable other) const
        { return current_num == other.current_num; }
    bool operator!=(iterable other) const
        { return !(*this == other); }
};

```

В показанном выше фрагменте кода объявлен класс-шаблон `EnumerableRange` с двумя параметрами-значениями (нижняя и верхняя границы диапазона) и вложенный в него класс `iterable`, порождённый от стандартного класса-шаблона `std::iterator` – последнее необходимо для того, чтобы данный тип можно было использовать в конструкции цикла по диапазону (англ. *range-based for*). Теперь нужно определить два открытых метода, `begin` и `end`, которые также необходимы клиентскому коду для прохода в цикле по такой последовательности.

```

    iterable begin() { return iterable(Start); }
    iterable end()
        { return iterable(End >= Start ? End + 1 : End - 1); }
};

```

Таким образом, объявленный выше класс можно использовать в конструкциях вида

```

for (auto l : EnumerableRange<5, 25>())
    { std::cout << l << ' '; }

```

В предыдущей главе был объявлен интерфейс `IEnumerable<T>`. Наш замысел состоял в том, чтобы как можно точнее придерживаться документации к библиотеке `Reactive eXtensions`. Классы, представленные здесь, весьма похожи на реализацию интерфейсов `IEnumerable<T>` и `IEnumerator<T>` из предыдущей главы. Как уже говорилось ранее, семантику вытягивания можно преобразовать в семантику вталкивания, если немного переработать код. Создадим класс наблюдателя `Observer`, обладающий тремя методами. В определениях этих методов воспользуемся классом-обёрткой над функциями из стандартной библиотеки.

```
struct Observer {
    std::function<void(const long&)> ondata;
    std::function<void()> oncompleted;
    std::function<void(const std::exception &)> onexception;
};
```



Класс `ObservableRange`, показанный ниже, хранит в себе перечень подписчиков в виде контейнера `std::vector`. Всякий раз, когда источник генерирует очередное значение из диапазона, все подписчики оповещаются о событии. Если вызывать обработчики событий из асинхронного метода, потребитель окажется хорошо изолирован от источника данного потока значений. Классы в этом примере не реализуют интерфейсы `Iobservable` и `Iobserver`, но они всё равно позволяют подписываться на события с помощью метода `subscribe`.

```
template<long Start, long End>
class ObservableRange {
private:
    // контейнер наблюдателей
    std::vector<std::pair<const Observer&, int>> _observers;
    int _id = 0;
```

Подписчиков будем хранить в контейнере в виде пар. Первым компонентом пары будет ссылка на наблюдателя, а вторым – целое число, однозначно идентифицирующее подписчика¹. С помощью этих идентификаторов наблюдатели могут отписываться от источника событий.

```
class iterable : public std::iterator<
    std::input_iterator_tag, // category (категория)
    long,                    // value_type (тип значения)
    long,                    // difference_type (тип разности)
    const long*,             // pointer type (тип указателя)
    long>                    // reference type (тип ссылки)
{
    long current_num = Start;
public:
    reference operator*() const { return current_num; }
    explicit iterable(long val = 0) : current_num(val) {}
    iterable& operator++() {
        current_num = ( End >= Start
            ? current_num + 1
            : current_num - 1;
        return *this;
    }
    iterable operator++(int) {
        iterable retval = *this;
        ++(*this);
        return retval;
    }
};
```



¹ Вероятно, лучшим решением был бы ассоциативный контейнер `std::map`. – Прим. перев.

```

    bool operator==(iterable other) const
    { return current_num == other.current_num; }
    bool operator!=(iterable other) const
    { return !(*this == other); }
};

iterable begin()
{ return iterable(Start); }
iterable end()
{ return iterable(End >= Start ? End + 1 : End - 1); }

// генерировать значения из диапазона;
// этот метод следовало бы вызывать с помощью std::async
void generate_async() {
    auto& subscribers = _observers;
    for (auto l : *this)
        for (const auto& obs : subscribers) {
            const Observer& ob = obs.first;
            ob.ondata(l);
        }
}

public:
    // генерация последовательности; метод generate_async лучше
    // вызывать через std::async, чтобы сразу вернуть управление
    void generate() { generate_async(); }

    // подписка наблюдателей
    virtual int subscribe(const Observer& call) {
        _observers.emplace_back(call, ++_id);
        return _id;
    }

    // отписка наблюдателя не реализована для краткости примера
    virtual void unsubscribe(const int subscription) {}
};

int main() {
    // воспользоваться циклом по диапазону
    for (long l : EnumerableRange<5, 25>())
        { std::cout << l << ' '; }
    std::cout << endl;

    // создать источник - экземпляр класса ObservableRange
    auto j = ObservableRange<10, 20>();
    // создать наблюдателя
    Observer test_handler;
    test_handler.ondata = [=](const long & r)
        { cout << r << endl; };
    // подписать наблюдателя на события от источника
    int cnt = j.subscribe(test_handler);
}

```

```
j.generate(); // запустить генератор  
  
return 0;  
}
```

Итоги

В этой главе были изучены новые и усложнённые возможности языка C++, которыми нужно свободно владеть, чтобы создавать программы в реактивном стиле (да и вообще любые современные программы). Рассказано о выводе типов, вариативных шаблонах (т. е. шаблонах с переменным числом параметров), ссылках rvalue, семантике перемещения, лямбда-функциях, основах функционального программирования, композиции функций, показаны примеры реализации итераторов и источников событий. В следующей главе читатель узнает о средствах параллельного программирования, поддерживаемых современным стандартом языка C++.



Параллельное и многопоточное программирование на языке C++

С выходом стандарта C++ 11 в языке появилась превосходная поддержка параллельного программирования. Ранее для управления потоками приходилось пользоваться библиотеками, специфическими для определённой платформы. Корпорация Microsoft разработала собственный прикладной интерфейс для управления потоками, тогда как многие другие платформы (такие как GNU Linux или macOS) следуют модели потоков, определённой в стандарте POSIX. Закрепление средств многопоточного программирования на уровне стандарта языка помогает программистам создавать переносимый код, одинаково хорошо работающий на различных платформах.

Когда в 1998 г. вышел первоначальный стандарт языка C++, комитет, занимающийся разработкой языка, был твёрдо убеждён, что управление потоками, файловые системы, графические интерфейсы пользователя и многое другое лучше оставить на долю библиотек, зависящих от конкретной платформы. Герб Саттер (Herb Sutter) опубликовал в «Журнале д-ра Добба» (Dr. Dobbs Journal) статью под заглавием «Бесплатного супа больше не будет» (The Free Lunch Is Over), оказавшую значительное влияние на профессиональное сообщество, в которой пропагандировал такие средства программирования, которые позволяют извлекать максимум пользы из набиравших тогда популярность многоядерных процессоров. Задаче распараллеливания программ вполне отвечает модель функционального программирования. Такие языковые средства, как потоки, лямбда-функции, семантика перемещения и гарантии памяти, помогают разработчикам создавать многопоточный параллельный код без лишних хлопот. Цель этой главы состоит в том, чтобы рассказать разработчикам о биб-

лиотечных средствах многопоточного программирования и дать рекомендации по их использованию.

В этой главе будут освещены следующие вопросы:

- что такое параллельное программирование;
- как написать многопоточную программу Hello World;
- как управлять временем жизни и ресурсами потоков;
- обмен данными между потоками;
- как структуру данных сделать потокобезопасной.

ЧТО ТАКОЕ ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Говоря упрощённо, параллельность означает способность выполнять более одного действия одновременно. Понятие параллельности можно приложить ко множеству ситуаций из нашей повседневной жизни: например, к поеданию воздушной кукурузы во время просмотра фильма или к пользованию обеими руками одновременно для двух различных действий и т. д. Однако что же параллельность значит для компьютера?

Компьютерные системы обрели способность переключать задачи уже много десятилетий назад, и долгое время существуют многозадачные операционные системы. Чем вызвана внезапная повторная вспышка интереса к параллельным вычислениям? Производители микропроцессоров постоянно наращивали их вычислительную мощность, помещая всё больше транзисторов на кремниевый кристалл. На очередном этапе этой гонки дальнейшее наращивание плотности размещения элементов оказалось уже невозможным из-за фундаментальных физических ограничений. Процессоры той эпохи обладали единственным потоком выполнения команд, а выполнение нескольких потоков достигалось за счёт переключения между потоками. С точки зрения внутреннего устройства процессора, в каждый момент времени выполнялся лишь один поток, но поскольку переключение между потоками происходило весьма часто и быстро (по меркам человеческого восприятия), у пользователей создавалось впечатление, что несколько программ выполняется в самом деле одновременно.

Около 2005 г. корпорация Intel объявила о выходе нового многоядерного процессора (т. е. процессора, на схемотехническом уровне способного действительно выполнять несколько потоков команд одновременно), что в корне изменило картину. Вместо того чтобы нагружать единственный процессор всеми задачами, поочерёдно переключаясь между ними, многоядерный процессор дал возможность в самом деле выполнять их параллельно. Но это поставило перед программистами новую задачу: писать свои программы так, чтобы использовать эту предоставляемую аппаратурой возможность. Кроме того, проявилось ещё одно затруднение: истинная, поддерживаемая аппаратурой параллельная обработка имеет ряд отличий от иллюзии, создаваемой переключением между задачами. До появления многоядерных процессоров производители аппаратуры соревновались в наращивании тактовой частоты; если бы тенденция оставалась неизменной, отметка в 10 ГГц была бы достигнута

до конца первого десятилетия XXI века. Однако ориентир в гонке производительности резко сменился, и теперь производители процессоров наращивают количество ядер при относительно неизменной частоте. Как писал Герб Саттер в статье «Бесплатного супа больше не будет» (<http://www.gotw.ca/publications/concurrency-ddj.htm>), «приложения должны стать параллельными, если вы хотите на все 100 % использовать растущую пропускную способность процессоров, которые уже начали появляться на рынке и будут править бал на нем в последующие несколько лет»¹. Автор статьи предупреждает программистов, что даже те, кто ранее не видел необходимости в параллельных вычислениях, должны теперь считаться с этим при разработке программ.

Современная стандартная библиотека языка C++ предоставляет набор инструментов для поддержки многопоточного и параллельного программирования. Это, во-первых, класс `std::thread` вместе с объектами синхронизации, такими как `std::mutex`, `std::lock_guard`, `std::unique_lock`, `std::condition_variable` и др., – всё это вместе позволяет программистам писать многопоточный код, оставаясь в пределах стандарта. Во-вторых, для поддержки параллельности на основе задач (как в платформе .Net и языке Java) в язык C++ введены классы `std::future` и `std::promise`, работающие всегда в паре и позволяющие отделить вызов функции от ожидания её результата.

Наконец, для борьбы с лишними затратами на управление потоками в стандартную библиотеку включён класс `std::async`, который мы подробно разберём в следующей главе, посвящённой разработке неблокирующих параллельных программ (по меньшей мере, минимизирующих блокировки, насколько возможно).



Многопоточность означает, что два или более потоков вычислений могут стартовать, выполняться и завершаться в перекрывающиеся отрезки времени (как в модели с разделением времени). Параллельность означает, что две (или более) задачи могут выполняться на самом деле одновременно (скажем, на разных ядрах одного процессора). Многопоточность главным образом позволяет улучшить время отклика системы, тогда как параллельность даёт возможность наиболее эффективно использовать ресурсы системы.

Здравствуй, мир потоков!

Давайте же начнём создавать свою первую программу, основанную на классе `std::thread`. Чтобы компилировать примеры программ из этой главы, читателю понадобится компилятор с поддержкой стандарта C++ 11 или более позднего. Сначала возьмём в качестве образца классическую, простейшую программу «Здравствуй, мир», прежде чем переходить к многопоточной реализации.

```
// Спасибо Деннису Ритчи и Брайану Кернигану за этот обычай
#include <iostream>
int main()
```

¹ Цит. по русскому переводу статьи: <https://habr.com/post/145432/>.



```
{
    std::cout << "Hello world\n";
    return 0;
}
```

Эта программа просто выводит текст «Hello World» (англ. «Здравствуй, мир») на стандартное устройство вывода (как правило, на консоль). Теперь рассмотрим программу, которая делает то же самое, но использует для этого фоновый поток, также часто называемый рабочим потоком.

```
#include <iostream>
#include <string>
#include <thread>

// Будет выполняться в отдельном потоке
void thread_proc(std::string msg)
{
    std::cout << "thread_proc says: " << msg;
}

int main()
{
    // Создать поток и выполнить в нём функцию 'thread_proc'
    std::thread t(thread_proc, "Hello World\n");

    // Дождаться завершения потока и завершить программу
    t.join();
}
```

Первое отличие этого примера от простейшей реализации – это подключение стандартного заголовочного файла `<thread>`. В нём объявлены все функции и классы для работы с потоками. Однако объявления классов, обеспечивающих синхронизацию и атомарный доступ к данным, расположены в других файлах. Читатели, знакомые со средствами управления потоками системы Windows и стандарта POSIX, знают, что всякому потоку нужна своя начальная функция. Данному принципу следует также и стандартная библиотека. В этом примере функция `thread_proc` становится начальной функцией потока, создаваемого в функции `main`. Начальная функция (в данном случае заданная посредством указателя) передаётся в конструктор объекта `t` типа `std::thread`, и конструктор начинает выполнение нового потока.

Наиболее заметное отличие этой реализации от предыдущей состоит в том, что новое приложение выводит сообщение на консоль из нового (фоновое, рабочего) потока, и, таким образом, приложение в целом выполняется в два потока. Как только рабочий поток запущен на выполнение, главный поток продолжает свою работу. Если бы главный поток не ждал завершения второго потока, функция `main` завершилась бы сразу, что привело бы к завершению всего приложения – независимо от того, завершились ли остальные потоки. Для этого и необходимо вызывать функцию `join`, перед тем как завершится главный поток, – он должен дождаться окончания вспомогательного потока `t`.

УПРАВЛЕНИЕ ПОТОКАМИ

Выполнение приложения начинается (после выполнения некоторых системных действий) с функции `main`, играющей роль точки входа в программу, и для этого создаётся поток по умолчанию. Таким образом, каждая программа обладает хотя бы одним потоком выполнения. Во время своего выполнения программа может создавать любое число дополнительных потоков, пользуясь для этого средствами стандартной библиотеки или специфическими средствами конкретной платформы. Эти потоки могут выполняться параллельно, если в наличии имеется достаточное число процессорных ядер. Если же потоков оказывается больше, чем ядер процессора, то уже невозможно выполнять все потоки полностью одновременно. Следовательно, в этом случае тоже будет иметь место переключение между потоками. Сколько бы потоков ни создавалось из главного потока, все они выполняются параллельно с главным потоком или делят с ним ресурс процессора. Кроме того, когда вместе с функцией `main` завершается начальный поток, завершается вся программа: это приводит к завершению всех остальных её потоков. Поэтому, если дочерние потоки продолжают выполнять важную работу, главному потоку необходимо дождаться их нормального завершения, прежде чем завершаться самому. Рассмотрим подробнее, как происходят запуск потоков и ожидание их окончания.

Запуск потока

В предыдущем примере мы видели, что указатель на функцию передаётся в качестве аргумента в конструктор объекта `std::thread`, и поток запускается. Переданная функция начинает выполняться в отдельном потоке. Таким образом, запуск нового потока осуществляется конструктором объекта `std::thread`. Кроме передачи указателя на функцию, есть и другие способы проинициализировать объект-поток. Функциональный объект также можно использовать в качестве аргумента при создании потока. Вообще говоря, стандартная библиотека языка C++ гарантирует, что объекты класса `std::thread` могут работать с любыми сущностями, которые можно вызывать как функции (т. е. с сущностями, обладающими операцией функционального применения). Тем самым стандарт поддерживает инициализацию потоков такими аргументами:

- указатели на функции (как показано в примере);
- функциональные объекты, т. е. объекты с перегруженной операцией вызова;
- лямбда-функции.

Всё, что может быть вызвано, может быть кандидатом для создания потока. Покажем пример того, как создавать поток из объекта с перегруженной операцией вызова.

```
class parallel_job
{
public:
```

```
void operator() () {
    some_implementation(); // какая-то реализация
}
};
```

```
parallel_job job;
std::thread t(job);
```

В процессе создания потока объект `job` копируется в его адресное пространство – следовательно, функциональный объект должен поддерживать копирование. Если копирование невозможно или нежелательно, его можно избежать, воспользовавшись функцией `std::move`, как показано ниже:

```
std::thread t(std::move(job));
```

Конечно, перемещение вместо копирования произойдёт и в том случае, если передать временный объект (т. е. выражение категории `rvalue`), что может быть оформлено в коде следующим образом:

```
std::thread t(parallel_job());
```

Эту строчку, впрочем, можно ошибочно принять за объявление функции, которая принимает указатель на функцию и возвращает объект типа `std::thread`. Можно избежать путаницы, если воспользоваться единым синтаксисом инициализации, например:

```
std::thread t{ parallel_job{} };
```

Избежать этой же путаницы можно и иным способом: с помощью дополнительной пары скобок, как показано в следующем примере:

```
std::thread t((parallel_job()));
```

Ещё один интересный способ создавать потоки состоит в том, чтобы в конструктор объекта `std::thread` передавать лямбда-функцию. Лямбда-функции могут захватывать локальные переменные, тем самым устраняя потребность в передаче аргументов. Хотя лямбда-функции могут оказаться весьма удобными, когда нужно создать анонимную функцию, связанную с текущим контекстом, это не значит, что их стоит использовать всюду. Ниже показан пример создания потока на основе лямбда-функции:

```
std::thread t( [] () { some_implementation(); } );
```

Присоединение к потоку

Читатель мог заметить, что в примере многопоточной программы «Здравствуй, мир» в самом конце функции `main`, т. е. перед завершением программы, стоит вызов `t.join()`. Вызов метода `join` для объекта, инкапсулирующего рабочий поток, гарантирует, что поток, вызвавший этот метод, дожждётся завершения потока, завернутого в объект. Без такого вызова может оказаться, что раньше, чем вспомогательный поток начнёт свою работу, главный поток

завершится, что вызовет немедленное завершение всех созданных им вспомогательных потоков.

Метод `join` либо ждёт, пока поток не завершится, либо не делает ничего, если поток уже завершился. Для более тонкого управления потоками предназначены иные механизмы, такие как двоичные семафоры (англ. `mutex`), условные переменные (`condition variable`), фьючерсы (`future`), о которых будет рассказано в следующих разделах этой главы и в следующей главе. Метод `join` освобождает память, используемую объектом-потоком, и тем самым гарантирует, что объект более не связан ни с каким выполняющимся потоком. Это означает, что метод `join` можно вызывать лишь один раз для каждого объекта-потока. После вызова метода `join` метод `joinable` того же объекта вернёт значение `false`. Чтобы лучше понять метод `join`, предыдущий пример с функциональным объектом можно изменить следующим образом:

```
class parallel_job
{
    int& _iterations;

public:
    parallel_job(int& iterations):
        _iterations(iterations)
    {}

    void operator() () {
        for (int i = 0; i < _iterations; ++i)
        {
            some_implementation(i); // какая-то реализация
        }
    }
};

void func()
{
    int local_var = 10000;
    parallel_job job(local_var);
    std::thread t(job);
    if (t.joinable())
        t.join();
}
```



В этом примере функция `func` перед своим завершением проверяет, выполняется ли ещё созданный в ней поток `t`. С помощью метода `joinable` функция проверяет, можно ли подключиться к этому потоку (разумеется, в данном примере он всегда вернёт значение `true`), и если проверка дала положительный результат, подключается и ожидает его завершения.

Теперь попробуем предотвратить ожидание функцией `func` своего дочернего потока. Существует стандартный механизм, позволяющий потоку продолжать свою работу даже после завершения функции, в которой он был запущен, а именно метод `detach`.

```
if (t.joinable())
    t.detach();
```

Однако есть ряд тонкостей, о которых нужно задуматься прежде, чем отсоединяться от потока методом `detach`. В данном примере поток `t`, скорее всего, всё ещё будет выполняться в момент завершения функции `func`. Как видно из показанного выше исходного кода, поток использует ссылку на локальную переменную функции `func`, что очевидно приведёт к серьёзной ошибке, так как на стеке по этому же адресу наверняка будут размещены другие данные (в большинстве вычислительных архитектур). Такую опасность всегда нужно иметь в виду, прежде чем применять в своей программе метод `detach`. Обычный способ защититься от неё – сделать поток самодостаточным, а необходимые для работы потока данные копировать в него, а не передавать в совместное пользование через указатели.



Передача аргументов в поток

Выше мы разобрались, как запускать поток и ожидать его завершения. Теперь разберём, как при инициализации потока передавать аргументы для функции, которая будет в потоке выполняться. В качестве примера рассмотрим вычисление факториала числа.

```
class Factorial
{
private:
    long double _fact;

public:
    Factorial() : _fact (1)
    {}

    void operator() (int number)
    {
        _fact = 1;
        for (int i = 1; i <= number; ++i)
        {
            _fact *= i;
        }

        std::cout
            << "Факториал числа "
            << number
            << " равен "
            << _fact
            << std::endl;
    }
};

int main()
{
```





```
Factorial fact;
std::thread t1(fact, 10);
t1.join();
}
```

Как видно из этого примера, чтобы в отдельном потоке запустить функцию (или функциональный объект) с определёнными аргументами, нужно передать эти аргументы в конструктор объекта `std::thread`. При этом нужно понимать, что значения этих аргументов копируются во внутреннее хранилище потока. Как мы уже убедились выше, потоку для нормального выполнения необходимы собственные копии всех аргументов – в противном случае могут возникать ошибки из-за окончания времени жизни локальных переменных. Чтобы лучше изучить передачу аргументов в поток, обратимся снова к нашему первому примеру «Здравствуй, мир»:

```
void thread_proc(std::string msg);
std::thread t(thread_proc, "Hello World\n");
```

Здесь функция `thread_proc` принимает один аргумент типа `std::string`, однако при создании потока передаётся значение типа `const char*`. Это значение поступает на самом деле в конструктор объекта `std::thread`, там оно преобразовывается в объект типа `std::string` и копируется во внутреннее пространство потока. Таким образом, клиентский код передаёт значение типа `const char*`, а в поток попадает объект типа `std::string`. Этот механизм неявного преобразования нужно иметь в виду, выбирая тип аргумента для функции потока. Посмотрим, что получится, если передать в конструктор потока указатель на локальный массив символов:

```
void thread_proc(std::string msg);
void func() {
    char buf[512] = "Hello World\n";
    std::thread t(thread_proc, buf);
    t.detach();
}
```

Аргумент, переданный в конструктор потока, – это указатель на массив `buf`, объявленный локально в функции `func`. Есть теоретическая возможность, что функция `func` завершится до того, как массив символов `buf` будет преобразован в объект типа `std::string`. Это может привести к неопределённому поведению. Данную проблему можно решить, если в явном виде привести массив `buf` к типу `std::string` перед передачей в конструктор объекта `std::thread`:

```
std::thread t(thread_proc, std::string(buf));
```

Рассмотрим теперь случай, когда потоку действительно необходимо изменять объект по ссылке, полученной через аргумент. В обычном случае конструктор потока создаёт копию значения, переданного ему в качестве аргумента, именно для того, чтобы сделать невозможной передачу в поток ссылки на локальную переменную, однако стандартная библиотека содержит и сред-

ство для передачи аргумента по ссылке. Во многих реальных системах можно видеть примеры того, как с одной и той же структурой данных работает несколько потоков. Ниже показан пример того, как передать в поток ссылку.

```
void update_data(shared_data& data);

void another_func() {
    shared_data data;
    std::thread t(update_data, std::ref(data));
    t.join();
    do_something_else(data);
}
```

В этом примере аргумент, передаваемый в конструктор объекта `std::thread`, обернут функцией `std::ref`, благодаря чему в функцию потока гарантированно попадёт ссылка на фактически переданный аргумент. Читатель наверняка обратил внимание на то, что функция `update_data` в соответствии с прототипом принимает ссылку на объект типа `shared_data`, тогда зачем же нужна обёртка `std::ref` над аргументом, который и без того является ссылкой? Чтобы понять это, рассмотрим следующий код:

```
std::thread t(update_data, data);
```

Конечно же, функция `update_data` трактует переданный ей аргумент типа `shared_data` как ссылку. Однако объект `data` является аргументом не этой функции, а конструктора объекта-потока, который просто копирует этот объект во внутренние структуры данных. Поэтому, когда выполнение потока дойдёт до вызова функции `update_data`, ей будет передана ссылка на локальную копию аргумента, а не ссылка на исходный объект-аргумент. Обёртывание аргумента в функцию `std::ref`, напротив, обеспечивает передачу именно ссылки на сам объект `data`.

ИСПОЛЬЗОВАНИЕ ЛЯМБДА-ФУНКЦИЙ

Убедимся же теперь в том, насколько в многопоточном программировании полезны лямбда-функции. Следующий фрагмент кода создаёт пять потоков и помещает их в контейнер. Каждый поток создаётся на основе лямбда-функции. Каждый поток получает свой номер (текущее значение счётчика цикла), захваченный по значению.

```
int main()
{
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i)
    {
        threads.push_back(std::thread(
            [i]() { std::cout << "Поток " << i << std::endl; }));
    }
}
```



```

std::cout << "\nГлавная функция";

std::for_each(
    threads.begin(),
    threads.end(),
    [](std::thread &t) { t.join(); });

return 0;
}

```



Контейнер `threads` содержит в себе пять потоков, запущенных в теле цикла. Главный поток присоединяет эти потоки снова к себе (т. е. ожидает их завершения) в конце функции `main`. Результат выполнения этого кода может выглядеть, например, так:

```

Поток 0
Поток 4
Поток 1
Поток 3
Поток 2
Главная функция

```

Текст, печатаемый программой, наверняка будет отличаться при каждом запуске. Эта программа может служить хорошим примером недетерминированности, внутренне присущей многопоточному программированию. В следующем разделе мы займёмся тем, как объекты типа `std::thread` ведут себя при перемещении.

Управление владением

В примерах, разобранных ранее в этой главе, читатель мог заметить, что функция, запустившая новый поток, должна либо дождаться его завершения с помощью метода `join`, либо, вызвав метод `detach`, утратить связь с этим потоком и отпустить его. В современном стандарте языка C++ есть множество типов данных, объекты которых можно перемещать, но нельзя копировать, и тип `std::thread` — один из них. Это означает, что никакие два объекта не могут владеть одним и тем же потоком, но исключительное владение потоком может передаваться от одного объекта `std::thread` к другому посредством семантики перемещения.

Есть много ситуаций, в которых может понадобиться передача владения потоком от объекта к объекту, например если функция, создавшая поток, хочет завершиться, не дожидаясь его окончания. Тогда она может передать владение потоком той функции, из которой вызвана сама, вернув ей в качестве значения объект типа `std::thread`. Другим примером может быть передача потока в функцию в качестве аргумента, чтобы она дождалась его завершения. В обоих случаях цель может быть достигнута путём передачи владения потоком от одного объекта-обёртки к другому.

Чтобы пояснить сказанное, определим две функции, которые позднее будем запускать в различных потоках:


```

void function1()
{
    std::cout << "Функция 1\n";
}

void function2()
{
    std::cout << " Функция 2\n";
}

```

Теперь рассмотрим главную функцию, которая запускает эти функции в отдельных потоках.

```

int main()
{
    std::thread t1(function1);
    // Передача владения потоком от объекта t1 к t2
    std::thread t2 = std::move(t1);

```



Сначала создаётся новый объект-поток t1. Затем владение потоком переходит от него к объекту t2, для чего используется функция `std::move`, вследствие чего создание объекта t2 выполняется перемещающим конструктором. С этого момента объект t1 более не связан ни с каким выполняющимся потоком. Поток, в котором выполняется функция `function1`, теперь связан с объектом t2. Пусть далее следует строка

```
t1 = std::thread(function2);
```

Здесь запускается новый поток вместе с созданием объекта-обёртки. Выражение в правой части присваивания является rvalue, поэтому выполняется перемещающая операция присваивания, которая передаёт владение потоком из временного объекта в правой части объекту t1. Обратим внимание, что поскольку в правой части присваивания стоит выражение категории rvalue, нет необходимости оборачивать это значение функцией `std::move`. Рассмотрим теперь следующие строки:

```

// пустой объект, не владеющий никаким потоком
std::thread t3;
// Владение потоком передаётся от t2 объекту t3
t3 = std::move(t2);

```

Объект t3 создаётся конструктором по умолчанию, без запуска какого-либо потока. Затем поток, которым владеет объект-обёртка t2, передаётся объекту t3 в исключительное владение посредством перемещающей операции присваивания, на что указывает использование функции `std::move`.

```

// ожидать объект t2 не нужно: он уже не связан с потоком
if (t1.joinable()) t1.join();
if (t3.joinable()) t3.join();
return 0;
}

```

Наконец, перед завершением программы необходимо дождаться завершения всех созданных в ней потоков. В данном случае только объекты `t1` и `t3` связаны с выполняющимися потоками.

Допустим теперь, что перед обращениями к методу `join` имеется такая строка:

```
t1 = std::move(t3);
```

Объект `t1` в этой точке программы уже связан с потоком, в котором выполняется функция `function2`. Когда оператор присваивания пытается передать в этот, уже занятый объект владение потоком из объекта `t3` (а именно потоком, в котором выполняется функция `function1`), программа аварийно завершается (с помощью функции `std::terminate`). Этим гарантируется корректное поведение деструктора объектов `std::thread`.

Поддержка семантики перемещения классом `std::thread` позволяет передавать владение потоком наружу из функции. Следующий фрагмент кода демонстрирует это:

```
void func()
{
    std::cout << "func()\n";
}

std::thread thread_creator()
{
    return std::thread(func);
}

void thread_wait_func()
{
    std::thread t = thread_creator();
    t.join();
}
```

Функция `thread_creator` возвращает объект типа `std::thread`, связанный с потоком, в котором выполняется функция `func`. Функция `thread_wait_func` вызывает функцию `thread_creator`, получает от неё объект-поток¹. Вызов функции является выражением категории `rvalue`, он используется для инициализации нового объекта типа `std::thread`. Тем самым владение потоком передаётся локальному объекту `t`, затем функция использует его, чтобы дождаться завершения потока.

СОВМЕСТНЫЙ ДОСТУП ПОТОКОВ К ДАННЫМ

Выше мы изучили создание потоков и несколько способов управления ими. Теперь разберём, как обеспечить совместный доступ нескольких потоков к об-

¹ Строго говоря, последовательность действий несколько отличается от этой упрощённой модели благодаря оптимизации возврата значения из функции, известной как «исключение копирования» (англ. *copy elision*).

щим данным. Возможность доступа к одним и тем же данным из различных потоков имеет исключительное значение для параллельного программирования. Сначала разберём, какие проблемы могут возникнуть при совместном доступе потоков к общим данным.

Проблем никаких возникнуть не может, если общие данные, к которым обращаются потоки, неизменяемы (т.е. открыты только для чтения), поскольку чтение данных тем или иным потоком никак на них не влияет и, следовательно, остальные потоки прочтут те же самые данные. Трудности возникают там, где потоки получают возможность модифицировать общие данные.

Когда потоки, имеющие доступ к сложной структуре данных, одновременно пытаются вносить в неё изменения, это легко может привести к нарушению инвариантов, характеризующих целостность этой структуры. Пусть, например, в контейнерном объекте хранятся некоторые элементы и счётчик этих элементов, и пусть над контейнером выполняется не совсем элементарная модифицирующая операция. Это может быть, скажем, удаление элемента из самобалансирующегося дерева или двухсвязного списка. Если один поток без специальных мер безопасности читает данные из контейнера, в то время как другой поток занимается удалением элемента, вполне может случиться, что читающий поток увидит структуру данных с частично удалённым элементом¹ и, следовательно, нарушенным инвариантом. Это может привести к непоправимому повреждению структуры данных и к краху программы.

i Инвариантом называют совокупность соотношений, истинность которых должна сохраняться на протяжении всего времени жизни объекта. Вписывание контрольных утверждений в программный код с целью автоматической проверки инвариантов помогает сделать код надёжным. Кроме того, это ещё и прекрасный способ документирования кода, а также хорошая защита от дефектов, возникающих при модификации кода. Более подробную информацию можно найти в Википедии по ссылке [https://en.wikipedia.org/wiki/Invariant_\(computer_science\)](https://en.wikipedia.org/wiki/Invariant_(computer_science)).

Попытки вносить изменения в общие данные из разных потоков нередко приводят программу в *состояние гонки* (англ. race condition) – по-видимому, наиболее частую причину ошибок в параллельных программах. Данный термин означает, что потоки наперегонки пытаются выполнить свои действия над данными. Таким образом, общий результат их деятельности зависит от порядка выполнения операций в разных потоках, от их относительной скорости. Чаще всего под гонками потоков понимают именно такие, потенциально приводящие к ошибкам гонки. Конечно, если каждый из потоков работает исключительно над своими данными, никакие гонки между ними не могут привести к ошибкам. Опасность гонок обычно связана с тем, что изменение общей

¹ Более конкретно, элемент уже удалён из списка, а счётчик элементов ещё не уменьшен на единицу; или, при удалении второго из трёх элементов двухсвязного списка, указатель на следующий элемент для первого элемента уже переставлен на третий, тогда как третий элемент всё ещё считает предыдущим второй и т. д. – *Прим. перев.*

структуры данных требует изменения двух или более составляющих её элементов данных (как удаление элемента из сбалансированного дерева или из двухсвязного списка). Поскольку модификация затрагивает разные значения и выполняется разными машинными командами, другой поток вполне может начать обращение к структуре данных между этими командами, т. е. когда модификация структуры данных выполнена лишь наполовину.

Состояние гонок часто бывает крайне трудно обнаружить и ещё труднее воспроизвести, поскольку доступный наблюдению эффект очень сильно зависит от случайных факторов, влияющих на очерёдность выполнения потоков. Едва ли не главная сложность при разработке параллельных программ как раз и состоит в предотвращении всевозможных гонок между потоками.

Есть несколько способов борьбы с нежелательными гонками потоков. Наиболее простой и распространённый из них состоит в использовании примитивов синхронизации, основанных на взаимной блокировке потоков. Общая структура данных помещается в некоторую оболочку, механизмы которой предотвращают одновременный доступ нескольких потоков. Далее в этой главе будет подробно рассказано о различных примитивах синхронизации и способах их использования.

Другое решение состоит в том, чтобы так переработать структуры данных и их инварианты, чтобы целостность данных и корректность операций над ними гарантировались даже при одновременном доступе из нескольких потоков. Такой подход к написанию программ, известный как *неблокирующее программирование*, довольно сложен. Неплокирующее программирование на языке C++ и лежащая в его основе модель памяти обсуждаются в главе 4 «Асинхронное программирование и неблокирующая синхронизация в языке C++».

Наконец, изменения структур данных можно обрабатывать в режиме транзакций, подобно тому, как обрабатываются модификации баз данных. Данная тема выходит за рамки настоящей книги и далее обсуждаться не будет.

Теперь приступим к изучению самого важного из стандартных средств синхронизации доступа к общим данным, а именно двоичного семафора, или мьютекса.

Двоичные семафоры

Двоичный семафор, также называемый мьютексом (англ. mutex), – это механизм, используемый в параллельном программировании для борьбы с гонками потоков. Основное назначение двоичного семафора состоит в том, чтобы не дать потоку войти в *критическую секцию*, пока другой поток находится в своей критической секции. Семафор может находиться в открытом и запертом состоянии, с его помощью поток может просигнализировать, что вошёл в критическую секцию, требующую исключительного доступа. Когда поток входит под семафор, последний запирается, и другие потоки, стремящиеся войти под него, вынуждены ждать, пока первый поток не освободит семафор. В стандарте

C++ 11 стандартная библиотека пополнилась классом `std::mutex`, который реализует эту функциональность.

Класс `std::mutex` содержит методы `lock` и `unlock` для входа в критическую секцию и выхода из неё. Работая с критическими секциями посредством этих методов, нужно тщательно следить за тем, чтобы всякому захвату семафора методом `lock` соответствовало его освобождение методом `unlock`, в противном случае остальные потоки будут до бесконечности ожидать своей очереди на вход в критическую секцию.

Вернёмся к примеру кода, которым выше было проиллюстрировано создание потоков на основе лямбда-функции. Анализируя этот пример, мы отметили, что текст, выводимый на консоль параллельно работающими потоками, перемешивается – т. е. имеет место состояние гонок между потоками за доступ к потоку `std::cout` как к единому глобальному ресурсу. Перепишем этот код, используя объект `std::mutex` для исключительного доступа к устройству вывода.

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

std::mutex m;

int main()
{
    std::vector<std::thread> threads;

    for (int i = 0; i < 5; ++i)
    {
        threads.push_back(std::thread( [i]() {
            m.lock();
            std::cout << "Поток " << i << std::endl;
            m.unlock();
        }));
    }

    std::for_each(
        threads.begin(),
        threads.end(),
        [](std::thread &t) { t.join(); });

    return 0;
}
```



Результат работы этой программы может выглядеть так:

```
Поток 2
Поток 4
Поток 0
Поток 1
Поток 3
```

В этой программе двоичный семафор используется для защиты общего ресурса (каковым является устройство вывода `std::cout`) от одновременного доступа из нескольких потоков (а именно от одновременного выполнения ими операций вывода). В отличие от первой версии данного примера, наличие критической секции надёжно предохраняет от смешивания сообщений, выводимых разными потоками (хотя порядок сообщений остаётся непредсказуемым). Методы `lock` и `unlock` вызываются в каждом потоке поочерёдно, чем и обеспечивается согласованная работа всей системы.

Следует, однако, отметить, что прямое использование методов `lock` и `unlock` класса `std::mutex` – это плохая практика, так как в подобном случае программист вынужден самостоятельно заботиться о том, чтобы освобождать семафор на всех возможных путях выполнения, ведущих к выходу из критической секции, включая и аварийный выход по исключению. Чтобы справиться с этой трудностью, в стандартной библиотеке языка C++ предусмотрен специальный шаблонный класс `std::lock_guard`, воплощающий для двоичных семафоров знаменитую идиому RAII (англ. Resource Acquisition Is Initialization – захват ресурса есть инициализация). Конструктор объекта данного класса захватывает и семафор, а деструктор освобождает его. Объявление этого класса содержится в стандартном заголовочном файле `<mutex>`. Воспользовавшись классом-обёрткой `std::lock_guard`, предыдущий пример можно переписать следующим образом:

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

std::mutex m;

int main()
{
    std::vector<std::thread> threads;

    for (int i = 0; i < 5; ++i)
    {
        threads.push_back(std::thread( [i]() {
            std::lock_guard<std::mutex> guard(m);
            std::cout << "Поток " << i << std::endl;
        }));
    }

    std::for_each(
        threads.begin(),
        threads.end(),
        [](std::thread &t) { t.join(); });

    return 0;
}
```

Двоичный семафор, общий для критических секций всех потоков, представлен в этом примере глобальной переменной, тогда как обёртывающий его

объект типа `std::lock_guard` локален и создаётся каждый раз, когда очередной поток входит в критическую секцию. Как только объект создан и выполнение потока пошло дальше, можно быть уверенными, что двоичный семафор захвачен. Как только выполнение потока выходит за область видимости объекта `guard`, автоматически вызывается его деструктор, что приводит к освобождению семафора.



Чрезвычайно важная для программирования на языке C++ идиома RAII означает, что время жизни того или иного ресурса (такого как дескриптор файла, соединение с базой данных, сетевое соединение, динамически выделенная область памяти, владение семафором и т. д.) привязано к времени жизни объекта-обёртки. Читатель может найти информацию об идиоме RAII в Википедии по ссылке https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization.

Предотвращение тупиков

Самая большая опасность, подстерегающая программиста при работе с двоичными семафорами, – это тупик. Легко представить себе пример тупика в реальной жизни. Чтобы послушать музыку, человеку нужны одновременно два ресурса: плеер и наушники. Если на двоих братьев имеется один такой комплект, вполне возможна ситуация, когда они захотят послушать музыку совершенно одновременно, при этом один из них положит руку на плеер, а другой в это же время схватит наушники. Теперь первый будет ожидать, пока освободятся наушники, а второй – ждать плеер. Каждый в конечном счёте ждёт себя самого, так как именно его ждёт тот, кого ждёт он сам. Это ожидание способно длиться до бесконечности, если только один из них не уступит из любви к ближнему.

То же самое происходит в программе, если братьев заменить потоками, а плеер и наушники – двумя семафорами. Пусть каждый из двух потоков благополучно захватывает по одному семафору и затем пытается получить другой. Первый поток не может продолжить работу, так как второй семафор занят вторым потоком. Поэтому первый поток никогда не отпустит первый семафор, но именно его ждёт второй поток и из-за этого не может отпустить второй семафор. Эта ситуация и называется тупиком, или мёртвой блокировкой (англ. *deadlock*).

Избежать тупика бывает довольно просто, если различные семафоры служат для защиты различных ресурсов и каждый поток в каждый момент времени нуждается лишь в одном из них. Однако на практике встречаются и более сложные ситуации. Лучшее, что можно здесь посоветовать, – это во всех потоках захватывать семафоры в одном и том же порядке, это делает тупик невозможным¹.

¹ В примере с братьями, желающими послушать музыку, это выглядело бы так: каждый из них в первую очередь захватывает только плеер; затем, только если это удалось, захватывает наушники. Тогда очевидно, что ровно одному из них повезёт, а другому останется ждать, пока музыкой насладится первый, что, конечно же, лучше, чем бесконечно долгое ожидание в тупике.

Рассмотрим в качестве примера программу, состоящую из двух потоков. Пусть один поток выводит на печать чётные, а другой – нечётные числа. Для синхронизации этих потоков будем использовать два двоичных семафора. Общим ресурсом, который нужно защитить от одновременного доступа, является устройство вывода `std::cout`. Код этой программы, потенциально заводящей в тупик, приведён ниже.

```
// Глобальные семафоры
std::mutex evenMutex;
std::mutex oddMutex;

// Функция для печати чётных чисел
void printEven(int max)
{
    for (unsigned i = 0; i <= max; i +=2)
    {
        oddMutex.lock();
        std::cout << i << ", ";
        evenMutex.lock();
        oddMutex.unlock();
        evenMutex.unlock();
    }
}
```

Функция `printEven`, как видно из её определения, печатает на консоль чётные числа от 0 до заданной верхней границы `max`. Определим подобную ей функцию `printOdd` для печати нечётных чисел из того же диапазона, как показано ниже.

```
// Функция для печати нечётных чисел
void printOdd(int max)
{
    for (unsigned i = 1; i <= max; i +=2)
    {
        evenMutex.lock();
        std::cout << i << ", ";
        oddMutex.lock();
        evenMutex.unlock();
        oddMutex.unlock();
    }
}
```

Теперь остаётся определить главную функцию, которая запускает два отдельных потока для печати чётных и нечётных чисел, соответственно, на основе показанных выше функций.

```
int main()
{
    auto max = 100;

    std::thread t1(printEven, max);
    std::thread t2(printOdd, max);
```




```

    if (t1.joinable())
        t1.join();
    if (t2.joinable())
        t2.join();

    return 0;
}

```

Как видно из кода, доступ к глобальному ресурсу `std::cout` защищён двумя семафорами: `oddMutex` и `evenMutex`, причём функции `printOdd` и `printEven` захватывают их в разном порядке. Этот код наверняка попадёт в тупик, так как каждый поток пытается ждать освобождения семафора, захваченного другим потоком, и при этом держит занятым семафор, который нужен другому потоку, чтобы отпустить первый семафор. Попытка выполнить эту программу естественно приведёт к зависанию (кроме чрезвычайно маловероятного случая, если итерации циклов в первом и втором потоках будут выполняться строго поочерёдно). Как уже говорилось выше, тупика можно избежать, если во всех потоках соблюдать одинаковый порядок захвата семафоров, как показано ниже.

```

void printEven(int max)
{
    for (unsigned i = 0; i <= max; i +=2)
    {
        evenMutex.lock();
        std::cout << i << ", ";
        oddMutex.lock();
        evenMutex.unlock();
        oddMutex.unlock();
    }
}

void printOdd(int max)
{
    for (unsigned i = 1; i <= max; i +=2)
    {
        evenMutex.lock();
        std::cout << i << ", ";
        oddMutex.lock();
        evenMutex.unlock();
        oddMutex.unlock();
    }
}

```

Однако такой код слишком тяжёл для понимания, ведь программисту нужно обеспечить одинаковый порядок операций во всех функциях, работающих с этими семафорами. Выше было рассказано, как идиома RAII помогает сделать код проще и безопаснее, если речь идёт о захвате одного семафора. Для безопасного захвата нескольких семафоров в стандартной библиотеке имеется ещё одно средство – функция `std::lock`, которая захватывает два или более семафоров одновременно, атомарным действием, гарантированно предотвращая ту-

пик. Если все семафоры, переданные ей в качестве аргументов, свободны, они будут захвачены; в противном случае эта функция не захватывает ни одного до тех пор, пока не сможет захватить все. Ниже показано, как использовать её в нашем примере с печатью чётных и нечётных чисел.

```
void printEven(int max)
{
    for (unsigned i = 0; i <= max; i +=2)
    {
        std::lock(evenMutex, oddMutex);
        std::lock_guard<std::mutex> lk_even(
            evenMutex, std::adopt_lock);
        std::lock_guard<std::mutex> lk_odd(
            oddMutex, std::adopt_lock);
        std::cout << i << ", ";
    }
}

void printOdd(int max)
{
    for (unsigned i = 1; i <= max; i +=2)
    {
        std::lock(evenMutex, oddMutex);
        std::lock_guard<std::mutex> lk_even(
            evenMutex, std::adopt_lock);
        std::lock_guard<std::mutex> lk_odd(
            oddMutex, std::adopt_lock);
        std::cout << i << ", ";
    }
}
```

При входе в очередную итерацию цикла функция `std::lock` захватывает оба семафора. Затем создаются два объекта-обёртки типа `std::lock_guard`, по одному для каждого семафора. Каждому из них в конструктор, помимо семафора, передаётся аргумент `std::adopt_lock`, который означает, что семафор уже захвачен и, следовательно, конструктору нужно лишь принять владение семафором, а не пытаться захватывать его. Тогда деструкторы объектов-обёрток обеспечат освобождение семафоров, даже если внутри критической секции произойдёт исключение.

В заключение подчеркнём, что функция `std::lock` помогает избежать тупика, если все потоки используют её для одновременного захвата нескольких семафоров. Конечно же, эта функция не поможет, если некоторые потоки продолжают захватывать семафоры по отдельности в произвольном порядке.

Тупики – это одна из самых трудных для обнаружения и исправления ошибок, какие только могут возникнуть в многопоточных программах. Чтобы избежать тупика, от программиста требуется большая аккуратность и самодисциплина.

Условные переменные

Выше мы разобрали, как с помощью двоичных семафоров синхронизировать обращения нескольких потоков к общему ресурсу. Однако синхронизация на основе семафоров может оказаться чересчур сложной и без должной осторожности может завести систему в тупик. В этом разделе рассмотрим более простой механизм на основе ожидания событий с использованием так называемых условных переменных.

В случае синхронизации на основе семафоров поток, выполняющий критическую секцию, может быть заблокирован любым другим потоком. Кроме того, если ожидание завершения какого-либо потока реализовать путём периодического опроса флага состояния, защищённого семафором, это приведёт к неоправданному расходу процессорного времени. Время, которое можно было бы эффективно использовать для других потоков, в этом случае тратилось бы на ожидание семафора.

Для решения этой проблемы в стандартной библиотеке языка C++ предусмотрены два вида условных переменных: `std::condition_variable` и `std::condition_variable_any`. Оба класса объявлены в заголовочном файле `<condition_variable>`, обоим для работы нужен некоторый семафор. Различие между ними состоит в том, что объект типа `std::condition_variable` может работать исключительно со стандартным двоичным семафором `std::mutex`, тогда как объект типа `std::condition_variable_any` допускает любую сущность, ведущую себя подобно семафору (т. е. обладающую семантикой семафора), отсюда и суффикс «any» (любой). Большая общность и гибкость достаются не бесплатно: объекты типа `std::condition_variable_any` потребляют больше памяти и обладают меньшим быстродействием. Поэтому их стоит использовать только тогда, когда нестандартные семафоры действительно необходимы по какой-то причине.

Ниже показано, как знакомую из предыдущего раздела программу с двумя потоками для генерации чётных и нечётных чисел реализовать по-новому, используя условные переменные.

```
std::mutex numMutex;
std::condition_variable syncCond;
auto bEvenReady = false;
auto bOddReady = false;

void printEven(int max)
{
    for (unsigned i = 0; i <= max; i +=2)
    {
        std::unique_lock<std::mutex> lk(numMutex);
        syncCond.wait(lk, []{return bEvenReady;});
        std::cout << i << ", ";
        bEvenReady = false;
        bOddReady = true;
        syncCond.notify_one();
    }
}
```

Текст программы начинается с глобального объявления двоичного семафора, условной переменной и двух флагов, чтобы их можно было использовать в обоих потоках. Функция `printEven`, выполняющаяся в отдельном потоке, печатает чётные числа, начиная с 0. При входе в тело цикла для захвата семафора используется объект-обёртка типа `std::unique_lock` вместо изученного ранее типа `std::lock_guard`, причина чего станет вскоре понятна. Затем функция вызывает метод `wait` (ожидать) для условной переменной (объекта типа `std::condition_variable`), передавая ему в качестве аргументов только что созданную обёртку над семафором и предикат – лямбда-функцию, выражающую условие, наступления которого необходимо ждать. Вместо последней можно использовать любую сущность, которую можно вызвать, и получить возвращаемое значение логического типа. В данном примере предикат просто возвращает значение глобального флага `bEvenReady` («чётный готов») – таким образом, функция продолжит своё выполнение, когда этот флаг получит значение «истина». Если предикат возвращает значение «ложь», функция `wait` освободит семафор и станет ждать сигнала от других потоков. Именно поэтому для управления семафором должен использоваться тип `std::unique_lock`, позволяющий отпирать и снова запирает семафор¹.

Как только очередное значение отправлено на стандартное устройство вывода `std::cout`, флаг `bEvenReady` («чётный готов») сбрасывается, а флаг `bOddReady` («нечётный готов») взводится. Затем для условной переменной вызывается метод `notify_one` («известить одного»), это пробуждает второй поток, отвечающий за нечётные числа, и заставляет его проверить, взведён ли его флаг. Реализация этого второго потока вполне симметрична первому:

```
void printOdd(int max)
{
    for (unsigned i = 1; i <= max; i +=2)
    {
        std::unique_lock<std::mutex> lk(numMutex);
        syncCond.wait(lk, []{return bOddReady;});
        std::cout << i << ", ";
        bEvenReady = true;
        bOddReady = false;
        syncCond.notify_one();
    }
}
```

Функция `printOdd` выполняется во втором рабочем потоке и печатает нечётные числа начиная с 1. Подобно функции `printEven`, каждая итерация цикла сначала ждёт наступления события, используя для этого условную переменную, семафор и флаг, объявленные в глобальной области видимости. В противоположность предыдущей функции эта функция ждёт, пока истинным станет

¹ Объект типа `std::lock_guard` запирает семафор ровно один раз, в конструкторе, и освобождает тоже однократно – в деструкторе. – *Прим. перев.*

флаг готовности нечётного `bOddReady`, затем сбрасывает его и устанавливает флаг готовности чётного `bEvenReady`. Потом вызов метода `notify_one` для условной переменной извещает поток, занимающийся чётными числами, что он может продолжить работу. Таким образом, два потока будут выполнять свои итерации строго поочерёдно.

```
int main()
{
    auto max = 100;
    bEvenReady = true;

    std::thread t1(printEven, max);
    std::thread t2(printOdd, max);

    if (t1.joinable())
        t1.join();
    if (t2.joinable())
        t2.join();

    return 0;
}
```

Главная функция программы просто создаёт два потока: поток `t1`, в котором выполняется функция `printEven`, и поток `t2`, связанный с функцией `printOdd`. Чтобы поочерёдное выполнение двух потоков могло начаться, устанавливается флаг, разрешающий работу потока с чётными числами.

ПОТОКОБЕЗОПАСНЫЙ СТЕК

К настоящему моменту читатель изучил, как запускать потоки и управлять ими, как синхронизировать операции, выполняемые в различных потоках при совместном доступе к общему ресурсу. При разработке реальных программных систем данные, как правило, бывают организованы в более или менее сложные структуры, которые нужно выбирать подходящим образом, чтобы обеспечить необходимую производительность системы. В этом разделе мы разберём, как с использованием двоичных семафоров и условных переменных реализовать стек, пригодный для работы в многопоточной среде. Этот класс будет обёрткой над стандартным классом `std::stack`, объявление которого находится в стандартном заголовочном файле `<stack>`. Наш потокобезопасный стек, в отличие от стандартного, будет поддерживать несколько вариантов операции выталкивания (`pop`) – это не только поможет пользователям класса писать более лаконичный код, но и даст возможность нам более отчётливо показать, как структуру данных, изначально ориентированную на последовательный доступ, погрузить в параллельную среду.

```
template <typename T>
class Stack
{
```



```
private:
    std::stack<T> myData;
    mutable std::mutex myMutex;
    std::condition_variable myCond;

public:
    Stack() = default;
    ~Stack() = default;
    Stack& operator=(const Stack&) = delete;

    Stack(const Stack& that)
    {
        std::lock_guard<std::mutex> lock(that.myMutex);
        myData = that.myData;
    }
```

Объект класса-шаблона `Stack` содержит в себе объект стандартного шаблонного класса `std::stack` (полезную нагрузку), а также двоичный семафор и условную переменную. Конструктор и деструктор нашего класса получают реализацию по умолчанию, генерируемую самим компилятором; копирующая операция присваивания запрещена, чтобы код, пытающийся выполнить такое присваивание, невозможно было даже скомпилировать¹. Конструктор копирования, однако, разрешён: основную работу выполняет копирующий конструктор завёрнутого в наш объект стандартного стека, нам остаётся лишь заблокировать доступ других потоков к объекту-источнику.

```
void push(T new_val)
{
    std::lock_guard<std::mutex> local_lock(myMutex);
    myData.push(new_val);
    myCond.notify_one();
}
```

Метод `push` (вталкивание в стек нового значения) представляет собой довольно простую обёртку над одноимённым методом стандартного стека, завёрнутого в наш объект. При этом состояние нашего объекта защищено от модификации другими потоками: семафор, управляющий исключительным доступом к объекту, захватывается объектом `std::lock_guard`. Далее следует вызов метода `notify_one` для условной переменной – если до вталкивания элемента стек был пуст, и при этом какой-то другой поток заблокирован, ожидая появления данных в стеке, он получит оповещение. Такое ожидание данных реализовано в двух вариантах операции `pop`, о которых речь пойдёт ниже.

```
bool try_pop(T& return_value)
{
    std::lock_guard<std::mutex> local_lock(myMutex);
```

¹ Вряд ли такое решение можно назвать оправданным. Нет препятствий к тому, чтобы разрешить присваивание, реализовав его по образцу копирующего конструктора. – *Прим. перев.*

```

    if (myData.empty()) return false;
    return_value = myData.top();
    myData.pop();
    return true;
}

```

Метод `try_pop` получает в качестве аргумента ссылку на переменную того же типа, что и хранящиеся в стеке значения. Возвращает метод значение логического типа: «истина» или «ложь». Этот метод не ждёт, пока в стеке появятся данные. Если в момент вызова стек пуст, метод просто возвращает значение `false`. В противном случае он изымает из стека верхний элемент, присваивает его по ссылке-аргументу и возвращает значение `true`. Поскольку данный метод сразу завершается в обоих случаях, ему не нужно ждать условную переменную, и для управления семафором вполне достаточно объекта типа `std::lock_guard`. Все разнообразные варианты операции `pop` нашего класса реализованы на основе метода `top` (возвращает верхний элемент стандартного стека), вслед за которым сразу вызывается метод `pop` (который у стандартного стека удаляет верхний элемент, ничего при этом не возвращая).

```

std::shared_ptr<T> try_pop()
{
    std::lock_guard<std::mutex> local_lock(myMutex);
    if (myData.empty()) return std::shared_ptr<T>();
    std::shared_ptr<T> return_value(
        std::make_shared<T>(myData.top()) );
    myData.pop();
    return return_value;
}

```

Здесь мы видим иной вариант метода `try_pop`, который не принимает аргументов и возвращает умный указатель. Как и в предыдущем случае, операция под названием `try_pop` никогда не ждёт появления элементов в стеке, а завершается сразу – отсюда использование объекта `std::lock_guard` для захвата семафора. Этот метод возвращает умный указатель на значение, снятое с верхушки стека (если оно существует) или пустой умный указатель (если стек пуст).

```

void wait_n_pop(T& return_value)
{
    std::unique_lock<std::mutex> local_lock(myMutex);
    myCond.wait(
        local_lock,
        [this]{ return !myData.empty(); });
    return_value = myData.top();
    myData.pop();
}


std::shared_ptr<T> wait_n_pop()
{
    std::unique_lock<std::mutex> local_lock(myMutex);
    myCond.wait(

```

```


        local_lock,
        [this]{ return !myData.empty(); });
std::shared_ptr<T> return_value(
    std::make_shared<T>(myData.top()));
return return_value;
}
};

```



Две версии операции `pop`, разобранные ранее, не ожидали, пока в стеке появятся данные – для пустого стека они немедленно завершались, тем или иным способом сообщая о неудаче. Ожидание реализовано в двух последних методах с использованием условной переменной. В одном из них значение, снятое с верхушки стека, передаётся наружу через аргумент типа ссылки, другой возвращает значение, завернутое в умный указатель¹. В обоих методах для управления семафором используется объект `std::unique_lock`, так как именно его можно передать в качестве аргумента методу `wait` класса `std::condition_variable`. Предикат, передаваемый в метод `wait` вторым аргументом, проверяет, пуст ли стек. Если стек пуст, метод `wait` временно освобождает семафор и ждёт, пока условная переменная не получит оповещение – а получить она его может только из метода `push`. Как только, в результате операции `push`, предикат вернёт значение «истина», метод `wait_n_pop` продолжит свою работу. Первая из двух его версий снимет элемент с верхушки стека и присвоит его по ссылке, вторая – вернёт обёрнутым в умный указатель.

Итоги



В этой главе мы разобрали средства многопоточного программирования, имеющиеся в стандартной библиотеке языка C++. В частности, рассказано о том, как запускать потоки и управлять ими, как передавать аргументы для запуска функции в отдельном потоке, о передаче владения потока от объекта к объекту, о совместном доступе потоков к общим данным и о других аспектах. Читатель узнал, что любую сущность языка C++, выглядящую как функция, можно запустить на выполнение в отдельном потоке. Разобран ряд примеров того, как в потоках запускать различные виды сущностей: обычные функции, объекты типа `std::function`, лямбда-выражения и функциональные объекты. В этой главе было рассказано о примитивах синхронизации, поддерживаемых стандартной библиотекой: простейших двоичных семафорах `std::mutex` и об объектах-обёртках `std::lock_guard` и `std::unique_lock`, которые реализуют идиому RAII для захвата и автоматического освобождения семафоров, тем самым избавляя

¹ Это совершенно излишне. Довольно и одного метода, возвращающего само снятое со стека значение, т. е. метода `T wait_n_pop()`. В самом деле, передавать значение через выходной параметр нужно только для того, чтобы возвращаемым значением функции сделать признак удачи или неудачи; а возвращать умный указатель нужно лишь затем, чтобы в случае неудачи вернуть пустой указатель. У метода же, который всегда дожидается появления данных, неудачи быть не может. – *Прим. перев.*

программиста от необходимости освобождать семафоры своими руками. Также рассмотрены условные переменные (`std::condition_variable`), представляющие собой ещё одно средство синхронизации потоков. Таким образом, данная глава закладывает фундамент для понимания многопоточного программирования на языке C++ и готовит читателя к восприятию следующих глав, посвящённых идиомам функционального программирования.

В следующей главе речь пойдёт о других инструментах многопоточного и параллельного программирования, поддерживаемых стандартной библиотекой языка C++, таких как параллельная обработка на основе задач и неблокирующая синхронизация.



Глава 4

.....

Асинхронное программирование и неблокирующая синхронизация в языке C++



В предыдущей главе были рассмотрены средства управления потоками, поддерживаемые стандартной библиотекой языка C++, и различные способы создания потоков, управления ими и синхронизации. Однако организовывать архитектуру программы на основе потоков – это довольно низкоуровневый подход, а получаемый при этом код бывает подвержен разнообразным ошибкам: помимо изученных выше гонок и тупиков, упомянем ещё динамический тупик (англ. *livelock*) – ситуацию, при которой потоки не заблокированы и, на первый взгляд, продолжают работать, но при этом большую часть вычислительных ресурсов тратят на попытки синхронизировать своё выполнение, чем на полезные вычисления. Помимо потоков, современный стандарт языка C++ поддерживает также модели памяти с гарантиями, что позволяет совершенно иначе взглянуть на разработку параллельных программ. Для того чтобы язык программирования был параллельным в самой своей основе, он должен предоставлять разработчику определённые гарантии относительно доступа к памяти и порядка, в котором выполняются операции над ней. Если для синхронизации потоков пользоваться такими средствами, как семафоры, условные переменные или сигналы, смоделированные на основе фьючерсов, программисту нет нужды задумываться о модели памяти: порядок выполнения операций в этом случае задан в явном виде. Но, зная модели памяти и её гарантии, оказывается возможным сделать параллельный код более быстрым за счёт полного исключения блокировок и ожиданий. Блокировки удаётся заменить так называемыми атомарными операциями – этот приём будет рассмотрен ниже.

Как было отмечено в главе 2 «Современный язык C++ и его ключевые идиомы», нулевая стоимость абстракции была и остаётся одним из наиболее важных принципов языка C++. С момента своего возникновения это язык системного программирования, и комитет по стандартизации сумел сохранить равновесие между поддержкой высокоуровневых механизмов абстрагирования и способностью языка выражать управление ресурсами на низком уровне, что необходимо для написания системных программ. В частности, в стандартную библиотеку включены так называемые атомарные типы вместе с набором относящихся к ним операций, позволяющие управлять ходом выполнения программы с высокой степенью детализации. Комитет по стандартизации опубликовал подробное описание модели памяти, а язык пополнился набором библиотек, позволяющих программистам сполна извлекать из этого пользу.

В предыдущей главе было рассказано, как синхронизировать действия, выполняемые в разных потоках, с помощью условных переменных. В этой главе займёмся средствами из стандартной библиотеки для организации асинхронных вычислений на основе задач, в том числе так называемыми *фьючерсами*. Таким образом, в главе будут разобраны следующие темы:

- асинхронные вычисления на основе задач и их поддержка языком C++;
- модель памяти в языке C++;
- атомарные типы и атомарные операции;
- синхронизируемые операции и порядок доступа к памяти;
- разработка неблокирующих структур данных.

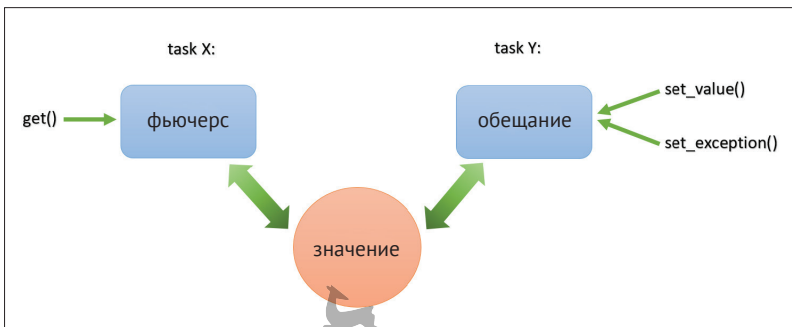
АСИНХРОННЫЕ ЗАДАЧИ В ЯЗЫКЕ C++

Задачей называют вычисление, которое потенциально может выполняться параллельно с другими вычислениями. Поток – это форма существования задачи в операционной системе. В предыдущей главе мы увидели, что задачу можно создать и сразу запустить на выполнение, сконструировав объект типа `std::thread`, – для этого вычисление, которое предстоит выполнять параллельно, нужно передать в конструктор данного объекта в качестве аргумента. Таким способом можно создать задачу из любой сущности, которую можно вызвать подобно функции: это может быть не только функция, но также функциональный объект или лямбда-выражение. Однако данный подход к организации параллельных вычислений на основе явного управления потоками слишком обременителен, так как потоки суть технические детали, от которых хотелось бы абстрагироваться. Предпочтителен был бы подход, при котором программист инициирует именно задачу, а система сама решает, создавать ли под неё новый поток и когда это делать. Программирование в терминах задач, по сравнению с программированием потоков, выполняется на более высоком концептуальном уровне и освобождает разработчика от таких подробностей, как управление потоками и блокировками. Для поддержки параллельных задач в стандартной библиотеке языка C++ существуют следующие средства:

- *фьючерсы* и *обещания* для работы с результатами выполнения параллельных задач;
- класс `packaged_task`, помогающий запускать задачи и получать результаты их выполнения;
- функция `async`, которая запуск задачи делает подобным вызову обычной функции.

Фьючерсы и обещания

Параллельные задачи в языке C++ ведут себя подобно каналам, по которым продвигаются данные. Через один конец, часто называемый **обещанием** (`promise`), данные поступают в канал, а после обработки достигают второго его конца, который называется **фьючерсом** (`future`). Важная черта фьючерсов и обещаний состоит в том, что они позволяют передавать данные от одной задачи к другой без явного использования блокировок или иных механизмов синхронизации. За передачу данных отвечает сама по себе система, т. е. среда выполнения. Идея, лежащая в основе такого способа сочленения задач, проста: когда задача желает передать вычисленное значение другой задаче, она отправляет его в обещание. Тогда внутренние механизмы стандартной библиотеки позаботятся о том, чтобы фьючерс, связанный с этим обещанием, получил данное значение. Затем иные задачи смогут, в свою очередь, читать значение из этого фьючерса. Работа механизма обещаний и фьючерсов схематически показана на следующем рисунке (рекомендуется рассматривать её справа налево).



Фьючерсы бывают особенно полезны, если задаче (или потоку) нужно ожидать *однократного* события. Чтобы через фьючерс просигнализировать о наступлении события, нужно, чтобы фьючерс выдал определённое значение, для этого значение нужно установить в соответствующее фьючерсу обещание. Тогда задача, ожидающая этот фьючерс, автоматически продолжит своё выполнение и получит переданное значение. Пока задача выполняется, её фьючерс может содержать или не содержать данные. В результате своего выпол-

нения задача может поместить значение во фьючерс (тем самым просигналив событие всем задачам, которые этого ожидают), и в дальнейшем оно не может быть изменено.

Шаблоны классов, предназначенные для программирования асинхронных задач, объявлены в стандартном заголовочном классе `<future>`. В стандартной библиотеке имеются фьючерсы двух разновидностей: единичные (тип `std::future<T>`) и множественные (тип `std::shared_future<T>`). Различие между ними примерно такое же, как между двумя видами умных указателей (типы `std::unique_ptr<T>` и `std::shared_ptr<T>`). Так, на один результат асинхронного вычисления (или, что то же самое, на одно событие) может ссылаться только один объект типа `std::future<T>`. В случае же типа `std::shared_future<T>`, напротив, сколько угодно объектов может быть связано с одним событием. При этом все объекты одновременно покажут готовность события, когда оно наступит. Параметр этих шаблонов – это тип данных, характеризующих событие (или, что то же самое, тип результата асинхронного вычисления). Специализации этих шаблонов типом `void` (т. е. типы `std::future<void>` и `std::shared_future<void>`) стоит использовать, если никакие данные с событием не связаны (иными словами, если сам факт наступления события и составляет все данные). Несмотря на то что фьючерсы используются для обмена данными между потоками, сами эти объекты не занимаются синхронизацией доступа к себе со стороны потоков. Если несколько потоков хотят обратиться к одному объекту `std::future`, они должны защитить объект семафором или иными механизмами синхронизации.

Классы `std::future` и `std::promise` работают в паре в процессе выполнения задачи и ожидания её результатов. Если дан объект `f` типа `std::future<T>`, к связанному с ним объекту типа `T` (результату задачи) можно обратиться с помощью метода `get` класса `std::future<T>`. Симметричным образом, имея объект типа `std::promise<T>`, можно установить ему значение с помощью метода `set_value` или установить состояние ошибки с помощью метода `set_exception`, в обоих случаях установленное состояние станет доступно через метод `get` соответствующего фьючерса. Следующий пример демонстрирует, как поместить значение в объект-обещание (в функции `func1`) и как это значение извлекается из объекта-фьючерса (в функции `func2`).

```
// обещание pr связано с запущенной задачей
void func1(std::promise<T>& pr)
{
    try
    {
        T val;
        process_data(val); // вычислить значение
        // значение будет получено через future<T>::get()
        pr.set_value(val);
    }
    catch(...)
    {
```

```

        // исключение будет получено в future<T>::get()
        pr.set_exception(std::current_exception());
    }
}

```

В этом фрагменте кода сначала делается попытка каким-то образом вычислить требуемое значение – оно помещается в локальную переменную `val` типа `T`. Если это удалось (т. е. если из функции `process_data` не было выброшено исключение), это значение устанавливается в объект-обещание `pr`. Если возникает исключение, оно тоже отправляется в этот объект. Рассмотрим теперь, как можно применить результат вычисления.

```

// фьючерс ft связан с уже запущенной задачей
void func2(std::future<T>& ft)
{
    try
    {
        // если соответствующему обещанию установлено состояние
        // ошибки, следующий вызов выбросит исключение, иначе –
        // вернёт установленное в обещании значение
        T result = ft.get();
        handle_value(result); // обработать полученные данные
    }
    catch(...)
    {
        // обработать исключение
    }
}

```



В этой функции делается попытка получить из фьючерса значение – результат выполнения соответствующей ему задачи. Если её обещанию было установлено значение, метод `get` отработает нормальным образом и вернёт это значение, которое можно дальше обрабатывать. Если же в обещание было помещено исключение, оно и будет выброшено методом `get`, и в нашей функции должна быть предусмотрена его обработка. В следующем разделе будет сказано о классе `std::packaged_task`, после этого можно будет завершить данный пример, показав, как связать воедино обещание и фьючерс.

Класс `std::packaged_task`

Теперь пришло время показать, как создать асинхронно выполняемую задачу и через фьючерс получить результат её выполнения. Для запуска задач и последующей работы с ними посредством фьючерсов в стандартной библиотеке служит класс-шаблон `std::packaged_task`. Он берёт на себя создание фьючерса, связанного с задачей, и позволяет запускать её в отдельном потоке, освобождая программиста от необходимости своими руками управлять блокировками для доступа к результату вычислений. Иначе говоря, объект типа `std::packaged_task` представляет собой обёртку над функцией, которую нужно выполнить, и объектом-обещанием. Данная обёртка управляет сохранением в обещании

результата работы функции, будь то вычисленное ею значение или выброшенное исключение. Метод `get_future` класса `std::packaged_task` отдаёт фьючерс, связанный с этим обещанием. Клиентскому коду доступен именно фьючерс – работа с объектом-обещанием скрыта глубоко в реализации самой обёртки. Рассмотрим следующий пример, где с помощью объекта `std::packaged_task` создаётся асинхронная задача на основе функции, вычисляющей сумму элементов контейнера.

```
// вычислить сумму элементов вектора
int calc_sum(std::vector<int> v)
{
    int sum = std::accumulate(v.begin(), v.end(), 0);
    return sum;
}

int main()
{
    // создать задачу на основе функции
    std::packaged_task<int(std::vector<int>)> task(calc_sum);

    // получить фьючерс этой задачи
    std::future<int> result = task.get_future();

    std::vector<int> nums{1,2,3,4,5,6,7,8,9,10};

    // задачу запустить в потоке, передав исходные данные
    std::thread t(std::move(task), std::move(nums));

    t.join();

    // получить результат выполнения асинхронной задачи
    int sum = result.get();

    std::cout << "Сумма " << sum << std::endl;
    return 0;
}
```

Шаблон класса `std::packaged_task` принимает в качестве параметра тип функции, на основе которой предполагается создавать задачу. В конструктор экземпляра этого класса передаётся указатель на функцию. Затем из сконструированной задачи можно извлечь её фьючерс с помощью метода `get_future`. При создании потока на основе задачи использована функция `std::move` – экземпляры класса `std::packaged_task` нельзя копировать, а можно лишь перемещать. Дело в том, что в объекты данного класса завёрнуты системные ресурсы, за освобождение которых отвечает деструктор, поэтому лишь один объект может ими владеть. При создании потока ему в конструктор вместо указателя на функцию передаётся созданная на её основе задача, в остальном запуск потока выглядит так же, как и раньше. Наконец, метод `get` получает результат выполнения асинхронной задачи, он и выводится на печать.

С таким же успехом объекты класса `std::packaged_task` можно создавать и на основе лямбда-выражений, что может оказаться удобным, если в асинхронном режиме требуется запустить небольшой блок кода:

```
std::packaged_task<int(std::vector<int>)> task(
    [](std::vector<int> v) {
        return std::accumulate(v.begin(), v.end(), 0);
    });
```

Основное предназначение фьючерсов состоит в том, чтобы клиентский код мог запрашивать результат асинхронно выполняемой задачи, не утруждая себя механизмами синхронизации потоков. В данном примере в отдельном потоке выполняется функция вычисления суммы, а главный поток пользуется вычисленным ею значением.

Функция `std::async`

В современном стандарте языка C++ имеется механизм, позволяющий помещать вызов функции в отдельную задачу, которая может выполняться как параллельно, так и в потоке, запрашивающем её результат. Это функция-шаблон `std::async`, которая скрывает от пользователя внутренние механизмы управления потоками. Функция `std::async` принимает объект, допускающий вызов (от указателя на функцию до объекта пользовательского класса с перегруженной операцией вызова) и возвращает объект класса `std::future`, в котором сохраняется результат вызова или выброшенное из него исключение. Давайте перепишем наш предыдущий пример с асинхронным вычислением суммы элементов вектора, в этот раз используя функцию `std::async`.

```
// вычислить сумму элементов вектора
int calc_sum(std::vector<int> v)
{
    int sum = std::accumulate(v.begin(), v.end(), 0);
    return sum;
}

int main()
{
    std::vector<int> nums{1,2,3,4,5,6,7,8,9,10};

    // запустить асинхронную задачу и получить её фьючерс
    std::future<int> result(std::async(
        std::launch::async, calc_sum, std::move(nums)));

    // получить результат выполнения асинхронной задачи
    int sum = result.get();

    std::cout << "Сумма " << sum << std::endl;
    return 0;
}
```

Главная выгода от использования функции `std::async` для асинхронных задач состоит в том, что запуск задачи и получение её результата весьма просто

оформляются в тексте программы и хорошо отделяются от подробностей её выполнения. Как видно из приведённого выше кода, функция `std::async` принимает следующие аргументы:

- флаг, определяющий политику запуска асинхронной задачи. В данном примере значение `std::launch::async` означает, что задачу сразу после создания нужно запустить в отдельном потоке выполнения. Если же вместо него указать флаг `std::launch::deferred`, то новый поток не создаётся, вместо этого во фьючерс помещается так называемое отложенное, или ленивое, вычисление: задача будет выполнена лишь тогда, когда клиентский код вызовет метод `get` её фьючерса. Если при создании задачи указать комбинацию обоих этих флагов, то система сама решит, запустить задачу в новом потоке или превратить её в отложенное вычисление. Наконец, если вообще не указывать этот аргумент, это также означает, что политику запуска выберет система;
- второй аргумент – это то, что должно выполняться в виде задачи. Это может быть указатель на функцию, лямбда-выражение, объект любого типа с перегруженной операцией вызова. В данном примере задача создаётся на основе функции `calc_sum`;
- далее следуют в любом количестве аргументы для функции (или функционального объекта). В данном примере для асинхронного запуска функции нужен один аргумент – контейнер, сумму элементов которого предстоит вычислить.

Конечно, этот пример можно было бы реализовать и с использованием лямбда-выражения:

```
std::future<int> result(async(
    std::launch::async,
    [](std::vector<int> v)
    {
        return std::accumulate(v.begin(), v.end(), 0);
    },
    std::move(nums)));
```

Лямбда-выражение, занявшее место указателя на функцию, просто возвращает результат функции `std::accumulate`. Если асинхронная задача достаточно проста, этой возможностью стоит воспользоваться, чтобы сделать код изящнее.

Программист, использующий в своём коде асинхронные вызовы, более не обязан вникать в подробности потоков и блокировок: внутренние механизмы стандартной библиотеки делают эту работу сами. Взамен программист получает свободу мыслить на более высоком уровне абстракции: какие задачи должны быть выполнены программой в том или ином порядке, с той или иной степенью распараллеливания, с каким угодно фактическим числом потоков. Система сама определит наилучшее число потоков, исходя из числа имеющихся в наличии процессорных ядер и степени их загруженности. Однако с этим связано и очевидное ограничение на использование асинхронных задач, в случае если задачи обращаются к общему ресурсу, требующему блокировки.

Модель памяти в языке C++

Язык C++ с момента своего возникновения был однопоточным языком. Хотя программисты и создавали на нём многопоточные программы, но для этого им приходилось пользоваться низкоуровневыми средствами конкретных платформ – сам по себе язык поддержки потоков не предоставлял. Современный язык C++ можно с полным основанием назвать языком многопоточного, параллельного и асинхронного программирования. Стандартом языка определены механизмы управления потоками и задачами, оформленные в виде библиотечных функций и классов, – со многими из них читатель познакомился в этой и предыдущей главах. Поскольку данные средства входят в стандартную библиотеку, в спецификации языка строго определено, как они должны вести себя, независимо от платформы и среды выполнения. Выработка единого, логически безупречного и не зависящего от платформы аппарата потоков, задач, примитивов синхронизации и т. д. – грандиозная задача, и комитет по стандартизации справился с ней великолепно. В рамках этой задачи комитет разработал и описал стандартную модель памяти, на которой только и может основываться единообразное поведение программы на разных платформах. Модель памяти состоит из двух аспектов:

- **структурного**, к которому относится способ расположения данных в памяти;
- **динамического**, к которому относится порядок доступа к памяти и выполнения операций над хранящимися в ней данными – в том числе из параллельных потоков.

С точки зрения языка C++, все данные состоят из *объектов*. Спецификация языка определяет объект как *область в памяти*, обладающую определённым типом и временем жизни. Объекты могут принадлежать как встроенным типам (таким как типы `int`, `double` или тип указателя), так и типам, которые объявил программист. Некоторые объекты содержат в себе подобъекты, иные их не содержат. При всех различиях, однако, неизменным остаётся главное: значение любой переменной есть некоторый объект (это относится в том числе к переменным, которые сами являются членами объектов), а всякий объект располагается в определённом месте в памяти. Рассмотрим, что это означает для параллельного программирования.

Параллельный доступ к памяти

Для многопоточных приложений всё зависит от того, к каким участкам памяти обращаются их потоки. Если потоки всегда имеют дело каждый со своей областью памяти, всё прекрасно. Но если хотя бы два потока хотя бы иногда обращаются к одному и тому же месту в памяти, программисту нужно быть очень осторожным. Как мы уже знаем из главы 3, попытки одновременного чтения разными потоками из одного участка памяти ещё не могут привести к проб-

лемам, однако как только один из потоков пытается изменить общие данные, доступные для чтения другому потоку, появляется опасность *гонки потоков*.

Неприятностей, связанных с гонками, можно избежать только одним способом: заставить потоки обращаться к общему участку памяти в определённом порядке. В главе 3 рассказывалось об одном из самых распространённых механизмов – блокировке с использованием семафоров. Другой способ, которому и посвящён данный раздел, состоит в том, чтобы воспользоваться атомарностью некоторых простейших операций над простейшими типами данных и с их помощью организовать порядок доступа потока к данным. В последующих разделах читатель увидит, что на основе атомарных операций можно даже смоделировать высокоуровневые примитивы синхронизации.

i Атомарной называют операцию, которая выполняется целиком, до конца, не может быть прервана посередине переключением контекста и даже при наличии других параллельных потоков защищена от их вмешательства. Таким образом, атомарная операция полностью защищена от прерываний, сигналов, других процессов и потоков. Читатель может узнать больше из следующей статьи: <https://en.wikipedia.org/wiki/Linearizability>.

Следует подчеркнуть, что если порядок доступа потоков к общей памяти не гарантирован никакими специальными механизмами (например, семафорами), атомарные операции сами по себе никак не защищают от гонки данных, то есть от ситуации, когда два потока вносят несовместимые изменения в разные участки сложной структуры памяти. Атомарные операции предохраняют от повреждения (в частности, от неопределённого значения, которое может возникнуть при одновременном выполнении двух операций присваивания) лишь отдельно взятую переменную достаточно простого типа (как правило, целочисленную).

СОГЛАШЕНИЕ О ПОРЯДКЕ МОДИФИКАЦИИ ПАМЯТИ

В течение всего времени выполнения программы все её потоки обязаны соблюдать определённые соглашения о порядке модификаций, вносимых ими в память. Всякая программа выполняется в среде, включающей в себя поток команд, регистры процессора, динамически распределяемую память, стек, буферы памяти, виртуальную память и т. д. Соглашение о порядке модификации – это своего рода договор между программистом и этой системой, который определяется моделью памяти. Со стороны системы за соблюдение договора отвечают компилятор (вместе с редактором связей), который превращает текст программы в исполняемый машинный код, процессор, который исполняет поток машинных команд, кеш и механизмы виртуальной памяти. Программиста договор обязывает соблюдать при написании программ определённые правила – это открывает системе возможность глубокой оптимизации кода. В свою очередь, для соблюдения этого набора правил по работе с памятью программисту приходят на помощь атомарные типы и атомарные операции, появившиеся в стандартной библиотеке.

Эти операции не просто атомарны – они синхронизируют выполнение программы и накладывают на него определённые ограничения. По сравнению с блокирующими высокоуровневыми примитивами синхронизации (такими как двоичные семафоры и условные переменные), о которых шла речь в главе 3, использование модели памяти позволяет в широких пределах варьировать степень синхронизации и налагаемые ею ограничения, подстраивая их под свои потребности. Пожалуй, главное, что нужно вынести из изучения модели памяти в языке C++, состоит в следующем: несмотря на то что язык C++ пополнился множеством современных идиом и конструкций, он был и остаётся ещё и языком системного программирования, и в этом качестве язык получил инструменты для управления доступом к памяти на низком уровне, позволяя тем самым создавать чрезвычайно оптимизированные программы.

АТОМАРНЫЕ ОПЕРАЦИИ И ТИПЫ В ЯЗЫКЕ C++

Неатомарная операция в общем случае может быть прервана на середине переключением потоков – тогда другие потоки увидят данные в промежуточном и, как правило, некорректном состоянии. Как отмечалось в главе 3, инварианты структур данных, совместно используемых несколькими потоками, в таких случаях оказываются нарушенными. Это особенно характерно для таких операций над структурой данных, которые включают модификацию нескольких простых значений. Хорошим примером может служить наполовину выполненное удаление элемента из сбалансированного двоичного дерева. Если другой поток вклинивается посередине этой операции и пытается читать данные из этого дерева (или, того хуже, внести в него своё изменение), результат может быть совершенно непредсказуемым.

Атомарными называются такие операции, которые не могут быть прерваны до полного завершения, – таким образом, никакой другой поток не может увидеть промежуточный результат атомарной операции. Если любая операция над объектами некоторого типа атомарна, то и сам тип называют атомарным. Язык C++ получил поддержку атомарных типов на уровне стандартной библиотеки.

АТОМАРНЫЕ ТИПЫ

Включённые в стандартную библиотеку средства для поддержки атомарных типов можно найти в заголовочном файле `<atomic>`. Любая реализация языка обязана гарантировать атомарность всех операций над объектами этих типов. Даже если аппаратная архитектура не поддерживает атомарность операций для тех или иных типов данных, реализация стандартной библиотеки должна собственными механизмами её обеспечить. Атомарные типы из стандартной библиотеки обладают статической функцией-членом `is_lock_free`, которая возвращает логическое значение `true` (истина), если атомарность этого типа поддерживается аппаратно, т. е. операции над этим типом напрямую реали-

зованы посредством атомарных машинных команд; в противном же случае, если функция `is_lock_free` возвращает значение `false`, атомарность операций над объектами данного типа обеспечивается с помощью блокирующих примитивов синхронизации.

Исключением из общего правила является тип¹ `std::atomic_flag`. Стандарт требует истинной, т. е. аппаратно поддерживаемой, атомарности этого типа. Поэтому функция `is_lock_free` для него не определена за ненадобностью. С другой стороны, это весьма простой тип с очень бедным набором операций: это лишь метод `clear` (сбросить флаг) и метод `test_and_set` (установить флаг и вернуть его предыдущее значение).

Остальные атомарные типы представлены в стандартной библиотеке как специализации шаблона `std::atomic<>` типом-параметром, атомарность операций над которым нужно обеспечить. По сравнению с типом `std::atomic_flag`, набор операций у этих типов заметно богаче, однако не у всех из них гарантируется аппаратная поддержка атомарности, свободная от блокирующей синхронизации. Наличие аппаратной атомарности зависит от целевой платформы. Впрочем, на большинстве распространённых платформ присутствует аппаратная поддержка операций над встроенными в язык типами.

Вместо шаблона `std::atomic<>` с типом-параметром можно также использовать соответствующие псевдонимы, показанные в следующей таблице.

Псевдоним	Специализация шаблона
<code>atomic_bool</code>	<code>std::atomic<bool></code>
<code>std::atomic_char</code>	<code>std::atomic<char></code>
<code>std::atomic_schar</code>	<code>std::atomic<signed char></code>
<code>std::atomic_uchar</code>	<code>std::atomic<unsigned char></code>
<code>std::atomic_short</code>	<code>std::atomic<short></code>
<code>std::atomic_ushort</code>	<code>std::atomic<unsigned short></code>
<code>std::atomic_int</code>	<code>std::atomic<int></code>
<code>std::atomic_uint</code>	<code>std::atomic<unsigned int></code>
<code>std::atomic_long</code>	<code>std::atomic<long></code>
<code>std::atomic_ulong</code>	<code>std::atomic<unsigned long></code>
<code>std::atomic_llong</code>	<code>std::atomic<long long></code>
<code>std::atomic_ullong</code>	<code>std::atomic<unsigned long long></code>
<code>std::atomic_char16_t</code>	<code>std::atomic<char16_t></code>
<code>std::atomic_char32_t</code>	<code>std::atomic<char32_t></code>
<code>std::atomic_wchar_t</code>	<code>std::atomic<wchar_t></code>

Помимо этих атомарных обёрток над всеми встроенными типами, стандартная библиотека содержит ещё и обёртки (как в форме специализаций

¹ Под флагом в программировании понимают переменную логического типа, причём значение «истина» называют установленным флагом, а значение «ложь» – сброшенным. – *Прим. перев.*

шаблона, так и в виде псевдонимов) над числовыми типами с фиксированной разрядностью (например, `std::uint32_t`) и другими числовыми типами, объявленными в стандартной библиотеке (например, `std::size_t`). Имена атомарных типов образованы по единому образцу: к имени стандартного числового типа спереди прибавляется префикс `atomic_`. Атомарные версии стандартных числовых типов перечислены в следующей таблице.

Псевдоним	Специализация шаблона
<code>std::atomic_size_t</code>	<code>std::atomic<size_t></code>
<code>std::atomic_intptr_t</code>	<code>std::atomic<intptr_t></code>
<code>std::atomic_uintptr_t</code>	<code>std::atomic<uintptr_t></code>
<code>std::atomic_ptrdiff_t</code>	<code>std::atomic<ptrdiff_t></code>
<code>std::atomic_intmax_t</code>	<code>std::atomic<intmax_t></code>
<code>std::atomic_uintmax_t</code>	<code>std::atomic<uintmax_t></code>
<code>std::atomic_int_least8_t</code>	<code>std::atomic<int_least8_t></code>
<code>std::atomic_uint_least8_t</code>	<code>std::atomic<uint_least8_t></code>
<code>std::atomic_int_least16_t</code>	<code>std::atomic<int_least16_t></code>
<code>std::atomic_uint_least16_t</code>	<code>std::atomic<uint_least16_t></code>
<code>std::atomic_int_least32_t</code>	<code>std::atomic<int_least32_t></code>
<code>std::atomic_uint_least32_t</code>	<code>std::atomic<uint_least32_t></code>
<code>std::atomic_int_least64_t</code>	<code>std::atomic<int_least64_t></code>
<code>std::atomic_uint_least64_t</code>	<code>std::atomic<uint_least64_t></code>
<code>std::atomic_int_fast8_t</code>	<code>std::atomic<int_fast8_t></code>
<code>std::atomic_uint_fast8_t</code>	<code>std::atomic<uint_fast8_t></code>
<code>std::atomic_int_fast16_t</code>	<code>std::atomic<int_fast16_t></code>
<code>std::atomic_uint_fast16_t</code>	<code>std::atomic<uint_fast16_t></code>
<code>std::atomic_int_fast32_t</code>	<code>std::atomic<int_fast32_t></code>
<code>std::atomic_uint_fast32_t</code>	<code>std::atomic<uint_fast32_t></code>
<code>std::atomic_int_fast64_t</code>	<code>std::atomic<int_fast64_t></code>
<code>std::atomic_uint_fast64_t</code>	<code>std::atomic<uint_fast64_t></code>

Помимо специализаций шаблона `std::atomic<>` для множества стандартных числовых типов, у него есть и общее определение, куда в качестве параметра можно подставить любой пользовательский тип данных. Имеется также специализация для любых типов указателей. Набор операций, поддерживаемых в общем случае (т. е. когда шаблон параметризован пользовательским типом), невелик: это метод `load` (получить текущее значение из атомарной обёртки), метод `store` (поместить новое значение в атомарную обёртку), `exchange` (обменять значения в атомарной и обычной переменной) и методы `compare_exchange_weak` и `compare_exchange_strong` (сравнить значение атомарной переменной со значением обычной переменной и, если они совпали, поместить в атомарную переменную новое значение – всё за одну атомарную операцию). У всех атомарных операций есть необязательный аргумент, управляющий порядком доступа к памяти: как обычные, неатомарные операции доступа к данной переменной должны упорядочиваться вокруг атомарной операции. Модели памяти

и упорядочивание доступа будут подробнее разобраны в последующих разделах этой главы. Сейчас достаточно иметь в виду, что атомарные операции делятся на три разновидности:

- операции записи;
- операции чтения;
- операции чтения-модификации-записи.

С каждым видом атомарных операций есть своё подмножество допустимых способов упорядочивания доступа. Если данный параметр не указывать, по умолчанию принимается самый сильный в смысле предоставляемых гарантий, но в то же время и наихудший в смысле быстродействия метод упорядочивания доступа.

В отличие от фундаментальных типов языка C++, соответствующие атомарные типы-обёртки не поддерживают копирование и присваивание. Иными словами, для этих типов в явном виде запрещены конструкторы копирования и операции присваивания. Дело в том, что в копировании или присваивании участвуют два объекта, а атомарность может быть аппаратно гарантирована только для операций над одним объектом. В самом деле, для копирования или присваивания нужно получить значение из одного атомарного объекта и записать его во второй атомарный объект. Комбинация из двух атомарных операций сама не обязана быть атомарной, так как другой поток может вклиниться между этими операциями. У атомарных типов имеются операции неявного приведения к соответствующим неатомарным фундаментальным типам.

Рассмотрим теперь подробно, какие операции можно выполнять над каждым из стандартных атомарных типов, начиная с типа `std::atomic_flag`.

Тип `std::atomic_flag`

Тип `std::atomic_flag` моделирует флаг, т. е. логическое значение, и является простейшим из всех имеющихся в стандартной библиотеке атомарных типов. Это единственный атомарный тип, для которого гарантирована неблокирующая реализация всех операций на любой платформе. С другой стороны, этот тип обладает очень бедным набором операций, поэтому его стоит использовать лишь в качестве строительного блока для создания более сложных конструкций.

Объект типа `std::atomic_flag` всегда нужно инициализировать константой `ATOMIC_FLAG_INIT`, чтобы привести флаг в сброшенное состояние:

```
std::atomic_flag flg = ATOMIC_FLAG_INIT;
```

Это единственный атомарный тип в стандартной библиотеке, объекты которого нуждаются в подобной инициализации, где бы они ни были объявлены. Когда объект проинициализирован, над ним можно выполнять лишь три операции: уничтожить объект, сбросить флаг и (за одно атомарное действие) установить флаг и вернуть его прежнее значение. Это, соответственно, деструктор, метод `clear` и метод `test_and_set`. С точки зрения классификации, описанной

в конце предыдущего раздела, метод `clear` представляет собой операцию записи, тогда как метод `test_and_set` есть операция чтения-модификации-записи.

```
flg.clear();
bool val = flg.test_and_set(std::memory_order_relaxed);
```

В этом фрагменте кода вызов метода `clear` сбрасывает флаг (т. е. записывает в него логическое значение «ложь»), используя порядок доступа к памяти по умолчанию (а именно порядок `std::memory_order_seq_cst`, самый сильный из возможных, – он гарантирует не только атомарность, но и согласованную последовательность операций чтения и записи, см. ниже). Вызов метода `test_and_set` в следующей строке содержит аргумент, явно указывающий слабый (и более быстрый) порядок доступа к памяти: от операции требуется лишь атомарность, а на последовательность обращений к памяти никакие условия не налагаются.

Предельная простота типа `std::atomic_flag` и его операций делает его идеальным строительным материалом для так называемого цикла ожидания (англ. *spin-lock*). Это примитив синхронизации, который ведёт себя подобно двоичному семафору, т. е. обладает операциями захвата и освобождения; если захват уже выполнен, повторная операция захвата блокирует поток до тех пор, пока не произойдёт освобождение. В данном случае операция захвата представляет собой цикл, который постоянно проверяет флаг состояния блокировки – отсюда и термин «цикл ожидания».

```
class spin_lock
{
    std::atomic_flag flg;

public:
    spin_lock() : flg(ATOMIC_FLAG_INIT)
    {}

    void lock()
    {
        // вход в критическую секцию: выполнять цикл, пока флаг
        // "занято" не сброшен, и сразу снова установить его
        while (flg.test_and_set(std::memory_order_acquire));
    }

    void unlock()
    {
        // выход из критической секции: сбросить флаг "занято"
        flg.clear(std::memory_order_release);
    }
};
```

Переменная-член `flg` типа `std::atomic_flag` отвечает за состояние объекта: занят или свободен, сразу после создания объект находится в состоянии «свободен». В методе `lock` цикл будет выполняться до тех пор, пока флаг показывает, что объект занят. На каждой итерации цикла не делается никаких иных

операций, кроме проверки значения флага. Если поток, который в настоящее время держит блокировку, вызовет метод `clear` и тем самым сбросит флаг, операция `test_and_set` вернёт значение «ложь» (что обеспечит выход из цикла ожидания) и одновременно с этим снова установит флаг в значение «истина», т. е. «занято». Таким образом, наш класс действительно позволяет добиться взаимной блокировки потоков.

В силу крайней ограниченности набора операций тип `std::atomic_flag` не может выполнять роль атомарного логического типа. В частности, в нём нет операции, которая бы просто получала текущее значение флага, не модифицируя его. Поэтому перейдём к типу `std::atomic<bool>`, который поддерживает все операции, характерные для логического типа.

Тип `std::atomic<bool>`

В отличие от типа `std::atomic_flag`, тип `std::atomic<bool>` представляет собой полноценный логический тип. Однако ни копирование, ни присваивание объектов этого типа невозможно. Объект данного типа можно инициализировать значением `true` или `false`. В дальнейшем такому объекту можно присвоить значение обычного (неатомарного) типа `bool`:

```
std::atomic<bool> flg(true);
flg = false;
```

Следует отметить одну важную особенность операции присваивания, общую для всех атомарных типов: в отличие от операций присваивания для обычных типов, она возвращает не ссылку на объект из левой части операции, а присвоенное значение неатомарного типа. В самом деле, если бы результатом присваивания была ссылка, выполнивший присваивание поток мог бы увидеть значение, присвоенное каким-то другим потоком, что противоречит самому понятию атомарной операции. Возврат значения неатомарного типа, напротив, устраняет нежелательное взаимное влияние потоков подобного рода: программист может быть уверен, что значение, возвращённое операцией присваивания, – это то же самое значение, которое только что было присвоено атомарному объекту.

Перейдём к рассмотрению операций, поддерживаемых типом `std::atomic<bool>`. Прежде всего это операция `store`, которая записывает в атомарный объект новое значение `true` или `false` (для сравнения, операция `clear`, определённая для типа `std::atomic_flag`, позволяет записать в объект лишь значение `false`). Конечно же, эта операция записи атомарна. Подобным же образом вместо операции `test_and_set`, определённой для типа `std::atomic_flag`, тип `std::atomic<bool>` обладает более общим методом `exchange`, который заменяет хранившееся в атомарном объекте значение любым новым и возвращает старое значение. Этот метод представляет собой атомарную операцию вида «чтение-модификация-запись». Кроме того, тип `std::atomic<bool>` поддерживает простой, немодифицирующий запрос текущего значения, хранящегося

в атомарном объекте, – это метод `load`, являющийся, согласно нашей классификации, операцией чтения.

```
std::atomic<bool> flg;
flg.store(true);
bool val = flg.load(std::memory_order_acquire);
val = flg.exchange(false, std::memory_order_acq_rel);
```



Помимо этого, в типе `std::atomic<bool>` имеется ещё одна операция чтения-модификации-записи, реализующая широко распространённую идею **сравнения и обмена** (англ. compare-and-swap, CaS). Эта операция записывает в атомарный объект новое (т. н. желаемое) значение только в том случае, если его текущее значение равняется значению некоторой переменной (т. н. ожидаемому значению). Если же текущее значение атомарного объекта отлично от ожидаемого, текущее значение присваивается той переменной, в которой хранилось ожидаемое значение. Атомарные типы из стандартной библиотеки содержат два метода, реализующих данную операцию, – это методы `compare_exchange_weak` и `compare_exchange_strong`, называемые соответственно слабой и сильной операциями. Оба метода для всех атомарных типов возвращают значение логического типа: значение `true`, если запись произведена, и значение `false` в противном случае.

Метод `compare_exchange_weak` может завершиться неудачей (т. е. не выполнить запись нового значения и вернуть значение `false`), даже если текущее значение атомарного объекта совпадает с ожидаемым. Это может произойти на некоторых аппаратных платформах, где отсутствует процессорная инструкция атомарного сравнения и обмена, т. е. если процессор не может гарантировать выполнения всей этой сложной операции как единого целого. Когда операционной системе приходится на такой платформе обслуживать больше потоков, чем имеется в наличии процессоров, поток, пытающийся выполнить сравнение и обмен, может быть прерван переключением на другой поток посередине операции (скажем, когда сравнение уже выполнено, а обмен – ещё нет). Эта ситуация известна как **ложный отказ**.

Поскольку операция `compare_exchange_weak` может завершиться ложным отказом, её часто помещают в цикл, как в следующем примере:

```
bool expected = false;
std::atomic<bool> flg;
...
while (!flg.compare_exchange_weak(expected, true));
```

Цикл в этом примере продолжает выполнять сравнение и обмен до тех пор, пока переменная `expected` не изменит своё значение с `false` на `true`, преодолев, возможно, неоднократные ложные отказы.

Метод `compare_exchange_strong`, напротив, возвращает значение `false` лишь в том случае, если текущее значение атомарного объекта отличалось от ожидаемого. Это позволяет избежать циклов, подобных показанному выше, с многократными попытками установить атомарной переменной новое значение.

Сильные и слабые операции сравнения и обмена могут принимать два необязательных аргумента, задающих порядок доступа к памяти в двух случаях: при успешном и при неуспешном сравнении текущего значения с ожидаемым.

```
bool expected = false;
std::atomic<bool> flg;
flg.compare_exchange_weak(
    expected,
    true,
    std::memory_order_acq_rel,
    std::memory_order_acquire);
flg.compare_exchange_weak(
    expected,
    true,
    std::memory_order_release);
```

Если никакой порядок доступа к памяти явно не задан, по умолчанию для успешного и неуспешного сравнений используется порядок `std::memory_order_seq_cst`. Если указан порядок доступа лишь для случая успешного сравнения, он же используется и при неуспехе, с тем лишь отличием, что семантика освобождения (*release*) при этом игнорируется. Так, например, порядок `std::memory_order_acq_rel` превратится в порядок `std::memory_order_acquire`, а порядок `std::memory_order_release` – в порядок `std::memory_order_relaxed`.

Точные определения всех порядков доступа к памяти и их особенности будут подробно изложены позже в этой главе. Сейчас, однако, рассмотрим группу атомарных целочисленных типов.

Стандартные атомарные целочисленные типы

Подобно типу `std::atomic<bool>`, атомарные целочисленные типы из стандартной библиотеки не поддерживают ни копирование, ни присваивание объектов. Вместо этого их можно инициализировать значениями соответствующих неатомарных типов и можно им присваивать такие значения. Помимо непременно для всех атомарных типов метода `is_lock_free`, стандартные атомарные целочисленные типы, такие как `std::atomic<int>`, `std::atomic<unsigned long long>`, также обладают методами `load`, `store`, `exchange`, `compare_exchange_weak` и `compare_exchange_strong` с такой же семантикой, как в типе `std::atomic<bool>`.

Целочисленные атомарные типы поддерживают разнообразные арифметические операции, такие как `fetch_add`, `fetch_sub`, `fetch_and`, `fetch_or` и `fetch_xor`, комбинированные операции с присваиванием (`+=`, `-=`, `&=` и `^=`), а также префиксные и постфиксные операции инкремента и декремента `++` и `--`.

Арифметические функции, такие как `fetch_add` и `fetch_sub`, выполнив нужное арифметическое действие, возвращают старое значение атомарного объекта – в отличие от соответствующих перегруженных комбинированных операций с присваиванием (например, `+=` и `-=`), которые возвращают новое значение. Операции пре- и постинкремента и декремента следуют обычным для языков C и C++ соглашениям: постфиксные варианты этих операций возвращают старое значение объекта, а префиксные – новое. Следующий простой пример

демонстрирует свойства различных операций над атомарными целочисленными объектами.

```
int main()
{
    std::atomic<int> value;

    std::cout << "Возвращённое значение: " << value.fetch_add(5) << '\n';
    std::cout << "Значение после операции: " << value << '\n';

    std::cout << "Возвращённое значение: " << value.fetch_sub(3) << '\n';
    std::cout << "Значение после операции: " << value << '\n';

    std::cout << "Возвращённое значение: " << value++ << '\n';
    std::cout << "Значение после операции: " << value << '\n';

    std::cout << "Возвращённое значение: " << ++value << '\n';
    std::cout << "Значение после операции: " << value << '\n';

    value += 1;
    std::cout << "Значение после операции: " << value << '\n';

    value -= 1;
    std::cout << "Значение после операции: " << value << '\n';
    return 0;
}
```

Результат выполнения этого кода должен выглядеть следующим образом:

```
Возвращённое значение: 0
Значение после операции: 5
Возвращённое значение: 5
Значение после операции: 2
Возвращённое значение: 2
Значение после операции: 3
Возвращённое значение: 4
Значение после операции: 4
Значение после операции: 5
Значение после операции: 4
```

Все типы, перечисленные выше в таблице, за исключением типов `std::atomic_flag` и `std::atomic<bool>`, – это атомарные целочисленные типы. Рассмотрим теперь специализацию атомарного шаблона для типа указателя, т. е. семейство типов `std::atomic<T*>`.

Тип `std::atomic<T*>` и арифметика указателей

Помимо обычного для всех атомарных типов набора операций `load`, `store`, `exchange`, `compare_exchange_weak` и `compare_exchange_strong`, атомарные типы указателей содержат также характерные для типов указателей арифметические операции. Методы `fetch_add` и `fetch_sub` обеспечивают атомарное прибавление целого числа к указателю и вычитание числа из указателя, как и соответствующую

щие им перегруженные операции `+=` и `-=`, а также операции пре- и постинкремента и декремента `++` и `--`.

Эти операции работают точно так же, как и операции арифметики обычных, неатомарных указателей. Например, пусть объект `obj` типа `std::atomic<some_class*>` указывает на первый элемент массива объектов класса `some_class`. Тогда операция `obj+=2` меняет значение этого объекта таким образом, что отныне он указывает на третий элемент массива и возвращает обычный (неатомарный) указатель типа `some_class*` на этот объект. Как отмечалось в предыдущем разделе, посвящённом атомарным целочисленным типам, функции наподобие `fetch_add` и `fetch_sub`, изменив значение атомарного объекта, возвращают его старое значение, т. е. в данном примере они вернули бы указатель на первый элемент массива.

Атомарные операции, оформленные в виде функций, также позволяют программисту задавать отдельным аргументом порядок доступа к памяти, например:

```
obj.fetch_add(3, std::memory_order_release);
```

Поскольку операции `fetch_add` и `fetch_sub` относятся к операциям чтения-модификации-записи, в них можно использовать любой порядок доступа к памяти, определённый в стандартной библиотеке. Перегруженным комбинированным операциям с присваиванием (`+=` и `-=`) и операциям инкремента и декремента невозможно передать дополнительный аргумент, поэтому в них всегда используется самая сильная семантика доступа, а именно `std::memory_order_seq_cst`.

Общий случай шаблона `std::atomic<>`

В наиболее общем случае класс-шаблон `std::atomic<>` позволяет создавать атомарные версии пользовательских типов данных (англ. user-defined type, UDT). Чтобы пользовательский тип можно было использовать атомарным образом, при его разработке нужно следовать определённым правилам. Тип `std::atomic<udt>` для пользовательского класса `udt` можно образовать, если этот класс обладает тривиальной операцией копирующего присваивания. Это значит, что пользовательский класс `udt` не должен содержать виртуальных функций и не должен иметь виртуальный базовый класс, и его операция копирования должна быть по умолчанию сгенерирована компилятором. Этим же свойством – наличием тривиальной операции копирующего присваивания – должны обладать все базовые классы и нестатические члены-данные класса `udt`. Соблюдение всех этих условий позволяет компилятору реализовать копирующее присваивание просто побайтным копированием представления объекта в памяти (например, функцией `memcpy`). Иными словами, для присваивания из объекта в объект не требуется выполнять какой-либо пользовательский код.

Помимо требования, относящегося к операции присваивания, пользовательский класс `udt` должен допускать *побитовое сравнение на равенство*. Это

означает, что для сравнения двух объектов на равенство должно быть достаточно сравнить их представления в памяти – например, функцией `memcmp`. Только соблюдение этих условий гарантирует возможность атомарных операций сравнения и обмена.

У типа, который получается из стандартного атомарного шаблона подстановкой некоторого пользовательского типа `udt`, т. е. у типа `std::atomic<udt>`, интерфейс ограничен методами `load`, `store`, `exchange`, `compare_exchange_weak` и `compare_exchange_strong`, а также операцией присваивания атомарному объекту значения неатомарного типа `udt` и операцией преобразования атомарного объекта к типу `udt`.

Порядок доступа к памяти

Выше мы рассмотрели атомарные типы и атомарные операции, определённые в стандартной библиотеке. Большая часть операций над атомарными типами позволяет в явном виде задавать порядок доступа к памяти. Пришло время разобраться, в чём состоит смысл и каковы типичные применения различных порядков доступа. Основная идея, лежащая в основе атомарных операций, состоит в синхронизации доступа к данным со стороны множества потоков, и для этого требуется определённым образом ограничить порядок выполнения машинных операций над памятью. Например, если запись данных в память происходит раньше, чем чтение из этого же участка памяти, многопоточная система работает корректно. В противном случае вероятно ошибка. Всего в стандартной библиотеке определено шесть вариантов упорядочивания доступа к памяти, которые можно использовать в операциях над атомарными типами: `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel` и `memory_order_seq_cst`. Если порядок доступа к памяти не задан явно, для всех атомарных операций по умолчанию используется порядок `memory_order_seq_cst`.

Эти шесть вариантов можно разделить на три категории:

- последовательно согласованный порядок: `memory_order_seq_cst`;
- порядки доступа с захватом и освобождением: `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`;
- ослабленный порядок: `memory_order_relaxed`.

Цена, которую приходится платить за эти модели упорядоченного доступа потерей быстродействия, различна и зависит от архитектуры процессора. Наличие различных моделей упорядоченного доступа позволяет высококвалифицированному программисту получать максимум выгоды, применяя, где нужно, более быстрые модели, по сравнению с надёжным, но медленным последовательно согласованным порядком доступа. Однако для того, чтобы выбрать наиболее подходящую модель упорядочения доступа, нужно хорошо понимать, как каждая из этих моделей влияет на поведение программы. Начнём рассмотрение с последовательно согласованного порядка доступа.

Последовательно согласованный порядок доступа

Понятие последовательной согласованности было введено Лэсли Лэмпортом в 1979 г. Последовательная согласованность предоставляет две гарантии касательно хода выполнения программы. В первую очередь это гарантия того, что машинные инструкции, осуществляющие доступ к памяти, выполняются точно в том же порядке, в котором они записаны в исходном коде, – за сохранение порядка инструкций отвечает компилятор. Во-вторых, это гарантии относительно глобального порядка выполнения атомарных операций разными потоками.

Для программиста внесение в поведение программы глобального порядка, которому следуют все операции во всех потоках, словно по единым часам, может послужить важным преимуществом, но оно же может оказаться и недостатком.

Важная черта последовательной согласованности состоит в том, что программа со множеством параллельных потоков работает в точности так, как задумано, но достигается это ценой значительного объёма вспомогательных операций, выполняемых системой. Следующая программа может служить простым примером, знакомящим с особенностями последовательно согласованных вычислений.

```
std::string result;
std::atomic<bool> ready(false);

void thread1(){
    while(!ready.load(std::memory_order_seq_cst));
    result += "согласованность";
}

void thread2(){
    result = "последовательная ";
    ready = true;
}

int main(){
    std::thread t1(thread1);
    std::thread t2(thread2);
    t1.join();
    t2.join();

    std::cout << "Результат : " << result << '\n';
}
```



В этой программе потоки thread1 и thread2 синхронизируются с помощью последовательной согласованности. В силу последовательной согласованности операций выполнение этой программы вполне *детерминировано*, и результат её работы всегда будет выглядеть следующим образом:

Результат: последовательная согласованность

Поток `thread1` выполняет цикл до тех пор, пока атомарная переменная `ready` не получит значение `true`. Как только поток `thread2` присвоит переменной `ready` значение `true`, поток `thread1` продолжит свою работу, поэтому операции над строковой переменной `result` всегда происходят в одном и том же порядке. В общем случае использование последовательно согласованного порядка доступа позволяет каждому потоку видеть операции, выполняющиеся в других потоках, происходящими в одном и том же порядке, поскольку выполнение всех потоков следует единым часам. В данном примере именно наличие единой оси времени позволяет использовать цикл в качестве примитива синхронизации. В следующем разделе рассмотрим семантику захвата и освобождения.

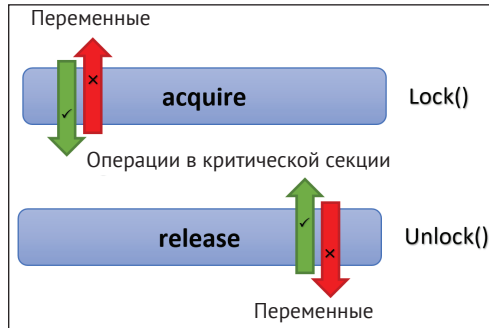
Семантика захвата и освобождения

Пора заняться более тонкими способами упорядочения доступа к памяти, имеющимися в стандартной библиотеке языка C++. Здесь начинается та область, в которой наивные представления о порядке выполнения операций из нескольких потоков перестают работать, так как при такой модели доступа к памяти уже нет глобальных часов, по которым могли бы синхронизироваться атомарные операции разных потоков. Семантика захвата и освобождения поддерживает синхронизацию лишь различных атомарных операций над одной и той же атомарной переменной. Конкретнее, операция чтения из атомарной переменной, выполняемая одним потоком, может быть синхронизована с операцией записи в эту же атомарную переменную, выполняемой другим потоком. Чтобы организовать правильное взаимодействие между потоками через атомарную переменную, программисту нужно увидеть в логике их работы отношение «происходит ранее» и вынудить потоки к его соблюдению. Это делает работу с моделью захвата и освобождения более сложной, но в то же время увлекательной. Использование семантики захвата и освобождения – это шаг на пути к неблокирующему программированию, поскольку программисту нет нужды заботиться о синхронизации целых потоков: всё, о чём нужно задумываться, – это правильный порядок доступа к одной атомарной переменной из нескольких потоков.

Как уже говорилось выше, главная идея, лежащая в основе семантики захвата и освобождения, состоит в синхронизации операции освобождения атомарной переменной одним потоком с операцией её захвата другим потоком вместе с накладыванием условия упорядоченности доступа. Как явствует из названия, операция захвата на краткое время блокирует атомарную переменную и предполагает чтение значения из атомарной переменной – таковы функции `load` и `test_and_set`. Соответственно, операции освобождения снимают блокировку переменной, это характерно для таких операций, как `store` и `clear`.

Если проводить аналогию с блокирующими примитивами синхронизации, захват и освобождение атомарной переменной подобны захвату и освобожде-

нию двоичного семафора. Таким образом, получается некое подобие критической секции, защищающей, однако, лишь одну атомарную переменную – никакую операцию над этой переменной нельзя перенести за пределы критической секции, перед или после неё. Напротив, любые операции над переменной можно безопасно внести в критическую секцию, поскольку в этом случае обращение к переменной переносится из незащищённого участка в защищённый. Это отображено на следующей диаграмме.



Таким образом, критическая секция ограничена двумя односторонне проходимыми барьерами: барьером захвата (операция может переноситься сквозь него вперёд) и барьером освобождения (сквозь него операции могут переноситься назад). Подобный способ рассуждения работает также для запуска потока и вызова метода `join`, захвата и освобождения двоичного семафора и вообще для любых примитивов синхронизации, имеющих в стандартной библиотеке.

Синхронизация в данном случае имеет место на уровне атомарной переменной, а не на уровне целых потоков. Воспользуемся этим, чтобы усовершенствовать реализацию цикла ожидания на основе стандартного атомарного типа `std::atomic_flag`.

```
class spin_lock
{
    std::atomic_flag flg;

public:
    spin_lock() : flg(ATOMIC_FLAG_INIT)
    {}

    void lock()
    {
        // захватить и войти в цикл ожидания
        while (flg.test_and_set(std::memory_order_acquire));
    }

    void unlock()
```

```

{
    // освободить
    flg.clear(std::memory_order_release);
}
};

```

Здесь функция `lock` по сути своей является захватывающей операцией. В отличие от предыдущей реализации, где по умолчанию использовалась последовательно согласованная модель доступа к памяти, в этом коде явно задан флаг захватывающего порядка доступа. Симметричным образом функция `unlock` представляет собой операцию освобождения ресурса, и в её реализации порядок доступа к памяти, использовавшийся по умолчанию, заменён явно указанной семантикой освобождения. Тем самым относительно тяжеловесная синхронизация атомарных операций с последовательной согласованностью заменена более дешёвой и быстрой семантикой захвата и освобождения.

Если этот цикл ожидания предполагается применять для более чем двух потоков, обычной семантики захвата, соответствующей значению `std::memory_order_acquire`, будет недостаточно, так как метод `lock` в этом случае превратится в операцию захвата и освобождения. Соответственно, модель доступа к памяти должна быть заменена на модель `std::memory_order_acq_rel`.

Таким образом, мы выяснили, что последовательно согласованный порядок доступа к памяти обеспечивает глобальную синхронизацию доступа между потоками, тогда как порядок с захватом и освобождением налагает ограничения лишь на порядок операций чтения и записи, выполняемых над одной атомарной переменной из нескольких потоков. Рассмотрим же теперь, в чём состоит ослабленный порядок доступа.

Ослабленный порядок доступа к памяти

Операции над переменными атомарного типа, выполняемые с ослабленным порядком доступа, т. е. с флагом `std::memory_order_relaxed`, вообще не синхронизируются между потоками. В отличие от остальных моделей упорядоченного доступа, доступных в стандартной библиотеке, эта модель не налагает никаких ограничений на порядок выполнения операций различными потоками. Семантика ослабленного порядка гарантирует лишь, что не может быть изменён компилятором (например, в целях оптимизации) порядок операций над одной и той же атомарной переменной, выполняемых в одном потоке, – это требование называют **гарантией неизменного порядка модификации**. Таким образом, ослабленный порядок доступа гарантирует лишь атомарность операций и неизменный порядок модификации. Следовательно, остальные потоки могут видеть модификации переменной в произвольном порядке.

Ослабленный порядок доступа можно с успехом использовать в тех случаях, когда синхронизация и правильный порядок доступа потоков к переменной не требуются, а атомарность операций нужна лишь как вспомогательный механизм для повышения производительности. Типичным примером может быть инкремент счётчика, от которого требуется лишь, чтобы все операции инкре-

мента в конечном счёте были выполнены, – таков, в частности, счётчик ссылок умного указателя `std::shared_ptr`. Декремент этого счётчика, выполняющийся в деструкторе умного указателя, напротив, требует синхронизации путём захвата и освобождения, так как деструктору необходимо точно знать, когда значение счётчика достигнет нуля.

Рассмотрим простой пример, в котором атомарный счётчик с ослабленным порядком доступа используется для подсчёта запущенных потоков.

```
std::atomic<int> count = {0};

void func()
{
    count.fetch_add(1, std::memory_order_relaxed);
}

int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n)
    {
        v.emplace_back(func);
    }

    for (auto& t : v)
    {
        t.join();
    }

    std::cout << "Число потоков : " << count << '\n';
}
```

В этой программе функция `main` запускает десять потоков, в каждом из которых выполняется функция `func`. Эта функция из каждого потока по одному разу наращивает на единицу целочисленный счётчик, используя для этого атомарную операцию `fetch_add`. Эта функция, в отличие от имеющихся в типе `std::atomic<int>` перегруженных комбинированных арифметических операций с присваиванием и перегруженных операций инкремента и декремента, может принимать в качестве аргумента модель упорядоченного доступа к памяти, и в этом примере ей передаётся ослабленная модель `std::memory_order_relaxed`. В самом деле, для корректного выполнения потоков совершенно не важно, в каком порядке будут выполняться операции инкремента, лишь бы они были выполнены все, – а это гарантируется атомарностью операции.

Таким образом, показанная выше программа печатает общее число запущенных потоков, т. е. выводит следующий текст:

Число потоков : 10

Конечно, результат работы этой программы остался бы неизменным, какую бы модель доступа к памяти ни использовать, однако именно ослабленная

модель, свободная от любых ограничений на порядок выполнения операций, обеспечивает наибольшую скорость выполнения.

До сих пор мы изучали различные модели доступа к памяти, предоставляемые ими гарантии и влияние этих гарантий на поведение разных атомарных и неатомарных операций. Посмотрим же теперь, как применить атомарные операции для создания неблокирующей структуры данных.

НЕБЛОКИРУЮЩАЯ ОЧЕРЕДЬ

Как читатель, безусловно, знает, данные в реальных программных системах обычно организованы в определённые структуры, каждая модификация которых требует внесения нескольких согласованных изменений в разные участки памяти. Если нужен доступ к структуре данных сразу из нескольких потоков, производительность может стать серьёзной проблемой. Так, в главе 3 читатель увидел, как реализовать стек, пригодный для использования в многопоточной среде. Его реализация была построена на двоичных семафорах и условных переменных, а захват семафора может потребовать значительного времени процессора. Чтобы пояснить создание неблокирующих структур данных, разберём в качестве примера очень простую очередь, основанную на модели производителей и потребителей (англ. *producer/consumer*), в которой для синхронизации используются не блокирующие примитивы, а исключительно атомарные переменные. Это, безусловно, повысит быстродействие системы. Вместо того чтобы использовать обёртки над стандартными структурами данных, создадим свою реализацию с нуля. Для простоты предположим, что имеется единственный производитель и ровно один потребитель.

```
template<typename T>
class Lock_free_Queue
{
private:
    struct Node
    {
        std::shared_ptr<T> my_data;
        Node* my_next_node;
        Node() : my_next_node(nullptr)
        {}
    };

    std::atomic<Node*> my_head_node;
    std::atomic<Node*> my_tail_node;

    Node* pop_head_node()
    {
        Node* const old_head_node = my_head_node.load();
        if(old_head_node == my_tail_node.load())
        {
            return nullptr;
        }
    }
};
```



```

    my_head_node.store(old_head_node->my_next_node);
    return old_head_node;
}

```

Класс `Lock_free_Queue` содержит объявление вложенного типа с именем `Node` (элемент списка) со своими членами-данными, которые представляют значение, хранящееся в элементе, и указатель на следующий элемент списка. Далее в классе `Lock_free_Queue` объявлены две переменные-члена – атомарные указатели на объекты только что объявленного типа `Node`. В одной из этих переменных хранится указатель на первый элемент списка, а во второй – указатель на последний элемент. Наконец, объявлен и реализован закрытый метод `pop_head_node`, который позволяет извлечь из списка первый элемент, если хотя бы один элемент в списке есть¹, при этом используются лишь атомарные операции. Атомарные операции в этом методе следуют последовательно согласованному порядку доступа к памяти, принятому по умолчанию.

```

public:
    Lock_free_Queue() :
        my_head_node(new Node),
        my_tail_node(my_head_node.load())
    {}

    Lock_free_Queue(const Lock_free_Queue&) = delete;
    Lock_free_Queue& operator=(const Lock_free_Queue&) = delete;

    ~Lock_free_Queue()
    {
        while(Node* const old_head_node = my_head_node.load())
        {
            my_head_node.store(old_head_node->my_next_node);
            delete old_head_node;
        }
    }

```

Конструктор объекта `Lock_free_Queue` создаёт объект типа `Node` и делает его одновременно головой и хвостом списка. Чтобы избежать ситуации, когда несколько объектов владеет одной и той же динамической структурой данных, конструктор копирования и копирующая операция присваивания для этого класса запрещены. Деструктор последовательно удаляет все элементы списка.

```

std::shared_ptr<T> dequeue()
{
    Node* old_head_node = pop_head_node();
    if(!old_head_node)
    {

```



¹ Следует обратить внимание, что в списке всегда присутствует один фиктивный элемент, он нужен для того, чтобы при вставке нового элемента в конец очереди его можно было поместить на заранее заготовленное место. Следовательно, фразу «в списке есть хотя бы один элемент» следует понимать как «хотя бы один, не считая фиктивного». – *Прим. перев.*

```

        return std::shared_ptr<T>();
    }
    std::shared_ptr<T> const result(old_head_node->my_data);
    delete old_head_node;
    return result;
}

void enqueue(T new_value)
{
    std::shared_ptr<T> new_data(std::make_shared<T>(new_value));
    Node* p = new Node;
    Node* const old_tail_node = my_tail_node.load();
    old_tail_node->my_data.swap(new_data);
    old_tail_node->my_next_node = p;
    my_tail_node.store(p);
}
};

```

В последнем фрагменте кода показана реализация двух главных операций над очередью¹, это вталкивание нового элемента в конец очереди (enqueue) и взятие имеющегося элемента из начала очереди (dequeue). Использование атомарных операций load и store обеспечивает отношение «происходит ранее» между методами enqueue и dequeue.

Итоги

В этой главе рассмотрены предоставляемые стандартной библиотекой средства для организации параллельных вычислений на основе задач. Читатель изучил, как использовать фьючерсы и обещания, класс `std::packaged_task` и функцию `std::async`. Также в главе рассказано о новой, ориентированной на параллельные вычисления модели памяти, закреплённой в современном стандарте языка C++. Вслед за этим были рассмотрены атомарные типы данных и операции над ними. Один из самых важных аспектов, связанных с атомарными операциями, – это разнообразие моделей упорядоченного доступа к памяти со своими специфическими гарантиями. Вместе взятые, две последние главы позволят нам в дальнейшем понять аспекты реактивной модели программирования, связанные с параллельным выполнением потоков.

В следующей главе мы от рассмотрения самого по себе языка C++ и тонкостей параллельного программирования перейдём к изучению интерфейсов, лежащих в основе реактивной модели программирования. Приступим же к изучению наблюдателей!

¹ Неблокирующее программирование и, в частности, разработка неблокирующих структур данных – обширная и усложнённая область, представленное в этой главе изложение представляется слишком кратким и фрагментарным, а использованный автором пример – недостаточно информативным. Читателю стоит обратиться к специализированным источникам. – *Прим. перев.*

Глава 5

.....



Знакомство с наблюдаемыми источниками

В последних трёх главах были изучены появившиеся в языке C++ нововведения: поддержка потоков, примитивы синхронизации, средства неблокирующего программирования и др. Рассмотрение этих вопросов можно считать необходимым подготовительным шагом перед серьёзным изучением реактивной модели программирования. Следование реактивной модели требует хорошего владения функциональной парадигмой, средствами параллельного программирования, планировщиками, функциональными объектами, шаблонами проектирования и методами обработки потоков событий – и это лишь краткий список. Выше мы разобрали или хотя бы коснулись таких тем, как функциональное программирование и функциональные объекты; кроме того, в предыдущей главе затронуты некоторые аспекты планирования задач. В этой главе будет описан чудесный мир шаблонов проектирования, что позволит понять суть реактивного программирования вообще и в особенности идею наблюдателей. В следующей главе речь пойдёт о работе с потоками событий, это позволит затем перейти к изучению библиотеки RxCpp. Популярность идеи шаблонов проектирования достигла критической массы с выходом в 1994 г. книги «Приёмы объектно-ориентированного проектирования. Паттерны проектирования» коллектива авторов, известных как «Банда четырёх» (англ. Gang of Four, сокращённо GoF). В ней представлен каталог из 23 шаблонов проектирования, разделённых на три группы: порождающие шаблоны, структурные шаблоны и шаблоны поведения. В каталоге Банды четырёх шаблон «Наблюдатель» отнесён к группе шаблонов поведения. Главная мысль, которую данная глава должна донести до читателя, состоит в том, что реактивную модель программирования можно понять через призму прочно вошедших в обиход классических шаблонов проектирования. В этой главе будут разобраны следующие вопросы:

- классический шаблон «Наблюдатель»;
- ограничения, присущие шаблону «Наблюдатель»;
- обобщённый взгляд на шаблоны проектирования и, в частности, на наблюдателей;
- моделирование иерархий, свойственных реальному миру, с помощью шаблона «Композит»;
- придание композитам гибкого поведения с использованием шаблона «Посетитель»;
- «уплощение» многоуровневых композитов в одноуровневые структуры с помощью шаблона «Итератор»;
- как «вывернуть наизнанку» итератор, преобразовав его в пару наблюдаемый источник – наблюдатель.

Шаблон «Наблюдатель»

Описанный в книге «Банды четырёх» шаблон «Наблюдатель» известен также под названием «Издатель-подписчик». Его основная идея проста. Объект `EventSource` (источник событий) связан отношением типа «один ко многим» с объектами `EventSink` (приёмник событий), которые постоянно ожидают оповещений о событиях. Объект `EventSource` должен обладать механизмом, позволяющим любому объекту-приёмнику подписаться на получение оповещений (возможно, избранных типов). Один объект-источник может испускать множество событий, в общем случае различных типов. Один объект `EventSource` может рассылать оповещения тысячам подписчиков, или приёмников, всякий раз, когда происходит значимое изменение его состояния или вообще нечто важное в его сфере ответственности. Для этого объект `EventSource` проходит по списку своих подписчиков и посылает оповещение каждому из них. Книга «Банды четырёх» была написана в эпоху, когда повсеместно преобладало последовательное, однопоточное программирование. Вопросы параллельной обработки главным образом принято было связывать со спецификой отдельно взятых платформ или, в крайнем случае, с интерфейсом управления потоками в стандарте POSIX. Напишем простую программу на языке C++, чтобы продемонстрировать суть шаблона «Наблюдатель» в целом. Цель этого упражнения состоит в том, чтобы быстро понять данный шаблон, поэтому вопросы надёжности и эффективности отодвинуты на второй план. Следующий код самодостаточен и вполне очевиден.

```
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

// упреждающее объявление класса-приёмника событий
template<class T>
class EventSourceValueObserver;
```



```
// упрощённая реализация источника событий
template<class T>
class EventSourceValueSubject {
    vector<EventSourceValueObserver<T>*> sinks;
    T State; // это должен быть тип значения

public:
    EventSourceValueSubject() { State = 0; }
    ~EventSourceValueSubject() {
        for (auto *n : sinks) { delete n; }
        sinks.clear();
    }

    bool Subscribe(EventSourceValueObserver<T> *sink)
    { sinks.push_back(sink); }

    void NotifyAll() {
        for (auto sink : sinks) { sink->Update(State); }
    }

    T GetState() { return State; }
    void SetState(T pstate) { State = pstate; NotifyAll(); }
};
```

В представленном выше фрагменте кода реализован простейший источник событий, обладающий состоянием какого-то достаточно простого (например, целочисленного) типа. Современный стандарт языка C++ содержит богатый набор средств, позволяющий во время компиляции определить, правильный ли тип-параметр пытается передать пользователь. Однако, поскольку этот пример призван лишь пояснить суть наблюдателей, не станем загромождать код, выписывая ограничения на тип-параметр. В будущем стандарте C++ 20 должны появиться так называемые концепты (подобные ограничениям или классам типов в некоторых других языках), которые позволят описывать требования к типам-параметрам в ясной и компактной форме, не прибегая к нынешнему тяжеловесному синтаксису. В реальных программных системах источник событий может хранить в себе сколь угодно сложное состояние, состоящее из множества переменных или потоков значений. Подписчики должны оповещаться о любом изменении этого состояния. В нашем примере, когда пользователь данного класса меняет его состояние, вызвав его метод `SetState`, автоматически вызывается метод `NotifyAll`. Последний, в свою очередь, проходит по списку приёмников и для каждого из них вызывает метод `Update`. Тогда приёмники могут выполнить свои специфические обработчики. Мы не станем реализовывать в данном примере другие методы (в частности, метод для отписки от оповещений), чтобы сфокусировать внимание на главном.

```
// класс приёмника событий (наблюдателя), общий случай
template <class T>
class EventSourceValueObserver{
    T OldState;
```

```
public:
    EventSourceValueObserver() { OldState = 0; }

    virtual ~EventSourceValueObserver() {}

    virtual void Update(T State) {
        cout << "Старое состояние " << OldState << endl;
        OldState = State;
        cout << " Новое состояние " << State << endl;
    }
};
```

В классе `EventSourceValueObserver` реализован метод `Update`, который позволяет каждому подписчику по-своему реагировать на изменения, о которых сообщает источник. В этом примере приёмник просто печатает на консоль старое и новое состояния. В реальных программных системах приёмник в ответ на оповещение может, например, перерисовать какие-то элементы графического интерфейса или же передать обновлённое состояние дальше по цепочке объектов-обработчиков, разослав собственное оповещение. Создадим ещё один класс приёмника события, унаследованный от класса `EventSourceValueObserver`.

```
// специализированный наблюдатель
class AnotherObserver : public EventSourceValueObserver<double>
{
public:
    AnotherObserver(): EventSourceValueObserver() {}
    virtual ~AnotherObserver() {}
    virtual void Update(double State) {
        cout << " Specialized Observer" << endl;
    }
};
```

Эта специализированная версия наблюдателя призвана продемонстрировать, что подписчики, совместно работающие в одной системе, могут относиться к разным классам (которые, конечно, должны все иметь общего предка – в нашем примере это класс `EventSourceValueObserver<T>`). Показанный здесь специализированный наблюдатель также не делает ничего сложного.

```
int main() {
    unique_ptr<EventSourceValueSubject<double>>
        evsrc(new EventSourceValueSubject<double>());
    // создать наблюдателей и подписать их на получение оповещений
    evsrc->Subscribe(new AnotherObserver());
    evsrc->Subscribe(new EventSourceValueObserver<double>());

    // изменить состояние источника событий, это должно
    // привести к вызову метода Update обоих наблюдателей
    evsrc->SetState(100);

    return 0;
}
```

В этом последнем фрагменте кода создаётся источник событий, затем к нему подключаются два подписчика. Теперь можно менять состояние источника и быть уверенными, что оповещение об этом придёт всем подписчикам. В этом и состоит суть шаблона «Наблюдатель». В обычных объектно-ориентированных системах, не основанных на данном шаблоне, работа с объектами происходит по следующей схеме:

- создать объект;
- вызвать метод этого объекта, чтобы вычислить некоторое значение или изменить его состояние;
- сделать что-то полезное с полученным значением или с новым состоянием объекта.

В случае же шаблона «Наблюдатель», однако, схема получается иной:

- создать объект-источник событий;
- подписать наблюдателей на оповещения этого источника;
- изменить состояние объекта-источника;
- в объектах-приёмниках обработать значение или состояние, полученное от источника.

Вынесение метода обработки обновлённого состояния позволяет лучше разграничить обязанности объектов и добиться более чёткого разделения системы на независимые модули. Шаблон «Наблюдатель» предоставляет хороший инструмент для создания программ, управляемых событиями. Потребитель событий не сам опрашивает источник о новых событиях, а, напротив, получает их автоматически. Большинство современных библиотек для создания графических интерфейсов основано на этом принципе.

ОГРАНИЧЕННОСТЬ КЛАССИЧЕСКОГО ШАБЛОНА «НАБЛЮДАТЕЛЬ»

Книга «Банды четырёх» была написана в эпоху, когда повсеместно доминировало последовательное, однопоточное программирование. Представленная в книге архитектура и реализация шаблона «Наблюдатель» обладают рядом недостатков, если оценивать, исходя из современных представлений о программировании. Вот лишь некоторые из них:

- слишком тесная взаимозависимость отправителей и наблюдателей;
- временем жизни источника событий управляют наблюдатели¹;
- наблюдатели (приёмники событий) могут заблокировать работу источника;
- реализация небезопасна при использовании в многопоточной среде;
- за фильтрацию событий отвечают приёмники. Было бы гораздо лучше, если бы данные фильтровал тот, кто ими владеет, – в данном случае

¹ С этим утверждением автора трудно согласиться: даже в примере программного кода и предыдущего раздела зависимость противоположна – источник событий управляет временем жизни подписчиков. Эту зависимость легко разорвать, если убрать деструктор из класса источника, а вместо обычных указателей использовать умные указатели `std::shared_ptr`. – *Прим. перев.*

фильтровать следовало бы на стороне источника перед рассылкой оповещений;

- большую часть времени наблюдатели не выполняют никакой полезной работы, потребляя ресурсы;
- в идеальном случае источнику событий вообще не следует знать своих подписчиков. Вместо этого пусть он оповещает об изменении некую среду, которая сама должна отвечать за рассылку события всем нужным адресатам. Более высокая степень изоляции отправителей и получателей в этом случае позволила бы применить такие полезные техники, как агрегирование событий, преобразование событий, фильтрация событий и приведение данных о событии к тому или иному каноническому виду – и этот список отнюдь не полон.

С проникновением в практику программирования на языке C++ средств, заимствованных из функциональной парадигмы, таких как неизменяемые данные, композиция функций, трансформация данных чистыми функциями, неблокирующее параллельное программирование и др., появилась возможность преодолеть ограничения классического шаблона «Наблюдатель». Решение, выработанное современной индустрией программного обеспечения, основано на понятии наблюдаемого источника как дальнейшего развития идеи источника событий.

Прилежный читатель уже при изучении классического шаблона «Наблюдатель» мог увидеть потенциал для применения асинхронной модели программирования. В самом деле, источник событий может вызывать метод Update своих подписчиков асинхронно, вместо того чтобы осуществлять вызовы строго последовательно. Используя, как при стрельбе самонаводящимися ракетами, принцип «запусти и забудь», можно устранить жёсткую связь источника событий с приёмниками. Фактическое обращение к методам подписчиков может происходить в фоновом потоке, асинхронной задаче, пакетной задаче или посредством любого другого механизма, подходящего для конкретной ситуации. Одно из преимуществ асинхронной рассылки оповещений состоит в том, что если какой-то из получателей окажется заблокирован (например, войдёт в бесконечный цикл или аварийно прекратит работу), остальные получатели всё равно получают свои оповещения. Асинхронный способ рассылки оповещений работает по следующей схеме.

Программист определяет методы для обработки данных, исключений и ситуации окончания потока данных.

- Интерфейс наблюдателя, или приёмника событий, должен содержать методы `OnData`, `OnError` и `OnCompleted`.
- Каждый приёмник событий должен воплощать этот интерфейс.
- Каждый наблюдаемый источник должен обладать методами для подписки и отписки.
- Приёмник событий должен подписаться на получение событий от определённого наблюдаемого источника, обратившись к его методу подписки.
- Всякий раз, когда происходит некоторое событие, наблюдатели (приёмники) получают оповещения от наблюдаемого источника.

Часть этой логики уже была описана в главе 1 при первом знакомстве с реактивным программированием. Там не упоминалась лишь возможность асинхронной обработки событий. В данной главе мы ещё раз рассмотрим этот комплекс понятий. Опыт, полученный автором в ходе выступлений на технические темы и бесед с разработчиками, свидетельствует, что нет смысла начинать изучение сразу с модели наблюдаемых источников. Многие разработчики с трудом понимают архитектуру, основанную на наблюдаемых источниках, поскольку им не вполне ясно, какую проблему эта архитектура решает. Поэтому мы и начали с классического шаблона «Наблюдатель», подготовив тем самым почву для изучения наблюдаемых источников и, в частности, наблюдаемых потоков.

ОБОБЩЁННЫЙ ВЗГЛЯД НА ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Идея шаблонов проектирования начала завоёвывать всеобщее признание в то самое время, когда программисты всего мира силились осмыслить сложный комплекс проблем, связанных с разработкой объектно-ориентированных программ. Книга «Банды четырёх» и представленный в ней каталог шаблонов предоставили сообществу разработчиков стройную систему подходов для разработки больших систем. В то же время вопросы параллельной и многопоточной обработки отнюдь не находились в центре внимания авторов знаменитого каталога (по крайней мере, в их работе интерес к данной теме не отражен).

Выше мы убедились, что обработка событий на основе классического шаблона «Наблюдатель» не свободна от ряда недостатков и ограничений, что в ряде случаев может привести к серьёзным трудностям. Каков выход из этого положения? Нужно всего лишь окинуть всю задачу обработки событий свежим взглядом, отступив на шаг назад. При этом мы вторгнемся на время в область философии, чтобы под новым углом посмотреть на задачу, решить которую призвана реактивная модель программирования, в особенности программирование на основе наблюдаемых потоков. Всё это поможет нам гладко перейти от классических шаблонов «Банды четырёх» к миру реактивного программирования и лежащим в его основе структурам функциональной парадигмы.

Материал этого раздела несколько абстрактен и приведён здесь с целью пояснить основополагающие принципы, на которых построен материал всей главы. Наш путь к объяснению наблюдаемых источников начнется с шаблонов «Композит» и «Посетитель», также изложенных в работе «Банды четырёх», и шаг за шагом будет приближать к основному предмету рассмотрения. Этот подход к разъяснению сложных понятий позаимствован из адвайта-веданты¹,

¹ Помимо краткого изложения, представленного здесь автором, следует упомянуть принцип тождества, занимающий в философии адвайты центральное место: Брахман (упрощённо говоря, абсолютная мировая душа) тождествен индивидуальной душе, а следствие тождественно причине. Множественность явлений материального мира в конечном счете есть иллюзия, так как каждое из них уже изначально заключено в Брахмане. Поэтому изучение сложного предмета можно начинать, восходя по лестнице простых частных случаев, всё равно они суть одно. – *Прим. перев.*

одного из мистико-философских учений Индии. Впрочем, изложение будет вестись в терминах западной философии. Если этот материал покажется читателю слишком абстрактным, его можно пропустить.

Гуру Натараджа (1895–1973) – индийский философ, внесший значительный вклад в развитие и популяризацию адвайта-веданты, философской школы, в основе которой лежит представление о недвойственности высшей силы, управляющей всех нас. Согласно его изложению адвайты, всё, что мы видим вокруг, будь то люди, животные или растения, суть проявления абсолюта, на санскрите называемого Брахманом. Единственное положительное утверждение, которое можно сделать о Брахмане, сводится к триаде «бытие – сознание (также суть) – блаженство» (санскр. сат, чит, ананд). Для философии веданты вообще характерен подход к определению сущности через описание её противоположности и метод доказательства от противного. В своей книге, озаглавленной «Всеобъемлющая философия» (англ. Unitive Philosophy), Натараджа связывает понятия сат, чит и ананд с тремя главными разделами, сформировавшимися в западной философии, – соответственно онтологией (учением о бытии), гносеологией (иначе эпистемологией – учением о познании) и аксиологией (учением о ценностях). В следующей таблице показано отображение триады сат-чит-ананд на родственные им философские категории.

Сат	Чит	Ананд
Бытие	Суть	Блаженство
Онтология	Гносеология	Аксиология
Кто я?	Что я могу знать?	Что мне делать?
Структура	Поведение	Предназначение

В веданте как семействе философских школ и в особенности в адвайте как одной из них всё мироздание рассматривается как единство бытия, сознания и блаженства. В дальнейшем мы будем опираться на эту таблицу, чтобы отобразить проблемы разработки программного обеспечения на категории структуры, поведения и предназначения. Любую систему в мире можно рассматривать с этих трёх точек зрения: структурной, поведенческой и целевой. Каноническую структуру объектно-ориентированных систем составляют иерархии¹. Разрабатывая программу, мы моделируем интересующий фрагмент мира в виде иерархий и применяем некоторые канонические методы для их обработки. Каталог «Банды четырёх» содержит шаблон «Композит», отнесённый к категории структурных шаблонов и предназначенный для выстраивания иерархии объектов, и шаблон «Наблюдатель», отнесённый к поведенческим и предназначенный для обработки таких иерархий.

¹ Авторское соотнесение категории ананд с аксиологией отнюдь не бесспорно. Гносеология также могла бы занять место в этой таблице. – Прим. перев.

² В классических изложениях теории ООП говорят об иерархии классов и иерархии объектов. По сути, автор предлагает дополнить канон ООП иерархией целей. – Прим. перев.

ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ МОДЕЛЬ ПРОГРАММИРОВАНИЯ И ИЕРАРХИИ

Этот раздел носит преимущественно теоретический характер. Читателям, не искушённым в шаблонах «Банды четырёх», он может показаться излишне сложным. В таком случае лучше пропустить этот раздел и перейти к разбору примера работающей программы. Когда пример вполне изучен, можно вернуться к изучению настоящего раздела.

Объектно-ориентированный подход хорош для моделирования разнообразных иерархий. Фактически иерархии можно считать канонической моделью данных в объектно-ориентированных системах. Среди всех шаблонов «Банды четырёх» для моделирования иерархий объектов лучше всего подходит шаблон «Композит». Он классифицируется как структурный шаблон. Часто рядом с шаблоном «Композит» используется и шаблон «Посетитель». Последний хорошо подходит для обработки композитных объектов, т. е. для придания иерархическим объектам некоторого поведения. Тем самым во многих реальных программных системах шаблоны «Композит» и «Посетитель» образуют неразрывную пару. Конечно, один и тот же композитный объект можно обрабатывать различными посетителями. Например, при разработке компилятора абстрактное синтаксическое дерево (англ. *abstract syntax tree*, AST) удобно представить объектом-композитом, а проверка типов, оптимизация и генерация кода, статический анализ и другие операции над ним могут быть реализованы посредством различных посетителей.

Одно из неудобств, возникающих при применении посетителя совместно с композитом, состоит в том, что реализация посетителя должна быть осведомлена о структуре композита, чтобы правильно выполнить обход всех входящих в него подобъектов. Более того, это способно сильно привести к лавинообразному разрастанию кода в случаях, когда посетителю нужно обработать определённым образом отобранное подмножество данных из иерархического композита. Ведь для каждого критерия фильтрации объектов мог бы понадобиться отдельный класс посетителя¹. Каталог «Банды четырёх» содержит ещё один шаблон, также относящийся к группе поведенческих, а именно «Итератор» – любой программист на языке C++ наверняка знаком с этим понятием. Шаблон «Итератор» удобен для обработки данных, организованных в контейнеры, внутренняя структура которых несущественна. Иерархическую струк-

¹ Данная проблема представляется несколько преувеличенной. Выбор посетителем объектов по неограниченному разнообразию критериев можно выразить небольшим объёмом кода, прибегнув к таким шаблонам «Банды четырёх», как «Декоратор» или «Стратегия». В первом случае на объект-посетитель, отвечающий за обработку данных, нужно наложить декоратор, который игнорирует посещаемый объект, если он не удовлетворяет нужному критерию. Во втором случае в интерфейсе объекта-посетителя должна быть изначально предусмотрена возможность параметризации некоторым предикатом (стратегией отбора). – *Прим. перев.*

туру данных произвольной сложности можно разглядеть – концептуально представить в виде линейной последовательности элементов, то есть привести к виду, удобному для обработки с помощью итератора. Примером может служить итератор, осуществляющий обход дерева в ширину или в глубину. Для прикладного программиста дерево через призму такого итератора выглядит линейной последовательностью. Приведение иерархии к виду, удобному для обхода итератором, концептуально состоит в её уплощении. Следует, однако, отметить, что итераторы в силу их втягивающей семантики несколько ограничены, поэтому нашим следующим шагом будет подойти к задаче с противоположной стороны и принять семантику вталкивания, воспользовавшись идеей наблюдаемого источника и наблюдателя как усовершенствованной версией шаблона «Наблюдатель». Данный раздел может показаться читателю излишне абстрактным, в этом случае можно проработать оставшуюся часть главы, вернуться и тогда уже понять, что здесь имеется в виду. Пока что можно подытожить наше рассмотрение следующим образом:

- иерархические структуры данных можно моделировать на основе шаблона «Композит»;
- для обработки композитных объектов можно использовать шаблон «Посетитель»;
- композиты произвольной сложности можно приводить к линейному, плоскому виду, применяя для их обхода итераторы;
- итераторам присуща семантика втягивания, нужно посмотреть на них с обратной стороны, чтобы получить семантику вталкивания;
- тем самым получается подход к реализации систем, основанный на наблюдаемых источниках и наблюдателях;
- наблюдаемые источники и итераторы образуют зеркальную пару: вталкивание данных одной стороной соответствует их втягиванию другой стороной.

Займёмся реализацией всего перечисленного, чтобы основательно понять наблюдаемые источники.

ОБРАБОТКА ВЫРАЖЕНИЙ С ПОМОЩЬЮ ШАБЛОНОВ «КОМПОЗИТ» И «ПОСЕТИТЕЛЬ»

Чтобы проиллюстрировать путь от каталога шаблонов «Банды четырёх» к наблюдаемым источникам, нам в качестве сквозного примера понадобится калькулятор на четыре действия. Поскольку древовидная структура выражения иерархична по своей природе, для её моделирования хорошо подходит шаблон «Композит». Ради компактности кода мы не станем здесь заниматься созданием синтаксического анализатора.

```
#include <iostream>
#include <memory>
#include <list>
```



```
#include <stack>
#include <functional>
#include <thread>
#include <future>
#include <random>
#include "FuncCompose.h" // в репозитории
using namespace std;
// Перечень операций, поддерживаемых вычислителем
enum class OPERATOR {
    ILLEGAL, PLUS, MINUS, MUL, DIV, UNARY_PLUS, UNARY_MINUS
};
```



Здесь объявлен тип-перечисление, который представляет четыре бинарные арифметические операции (сложение, вычитание, умножение и деление) и две унарные (плюс и минус), а также состояние ошибки вычислений. Помимо ряда стандартных заголовочных файлов, подключён наш собственный файл `FuncCompose.h`, который можно найти в системе GitHub, в репозитории с материалами к данной книге. В этом файле реализованы функция `Compose` и операция композиции функций (`()`). Тем самым мы получаем возможность строить композиции функций-преобразователей в стиле конвейеров командной оболочки Unix.

```
// Хранит стандартное число с плавающей точкой
class Number;
```

```
// Узлы синтаксического дерева:
class BinaryExpr; // - для бинарной операции
class UnaryExpr;  // - для унарной операции
```

```
class IExprVisitor; // Интерфейс посетителя
```

```
// Базовый для всех классов узлов синтаксического дерева
class Expr {
public:
    // двойная диспетчеризация: делегировать посетителю
    // обработку конкретного типа узла
    virtual double accept(IExprVisitor& expr_vis) = 0;
    virtual ~Expr() {}
};
```



```
// Интерфейс посетителя с методами для обработки разных типов узлов
struct IExprVisitor{
    virtual double Visit(Number& num) = 0;
    virtual double Visit(BinaryExpr& bin) = 0;
    virtual double Visit(UnaryExpr& un) = 0;
};
```

Класс `Expr` представляет собой базовый класс, общий для всех типов узлов, из которых состоят абстрактные синтаксические деревья выражений. Для нашей цели, демонстрации совместной работы посетителей и композитов «Банды четырёх», вполне хватит трёх типов узлов: констант, бинарных и унарных опе-

раций. Метод ассепт класса Expr получает на вход ссылку на объект-посетитель и обеспечивает вызов в посетителе метода-обработчика, соответствующего фактическому типу данного узла. Для этого текст реализации данного метода во всех порождённых классах должен выглядеть одинаково. Чтобы лучше понять принцип действия этой системы в целом, читателю рекомендуется отыскать в интернете материалы о *двойной диспетчеризации* и о шаблоне «Посетитель».

Интерфейс посетителя IExprVisitor содержит методы для обработки всех конкретных типов узлов, которые могут встретиться в дереве выражения. В нашем примере это методы для обработки числовых констант, бинарных и унарных операций. Рассмотрим реализацию классов, представляющих узлы дерева. Начнём с класса Number.

```
// класс-обёртка над числом с плавающей точкой
class Number : public Expr {
    double NUM;
public:
    double getNUM() { return NUM; }
    void setNUM(double num) { NUM = num; }
    Number(double n) { this->NUM = n; }
    ~Number() {}
    double accept(IExprVisitor& expr_vis) {
        return expr_vis.Visit(*this);
    }
};
```

Этот класс представляет собой надстройку над встроенным типом числа с плавающей точкой двойной точности. Его код вполне очевиден, и всё, на что стоит обратить внимание, – это реализация метода ассепт. Этот метод принимает в качестве аргумента ссылку на объект-посетитель (IExprVisitor&). Всё предназначение этого метода состоит в том, чтобы переадресовать вызов посетителю – именно тому его методу-обработчику, который отвечает за конкретный тип узла. В данном случае это будет метод Visit(Number&). Рассмотрим теперь класс, представляющий бинарные операции:

```
// выражение с бинарной операцией
class BinaryExpr : public Expr {
    Expr* left; Expr* right;
    OPERATOR OP;
public:
    BinaryExpr(Expr* l, Expr* r , OPERATOR op) {
        left = l;
        right = r;
        OP = op;
    }
    OPERATOR getOP() { return OP; }
    Expr& getLeft() { return *left; }
    Expr& getRight() { return *right; }
    ~BinaryExpr() {
        delete left;
        delete right;
    }
};
```

```

    }
    double accept(IEExprVisitor& expr_vis) {
        return expr_vis.Visit(*this);
    }
};

```

Класс `BinaryExpr` моделирует выражение, составленное с помощью бинарной операции¹ из левого и правого подвыражений-операндов. В нашем примере бинарные операции – это сложение, вычитание, умножение и деление. Операнды могут быть представлены любыми классами из нашей иерархии: каждый из операндов может, независимо от другого, быть объектом класса `Number`, `BinaryExpr` или `UnaryExpr`. Таким образом, древовидная структура выражения может достигать какой угодно высоты, а терминальными вершинами в нашем примере всегда будут объекты класса `Number`. Перейдём к реализации класса, моделирующего унарную операцию.

```

// выражение с унарной операцией
class UnaryExpr : public Expr {
    Expr * right;
    OPERATOR op;
public:
    UnaryExpr(Expr *operand , OPERATOR op) {
        right = operand;
        this-> op = op;
    }
    Expr& getRight( ) { return *right; }
    OPERATOR getOP() { return op; }
    virtual ~UnaryExpr() { delete right; right = 0; }
    double accept(IEExprVisitor& expr_vis) {
        return expr_vis.Visit(*this);
    }
};

```



Класс `UnaryExpr` моделирует выражение, полученное применением унарной операции к подвыражению-операнду. В нашем примере присутствуют операции «унарный плюс» и «унарный минус». Подвыражение представлено объектом любого класса, порождённого от абстрактного базового класса `Expr`: это может быть объект класса `Number` (число), `BinaryExpr` (бинарное выражение) или `UnaryExpr`.

Теперь, когда реализации всех типов узлов дерева построены, обратимся к реализации интерфейса посетителя. Ниже представлены два посетителя, отвечающие за две различные операции над выражениями.

¹ Предложенное автором решение не идеально. В частности, в объектах классов `BinaryExpr` и `UnaryExpr` код операции представлен значениями одного и того же типа-перечисления `OPERATOR`. Но это значит, что архитектура не препятствует появлению бессмысленных объектов – скажем, объекта класса `UnaryExpr` с кодом операции деления `DIV`. Использование «сырых» указателей также вряд ли стоит считать образцом для подражания. – *Прим. перев.*

```
// посетитель, вычисляющий значение выражения
class TreeEvaluatorVisitor : public IExprVisitor {
public:
    double Visit(Number& num) { return num.getNUM(); }

    double Visit(BinaryExpr& bin) {
        OPERATOR temp = bin.getOP();
        double lval = bin.getLeft().accept(*this);
        double rval = bin.getRight().accept(*this);
        return (temp == OPERATOR::PLUS)
            ? lval + rval
            : (temp == OPERATOR::MUL)
            ? lval*rval
            : (temp == OPERATOR::DIV)
            ? lval/rval
            : lval-rval;
    }

    double Visit(UnaryExpr& un) {
        OPERATOR temp = un.getOP();
        double rval = un.getRight().accept(*this);
        return (temp == OPERATOR::UNARY_PLUS)
            ? +rval
            : -rval;
    }
};
```

Представленный выше класс посетителя осуществляет обход абстрактного синтаксического дерева выражения в глубину и рекурсивно, от листовых узлов к корневому, вычисляет значение выражения. Создадим теперь такой обработчик выражений, который бы переводил выражение в обратную польскую форму записи¹.

```
// посетитель для перевода выражения в обратную польскую запись
class ReversePolishEvaluator : public IExprVisitor {
public:
    double Visit(Number& num) {
        cout << num.getNUM() << " " << endl;
        return 42;
    }

    double Visit(BinaryExpr& bin) {
        bin.getLeft().accept(*this);
        bin.getRight().accept(*this);
        OPERATOR temp = bin.getOP();
```

¹ К сожалению, автор предлагает читателю в качестве примера код, страдающий совершенно недопустимыми изъянами. Основной результат работы данного посетителя выражен побочным эффектом (выводом текста в поток `std::cout`), тогда как возвращаемое посетителем значение фиктивно. Приведение этого кода к мало-мальски приемлемому виду оставляем читателю в качестве самостоятельного упражнения. — *Прим. перев.*

```

auto const op = (temp==OPERATOR::PLUS)
? " + "
: (temp==OPERATOR::MUL)
? " * "
: (temp == OPERATOR::DIV) ?
" / "
: " - ";
cout << op;
return 42;
}

double Visit(UnaryExpr& un) {
    OPERATOR temp = un.getOP();
    un.getRight().accept(*this);
    cout << (temp == OPERATOR::UNARY_PLUS) ? " (+) " : " (-) ";
    return 42;
}
};

```

Обратную польскую форму записи выражений часто ещё называют постфиксной, так как знак операции в ней пишется после операндов. Данная форма записи позволяет вычислять значение выражения с помощью стека. Постфиксная запись выражений составляет основу архитектуры стековых виртуальных машин, к которым относятся виртуальная машина Java и среда выполнения .Net CLR.

Теперь осталось написать функцию `main`, чтобы продемонстрировать работу системы в целом.

```

int main() {
    unique_ptr<Expr> a(
        new BinaryExpr(
            new Number(10.0),
            new Number(20.0),
            OPERATOR::PLUS));

    unique_ptr<IExprVisitor> eval( new TreeEvaluatorVisitor());
    double result = a->accept(*eval);
    cout << "Результат вычисления => " << result << endl;

    unique_ptr<IExprVisitor> exp(new ReversePolishEvaluator());
    cout << "Выражение в постфиксной записи:" << endl;
    a->accept(*exp);
    return 0;
}

```

В этом фрагменте кода создаётся композитный объект (экземпляр класса `BinaryExpr`), затем создаются два посетителя (экземпляры классов `TreeEvaluatorVisitor` и `ReversePolishEvaluator`). Вызов метода `accept` запускает обработку выражения каждым из этих посетителей. В результате выполнения программы пользователь увидит значение выражения и его представление в обратной польской форме.

Таким образом, в этом разделе читатель узнал, как создавать композитные объекты¹ и обрабатывать их с помощью посетителей, реализующих единый интерфейс. Композиты совместно с посетителями могут иметь множество различных применений – например, обход каталога в файловой системе, обработка данных в формате XML, обработка текстовых документов и т. д. Широко распространено мнение, что тот, кто понял шаблоны «Композит» и «Посетитель», тем самым понял и весь каталог шаблонов «Банды четырёх».

Выше было показано, что шаблоны «Композит» и «Посетитель» образуют пару и отвечают, соответственно, за структурный и поведенческий аспекты системы, а также, в некоторой степени, за её целевое предназначение. Следует учесть, что реализация шаблона «Посетитель» основывается на предположении, что известно внутреннее устройство композитного объекта. Это нежелательно, так как противоречит основополагающему для ООП принципу абстрагирования от деталей реализации. Чтобы справиться с этим затруднением, создателю многоуровневого композита стоит продумать механизм для «разглаживания» иерархии объектов в нечто, концептуально выглядящее линейным списком, – в большинстве случаев это оказывается возможным. Это позволит пользователю таких композитов работать с объектами-компонентами через единый, основанный на итераторах интерфейс. Программный интерфейс, основанный на итераторах, хорош также для программирования в функциональном стиле. Рассмотрим этот подход подробнее.

РАЗГЛАЖИВАНИЕ МНОГУРОВНЕВЫХ КОМПОЗИТОВ ДЛЯ ИТЕРАТИВНОГО ДОСТУПА

Как уже говорилось выше, для реализации шаблона «Посетитель» программисту нужно знать структуру объекта-композита. Это может привести к так называемой *протекающей абстракции* – ситуации, когда сквозь абстракцию «просачиваются» подробности, для сокрытия которых данная абстракция как раз и предназначалась. В каталоге «Банды четырёх» есть шаблон, который поможет нам обходить дерево объектов, забыв о том, что это дерево. Проницательный читатель уже наверняка догадался, что речь идёт о шаблоне «Итератор». Применение итератора фактически означает разглаживание иерархического композита в линейную последовательность, поток компонентов. Продолжая пример из предыдущего раздела, разберём алгоритм, который разглаживает древовидное представление арифметического выражения. Прежде

¹ Чтобы полнее раскрыть тему композитов, стоит добавить, что классический шаблон «Композит» предполагает соединение в одном объекте-композите произвольного числа объектов-компонентов, а возможность создавать многоуровневые композиты, как правило, не используется. В данном разделе, напротив, композиты (унарные и бинарные выражения) содержат всегда фиксированное число компонентов (один и два соответственно), зато на первый план выходит возможность строить древовидные композиты произвольной высоты. – *Прим. перев.*

всего нужно определить структуру данных, позволяющую хранить содержимое абстрактного синтаксического дерева в виде линейной последовательности. Каждый объект в этом списке должен представлять либо операцию, либо значение-операнд. Объявим следующий тип¹:

```
// Вид узла: операция или значение
enum class ExprKind{
    ILLEGAL_EXP, OPERATOR, VALUE
};

// Представляет компонент выражения: операцию или значение
struct EXPR_ITEM {
    ExprKind knd;
    double Value;
    OPERATOR op;

    EXPR_ITEM():
        op(OPERATOR::ILLEGAL),
        Value(0),
        knd(ExprKind::ILLEGAL_EXP)
    {}

    bool SetOperator( OPERATOR op ){
        this->op = op;
        this->knd = ExprKind::OPERATOR;
        return true;
    }

    bool SetValue(double value) {
        this->knd = ExprKind::VALUE;
        this->Value = value;
        return true;
    }

    string toString() {
        DumpContents();
        return "";
    }

private:
    void DumpContents() { /* код для краткости опустим */ }
};
```



¹ Представленное здесь решение не очень изящно: в каждом объекте типа EXPR_ITEM содержатся сразу два поля, отвечающих за «полезную нагрузку»: числовое значение и код операции, но при этом лишь одно из них (какое именно – зависит от значения поля knd) может иметь смысл. Между тем стандарт C++ 17 включает шаблон `std::variant`, предназначенный именно для таких случаев: в любой момент времени объект типа `std::variant<T1,...,Tn>` содержит значение ровно одного из типов-параметров. В контексте данной главы стоит упомянуть, что стандартный механизм для работы с такими объектами основан на шаблоне «Посетитель». – Прим. ред.

В структуре данных `std::list<EXPR_ITEM>` будет храниться содержимое компонента-выражения, преобразованное в линейную последовательность узлов. Теперь можно создать класс посетителя, который выполняет разглаживание композита.

```
// Разглаживающий посетитель
class FlattenVisitor : public IExprVisitor {
    list<EXPR_ITEM> ils;

    EXPR_ITEM MakeListItem(double num) {
        EXPR_ITEM temp;
        temp.SetValue(num);
        return temp;
    }

    EXPR_ITEM MakeListItem(OPERATOR op) {
        EXPR_ITEM temp;
        temp.SetOperator(op);
        return temp;
    }

public:
    FlattenVisitor() {}

    list<EXPR_ITEM> FlattenedExpr() { return ils; }

    double Visit(Number& num) {
        ils.push_back(MakeListItem(num.getNUM()));
        return 42;
    }

    double Visit(BinaryExpr& bin) {
        bin.getLeft().accept(*this);
        bin.getRight().accept(*this);
        ils.push_back(MakeListItem(bin.getOP()));
        return 42;
    }

    double Visit(UnaryExpr& un) {
        un.getRight().accept(*this);
        ils.push_back(MakeListItem(un.getOP()));
        return 42;
    }
};
```

Этот класс посетителя, получив объект-выражение, строит его представление в виде списка объектов типа `EXPR_ITEM`. Определим небольшую вспомогательную функцию, которая скрывает от пользователя детали преобразования.

```
list<EXPR_ITEM> ExprList(Expr* r) {
    unique_ptr<FlattenVisitor> fl(new FlattenVisitor());
    r->accept(*fl);
    return fl->FlattenedExpr();
}
```


Когда дерево объектов превращено в линейную последовательность, элементы последней можно обрабатывать с помощью итераторов. Например, линейную последовательность объектов, представляющих числовые значения и операции, можно подать на вход стековой вычислительной машины. Реализуем такую машину, начав со стека.

```
// Стек для вычисления выражений в обратной польской записи
class DoubleStack : public stack<double> {
public:
    DoubleStack() {}
    void Push( double a ) { this->push(a); }
    double Pop() {
        double a = this->top();
        this->pop();
        return a;
    }
};
```

Представленный выше класс `DoubleStack` представляет собой обёртку над контейнером из стандартной библиотеки. Наличие такого промежуточного класса позволит нам сделать дальнейший код более компактным. Теперь создадим вычислитель линеаризованных выражений. Алгоритм работы этого вычислителя состоит в том, чтобы пройти по списку объектов типа `EXPR_ITEM` и для каждого из них выполнить соответствующее действие: если этот элемент представляет числовую константу – втолкнуть её значение в стек; если же элемент представляет операцию – извлечь из стека один или два операнда (в зависимости от операции), применить к ним операцию и втолкнуть в стек её результат. Когда весь список таким образом обработан, в стеке должно остаться ровно одно значение – оно и является результатом вычисления всего выражения.

```
// Итеративная обработка компонентов выражения
double Evaluate(list<EXPR_ITEM> ls) {
    DoubleStack stk;
    double n;
    for (EXPR_ITEM s : ls) {
        if (s.knd == ExprKind::VALUE)
            stk.Push(s.Value);
        else if (s.op == OPERATOR::PLUS)
            stk.Push(stk.Pop() + stk.Pop());
        else if (s.op == OPERATOR::MINUS)
            stk.Push(stk.Pop() - stk.Pop());
        else if (s.op == OPERATOR::DIV)
        {
            n = stk.Pop();
            stk.Push(stk.Pop() / n);
        }
        else if (s.op == OPERATOR::MUL)
            stk.Push(stk.Pop() * stk.Pop());
        else if (s.op == OPERATOR::UNARY_MINUS)
            stk.Push(-stk.Pop());
    }
```



```

    }

    return stk.Pop();
}

// Превратить дерево в список и сразу подать его на вычислитель
double Evaluate( Expr* r ) { return Evaluate(ExprList(r)); }

```

Код данного интерпретатора выражений, заданных линейными списками, вполне очевиден: он всего лишь просматривает список компонентов и для каждого из них выполняет соответствующее действие. Представленный в предыдущем разделе код интерпретатора выражений, заданных в виде деревьев, выглядел куда сложнее. Осталось написать лишь программу, демонстрирующую работу этих функций.

```

int main() {
    unique_ptr<Expr> a(
        new BinaryExpr(
            new Number(10.0),
            new Number(20.0),
            OPERATOR::PLUS));
    double result = Evaluate(&(*a));
    cout << result << endl;
    return 0;
}

```



ОПЕРАЦИИ ОТОБРАЖЕНИЯ И ФИЛЬТРАЦИИ СПИСКОВ

Операция отображения `map` принимает в качестве аргументов список и некоторую функцию, применяет функцию к каждому элементу списка и из полученных результатов формирует новый список. Операция фильтрации `filter` принимает на вход список и предикат и строит новый список из тех и только тех элементов списка-аргумента, что удовлетворяют предикату. На этих двух операциях строится конвейерная обработка последовательностей в духе функционального программирования. Эти и подобные им операции называют функциями высшего порядка, так как они фактически превращают функции, работающие с отдельными элементами, в функции, работающие над списками. Ниже представлена одна из возможных реализаций функции `map`, работающая с любыми стандартными контейнерами¹.

```

template <typename R, typename F>
R Map(R r, F& fn) {
    std::transform(


```

¹ Данная реализация тривиальна и практического интереса не представляет, так как требует, чтобы функция-преобразователь сохраняла тип аргумента. Можно предложить читателю в качестве самостоятельного упражнения разработать такую функцию `Map`, которая преобразовывает тип элементов контейнера. Для этого потребуется овладеть метапрограммированием на шаблонах. – *Прим. перев.*

```

        std::begin(r),
        std::end(r),
        std::begin(r),
        std::forward<F>(fn));
    return r;
}

```



Покажем также возможную реализацию функции `filter`, работающую с контейнерами типа `std::list` и `std::vector`.

```

template <typename R, typename F>
R Filter(R r, F&& fn) {
    R ret(r.size());
    auto first = std::begin(r);
    auto const last = std::end(r);
    auto result = std::begin(ret);
    size_t inserted = 0;
    while (first != last) {
        if (fn(*first)) {
            *result = *first;
            ++inserted;
            ++result;
        }
        ++first;
    }
    ret.resize(inserted);
    return ret;
}

```

Конечно, в общем случае рекомендуется, где возможно, пользоваться средствами из стандартной библиотеки. Для фильтрации контейнеров в ней имеется функция `std::copy_if`, но нам было важно узнать общее число прошедших через фильтр элементов, поэтому пришлось написать собственную реализацию этой функции¹.

¹ Данное утверждение не выдерживает никакой критики. Можно предложить, по меньшей мере, три способа обойтись стандартным алгоритмом, избежав его собственно-ручной реализации. Во-первых, функция `std::copy_if` возвращает итератор на элемент после последнего вставленного в контейнер-приёмник, обозначим его через `last_inserted`. Если контейнер имеет тип `std::vector<T>`, то его итераторы относятся к категории итераторов произвольного доступа (англ. *random access*), и количество вставленных элементов легко узнать за время $O(1)$ с помощью стандартной функции `std::distance(last_inserted - std::begin(ret))`. Если же тип контейнера есть `std::list<T>`, его итераторы принадлежат к категории двунаправленных (англ. *bidirectional*) и, следовательно, не поддерживают операцию вычитания. Тогда функция `std::distance` отработает за время $O(N)$, что в данном случае вполне приемлемо. Второй способ ещё проще: функция `erase`, которая имеется в обоих классах, `std::list<T>` и `std::vector<T>`, удаляет из контейнера сегмент между двумя заданными итераторами. В данном случае нужно вызвать её следующим образом: `ret.erase(last_inserted, std::end(ret))`. Наконец, третий, наиболее элегантный способ: не резервировать заранее место в контейнере-приёмнике и воспользоваться специальным итератором для вставки в конец контейнера (`std::back_inserter`). – *Прим. перев.*

Следующая функция выводит содержимое списка на печать¹:

```
void Iterate(list<EXPR_ITEM>& s) {
    for (auto n : s) { std::cout << n.toString() << '\n'; }
}
```



Теперь займёмся главной функцией, которая бы демонстрировала работу всех этих конструкций. В ней используется композиция функций в конвейер. Напомним, что реализация этой операции находится в заголовочном файле `FuncCompose.h`.

```
int main() {
    unique_ptr<Expr> a(
        new BinaryExpr(
            new Number(10.0),
            new Number(20.0),
            OPERATOR::PLUS));
    // Разгладить дерево в список и отфильтровать его
    auto cd = Filter(
        ExprList(&(*a)),
        [](auto as) { return as.knd != ExprKind::OPERATOR; } );
    // возвести в квадрат и умножить на 3
    auto cdr = Map(
        cd,
        [](auto s) { s.Value *=3; return s; } |
        [](auto s) { s.Value *= s.Value; return s; } );
    Iterate(cdr);
    return 0;
}
```

Функция `Filter` создаёт новый список, содержащий лишь те элементы списка-аргумента, которые содержат числовые константы из исходного выражения. Затем функция `Map` применяет композицию функций к этому списку объектов и возвращает новый список².

¹ Читателю не стоит следовать этому образцу при создании собственных программ. Для вывода содержимого контейнера на печать гораздо лучше подходит копирование контейнера с помощью функции `std::copy` в поток вывода посредством специального итератора `std::ostream_iterator`. – Прим. перев.

² Данный раздел предоставляет лишь элементарное введение в обработку контейнеров в функциональном стиле. Наиболее интересные аспекты оставлены без внимания. Постараемся отчасти восполнить этот пробел. Предложенные автором реализации отображения и фильтрации строят контейнеры-результаты как структуры данных в памяти, что может оказаться накладно, если контейнеры содержат большое число элементов. Для преодоления этой трудности предназначены так называемые фильтрующие и преобразующие итераторы. Первые позволяют, не затрачивая времени и памяти на построение нового контейнера, «смотреть» на контейнер с исходными данными так, будто в нём видны лишь элементы, удовлетворяющие предикаты. Подобным же образом вторые позволяют «смотреть» на элементы исходного контейнера сквозь функцию-преобразователь, то есть «видеть» не элемент, хранящийся в контейнере, а значение функции-преобразователя. В терминологии

ОТ НАБЛЮДАТЕЛЕЙ К НАБЛЮДАЕМЫМ ИСТОЧНИКАМ

В предыдущих разделах мы разобрали, как преобразовать иерархический объект в линейную последовательность компонентов, которую затем можно обрабатывать с помощью итератора. Шаблон «Итератор» предполагает, что потребитель данных втягивает их из контейнера, дальнейшая обработка данных происходит уже на стороне потребителя. При этом возникает одна проблема: источник данных слишком сильно связан с приёмником. Шаблон «Наблюдатель» мало помогает избавиться от этой связи.

Создадим класс, который сможет взять на себя функцию концентратора событий – пусть именно на него подписываются приёмники данных. Иными словами, добавим в систему промежуточное звено между источником и приёмником событий. Одна из очевидных выгод от появления такого посредника состоит в возможности агрегировать, преобразовывать и фильтровать события до того, как они попадут к потребителю. Потребитель может просто передать концентратору свои правила преобразования и фильтрации событий и делегировать ему эту часть работы. Рассмотрим снова интерфейсы наблюдателя и наблюдаемого источника данных.

```
struct OBSERVER {
    int id;
    std::function<void(const double)> ondata;
    std::function<void()> oncompleted;
    std::function<void(const std::exception &)> onexception;
};

struct OBSERVABLE {
    virtual bool Subscribe(OBSERVER *obs) = 0;
    // метод Unsubscribe в данном примере не используется
};
```

«Банды четырёх» фильтрующие и преобразующие итераторы представляют собой декораторы над итераторами исходного контейнера. Помимо названных двух, существуют и иные декораторы над итераторами: декоратор, отбирающий из контейнера не более заданного числа элементов; декоратор, пропускающий заданное число элементов; декоратор, «переворачивающий» контейнер, т. е. меняющий направление итераторов на противоположное. Разумеется, эти декораторы можно применять не только к итераторам стандартных контейнеров, но и к декорированым итераторам. Это позволяет определять сколь угодно сложные алгоритмы поэлементной обработки контейнеров – скажем, в списке целых чисел удвоить каждый элемент, отобрать из полученных значений те, что делятся на три, преобразовать полученные числа в строки и выстроить в обратном порядке. Повторимся, что контейнеры с промежуточными результатами этих преобразований в памяти не хранятся, а представляют собой виртуальные сущности, видимые исключительно через итераторы. На сегодняшний день существует несколько библиотек, реализующих описанную здесь функциональность. Наибольшую популярность завоевали библиотеки Boost.Range и Range-v3, также интерес представляет библиотека think-cell range. – *Прим. перев.*

Понятия наблюдаемого источника и наблюдателя были вкратце освещены в главах 1 и 2. Всякий объект, генерирующий события, должен воплощать интерфейс OBSERVABLE, а объект, принимающий события, должен обладать интерфейсом OBSERVER. Интерфейс OBSERVER обязывает объект иметь следующие методы:

- метод ondata для приёма данных о событии;
- метод onexception для обработки ошибок, возникших на стороне отправителя;
- метод oncompleted, сигнализирующий об окончании потока данных.

Класс источника событий, воплощающий интерфейс OBSERVABLE, должен обладать следующими методами:

- метод Subscribe для подписки приёмника на оповещения от данного источника;
- метод Unsubscribe для отписки от оповещений (в нашем примере не используется).

Ниже показана упрощённая реализация этих интерфейсов.

```
template<class T, class F, class M, class Marg, class Farg>
class EventSourceValueSubject: public OBSERVABLE {
    vector<OBSERVER> sinks;
    T *State;
    std::function<bool(Farg)> filter_func;
    std::function<Marg(Marg)> map_func;
```

Функции map_func и filter_func нужны для того, чтобы преобразовывать и фильтровать сообщения перед их асинхронной рассылкой подписчикам. Эти функции передаются в качестве параметров при инициализации объекта. В данном упрощённом примере мы исходим из предположения, что источник событий содержит лишь один объект-выражение. Вместо этого можно было бы в источнике событий хранить список выражений и отправлять подписчикам поток значений. При текущей реализации, однако, достаточно отправить наблюдателям одиночное значение.

```
public:
    EventSourceValueSubject(Expr *n, F&& filter, M&& mapper) {
        State = n;
        map_func = mapper;
        filter_func = filter;
        NotifyAll();
    }


    ~EventSourceValueSubject() { sinks.clear(); }

    virtual bool Subscribe(OBSERVER *sink) {
        sinks.push_back(*sink);
        return true;
    }
}
```

В этом примере сделано предположение, что объектом-выражением владеет не объект-источник событий, а внешний контекст, поэтому деструктор клас-

са `EventSourceValueSubject` не уничтожает выражение. В реальном приложении, скорее всего, для управления временем жизни выражения использовался бы умный указатель `std::shared_ptr`. Кроме того, для краткости мы не стали реализовывать метод отписки `Unsubscribe`. Конструктор источника событий в качестве аргументов принимает объект-выражение, предикат для фильтрации событий и функцию преобразования событий, которая может быть получена композицией функций посредством операции `|`.

```
void NotifyAll() {
    double ret = Evaluate(State);
    list<double> ls;
    ls.push_back(ret);
    auto result = Map(ls, map_func);
    auto resulttr = Filter(result, filter_func);
    if (resulttr.size() == 0) { return; }
```



Эта функция вычисляет значение выражения, завёрнутого в объект-источник событий, и помещает его в список. Функции `Map` и `Filter` преобразовывают этот список и отбрасывают из него элементы (пока что единственный элемент), не удовлетворяющие заданному условию. Может показаться, что обрабатывать список из ровно одного элемента – излишнее усложнение программы, но в следующих разделах мы доработаем этот пример так, что список будет состоять из произвольного числа элементов.


```
double dispatch_number = resulttr.front();
for (auto sink : sinks) {
    std::packaged_task<int> task([&]() {
        sink.ondata(dispatch_number);
        return 1;
    });
    std::future<int> result = task.get_future();
    task();
    double dresult = result.get();
}
}
```

В этом фрагменте кода для каждого приёмника событий создаётся своя асинхронная задача, завёрнутая в объект типа `std::packaged_task`, которая должна доставить событие этому приёмнику. В реальных программных системах для этого используется ещё одна вспомогательная сущность, называемая планировщиком (англ. scheduler). Благодаря асинхронному механизму рассылки приёмники событий не могут заблокировать объект-источник. В этом состоит одно из важнейших преимуществ наблюдаемого источника событий.

```
T* GetState() { return State; }

void SetState(T *pstate) {
    State = pstate;
    NotifyAll();
}

};
```



Эти вспомогательные методы позволяют получить из объекта-источника событий его текущее выражение и поместить в него новое выражение.

Для демонстрации работы нашей системы понадобится много тестовых исходных данных. Следующая функция использует генератор случайных чисел с равномерным законом распределения и генерирует случайные выражения. Равномерное распределение выбрано наугад, читатель может вместо него подставить любое другое и посмотреть, как изменится результат выполнения программы.

```
Expr *getRandomExpr(int start, int end) {
    std::random_device rd;
    std::default_random_engine reng(rd());
    std::uniform_int_distribution<int> uniform_dist(start, end);
    double mean = uniform_dist(reng);
    return new BinaryExpr(
        new Number(mean*1.0),
        new Number(mean*2.0),
        OPERATOR::PLUS);
}
```

Теперь остаётся написать главную функцию, демонстрирующую работу всех описанных выше частей. Сначала она создаёт объект класса EventSourceValueSubject, передавая ему начальные параметры: выражение, фильтр и преобразователь сообщений:

```
int main() {
    unique_ptr<Expr> a(
        new BinaryExpr(
            new Number(10.0),
            new Number(20.0),
            OPERATOR::PLUS));
    EventSourceValueSubject<
        Expr,
        std::function<bool(double)>,
        std::function<double(double)>,
        double,
        double>
    > temp(
        &(*a),
        [] (auto s) {return s > 40.0;},
        [] (auto s) {return s+s; } | [] (auto s) {return s*2;});
}
```

Функция-преобразователь образована композицией из двух лямбда-выражений. Подобным образом можно состыковывать сколь угодно длинные цепочки функций. В следующих главах, когда речь пойдёт о библиотеке RxCpp, мы будем часто пользоваться этим приёмом.

```
OBSERVER obs_one ;
obs_one.ondata = [](const double r) {
    cout << "*Значение " << r << endl;
};
```



```
OBSERVER obs_two ;
obs_two.ondata = [] (const double r) {
    cout << "***Значение " << r << endl;
};
```

Здесь созданы два объекта-наблюдателя, а их полям `ondata` присвоены лямбда-функции. Остальные поля-обработчики в этом примере не используются. Их было бы несложно реализовать, но для иллюстрации принципа действия реактивной системы довольно обработчика `ondata`.

```
temp.Subscribe(&obs_one);
temp.Subscribe(&obs_two);
```

Два объекта-наблюдателя подписаны на получение обновлений от объекта-источника.

```
Expr *expr = 0;
for (int i= 0; i < 10; ++i ) {
    cout << "-----" << i << " " << endl;
    expr = getRandomExpr(i*2, i*3);
    temp.SetState(expr);
    std::this_thread::sleep_for(2s);
    delete expr;
}
}
```

Последний фрагмент кода генерирует случайные выражения и подставляет их в объект-источник событий. Последний вычисляет значение выражения, преобразовывает его и, если преобразованное значение удовлетворяет условию фильтрации, рассылает его всем наблюдателям. Напомним, что реализованный нами источник событий не может быть заблокирован недобросовестным приёмником, так как работает асинхронно.

Таким образом, мы разобрали следующие вопросы:

- моделирование абстрактного синтаксического дерева выражения на основе шаблона «Композит»;
- обработка композитных объектов с помощью шаблона «Посетитель»;
- разглаживание дерева в линейную последовательность объектов и обработка последней с помощью итераторов, реализующих семантику втягивания;
- «выворачивание» итератора и превращение его в источник событий, основанный на семантике вталкивания.

Итоги

В этой главе мы изучили обширный материал и ещё более приблизились к постижению реактивной модели программирования. Читатель узнал о шаблоне «Банды четырёх» под названием «Наблюдатель» и о его недостатках. Небольшой экскурс в философию понадобился, чтобы объяснить триединый взгляд на мироздание: с точки зрения структуры, поведения и целевого назначения

сущего. Затем была изучена неразрывная пара шаблонов «Композит» и «Посетитель» в приложении к задаче обработки абстрактных синтаксических деревьев. Далее читатель узнал, как иерархический, многоуровневый композит «выровнять» в линейный список, элементы которого можно перебирать с помощью итератора. Наконец, немного изменив архитектуру системы, мы получили наблюдаемые источники событий. Обычно такие источники работают с потоками событий, но в нашем упрощённом примере источник генерировал одиночное событие. Вопросам обработки потоков событий посвящена следующая глава, тем самым будет завершена подготовительная работа, необходимая для изучения реактивного программирования.



Глава 6

.....

Введение в программирование потоков событий на языке C++

Эта глава – последняя в череде подготовительных глав, предваряющих рассказ о реактивном программировании на языке C++. Причина, по которой перед рассмотрением основного предмета книги пришлось сделать столь пространное введение, состоит в том, что реактивная модель программирования объединяет в себе множество понятий из различных разделов программирования, и все они необходимы для придания ей стройности и прочности. Чтобы научиться мыслить по-реактивному, программист должен хорошо владеть объектно-ориентированным и функциональным стилями программирования, встроенными в язык средствами работы с потоками, техниками неблокирующего параллельного программирования, моделью асинхронных задач, шаблонами проектирования, алгоритмами планировщика задач, моделью потоков данных, декларативным стилем программирования и даже в некоторой степени теорией графов! Эта книга начиналась с обзора моделей событийно-управляемого программирования, лежащих в основе нескольких графических оболочек, и связанных с ними способов структурирования кода вокруг обработки событий. Затем в главе 2 были рассмотрены важнейшие новшества, появившиеся в стандарте языка C++. Глава 3 была посвящена появившимся в языке средствам для работы с параллельными потоками вычислений, а в главе 4 рассказано об асинхронных задачах и техниках неблокирующего программирования. Наконец, в главе 5 реактивная модель программирования была показана сквозь призму шаблонов проектирования «Банды четырёх». Осталось разобрать обработку потоков событий. В этой главе будут рассмотрены следующие вопросы:

- в чем состоит модель программирования, основанная на потоках данных;
- преимущества программирования в терминах потоков данных;
- обработка потоков данных на языке C++ с использованием общедоступных библиотек;
- обработка потоков данных с помощью встраиваемого в язык C++ предметно-ориентированного языка Streamulus;
- обработка потоков событий как частного случая потоков данных.

Что такое программирование потоков данных

Прежде чем погружаться в подробности модели программирования потоков данных, отступим на шаг назад и проследим параллели с моделью программирования, присущей языку команд в стандарте POSIX. В сценариях командного интерпретатора, как правило, всякая команда есть программа и всякая программа есть команда. Команды можно сочленять, подавая выход одной программы на вход другой, чтобы вместе они обеспечивали выполнение определённой задачи. Более того, для решения сложных задач можно выстраивать сколь угодно длинные цепочки команд. Работу таких цепочек можно представить себе в виде потока данных, последовательно проходящего через различные фильтры и преобразователи. Сочленение команд в цепочки можно называть их *композицией*. В реальном мире встречаются ситуации, когда большую и сложную программу удаётся заменить небольшим командным сценарием, который выстраивает композицию из нескольких простых программ. Этот же принцип можно воплотить и в программе на языке C++, если данные, подаваемые на вход функции, представить в виде потока, последовательности или списка одиночных элементов. Передача данных с выхода одной функции (или функционального объекта) на вход следующей может происходить через стандартный контейнер, играющий роль промежуточного буфера.

i Однажды ведущий колонки «Жемчужины программирования» Джон Бентли¹ обратился к живой легенде программирования, профессору Стэнфордского университета Д. Кнуту с просьбой написать подробно откомментированную программу, которая в поступающем на её вход тексте отыскивает *n* наиболее часто встречающихся слов и печатает их в алфавитном порядке вместе с частотами их встречаемости. Решение Д. Кнута состояло из десяти страниц кода на языке Паскаль и включало в себя придуманную Кнутом специально для этой задачи хитроумную структуру данных! Дуглас Макилрой, автор реализации конвейера для командной оболочки UNIX, предложил собственное решение этой задачи, состоящее из шести команд, соединённых конвейером:

```
tr -cs A-Za-z '\n' | tr A-Z a-z | sort | uniq -c
| sort -rn | sed ${1}q
```

Вот какова мощь композиции команд!

¹ Позднее на основе материалов этой колонки Дж. Бентли выпустил знаменитую и поныне книгу «Жемчужины творчества программистов», которую, пользуясь случаем, хочется порекомендовать читателю. – *Прим. перев.*

Преимущества модели программирования потоков данных

Ставший традиционным объектно-ориентированный подход к программированию хорошо подходит для моделирования разнообразных иерархий. Обработка иерархических структур данных, как правило, значительно сложнее обработки линейных последовательностей. В модели программирования, основанной на потоках данных, входные данные можно рассматривать как поток однородных элементов, заключённый в некоторый контейнер, а результаты обработки – как линейную коллекцию других однородных элементов. Обработка не изменяет сами по себе исходные объекты данных, а строит на их основе новые объекты и помещает их в новый контейнер. Воспользовавшись приёмами обобщённого программирования на языке C++, можно реализовать систему обработки потоков данных, абстрагированную от фактического типа контейнеров, используемых для хранения промежуточных данных. Перечислим некоторые из преимуществ такой модели программирования:

- модель потоков данных упрощает логическую структуру программы;
- потоки данных способствуют применению ленивой модели вычислений и функционального стиля обработки данных;
- модель потоков данных способствует распараллеливанию вычислений (в том числе и потому, что исходные данные не подвергаются изменению);
- модель позволяет легко строить композиции сложных функций из более простых;
- модель способствует программированию в декларативном стиле;
- модель позволяет агрегировать, фильтровать и преобразовывать данные от различных источников;
- модель потоков данных способствует изоляции источников данных от обработчиков;
- программный код получается более очевидным и логичным;
- модель потоков данных хорошо совмещается с моделью асинхронных задач и асинхронной передачей данных;
- при создании своих алгоритмов обработки данных можно пользоваться сотнями потоковых операций, доступных в открытых источниках.

ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ STREAMS

Для создания прикладных программ на основе модели потоков данных удобно пользоваться общедоступной библиотекой Streams, созданной Дж. Шайнерманом (Jonah Scheinerman). Исходный код библиотеки можно найти по адресу <https://github.com/jscheiny/Streams>, а документацию к ней – по адресу <https://jscheiny.github.io/Streams/api.html>. Библиотеку можно кратко охарактеризовать следующим образом (согласно странице библиотеки в системе GitHub):

Streams – это библиотека для программирования на языке C++, которая предоставляет возможность ленивых вычислений и преобразования данных в функциональном стиле, упрощая использование контейнеров и алгоритмов из стандартной библиотеки. Библиотека Streams поддерживает множество операций, характерных для функционального программирования, например поэлементное отображение `map`, поэлементную фильтрацию `filter`, свёртку по бинарной операции `reduce`, а также множество других полезных операций: объединение, пересечение и разность множеств, преобразование последовательности значений в последовательности её частичных сумм и смежных разностей и многие другие.

Можно ожидать, что программист, хорошо освоивший стандартную библиотеку шаблонов (англ. standard template library, STL), будет себя комфортно чувствовать и с библиотекой Streams. Контейнеры библиотеки STL в ней трактуются как потоки исходных данных. В библиотеке используются идиомы функционального программирования, поддержка которых появилась в современном стандарте языка C++, также в ней реализован ленивый принцип вычислений. Идея ленивых вычислений чрезвычайно важна в контексте нашего рассмотрения, она лежит в основе как функционального программирования, так и реактивной модели программирования.

ЛЕНИВЫЕ ВЫЧИСЛЕНИЯ

При всём разнообразии языков программирования в них воплощены лишь два способа передачи аргументов в вызываемые функции:

- аппликативный порядок (англ. applicative order, AO);
- нормальный порядок (англ. normal order, NO).

При аппликативном порядке аргументы вычисляются вызывающей стороной, затем передаются вызываемой функции. Большая часть широко распространённых языков программирования (в том числе и язык C++) следует этому принципу. В случае же нормального порядка вычисление значений аргументов откладывается до тех пор, пока они не понадобятся в ходе выполнения вызванной функции¹. Некоторые языки функционального программирования,

¹ Связанные между собой понятия ленивых вычислений и нормального порядка вычислений заслуживают более развёрнутого описания. Рассмотрим выражение `f(0, g(1))`. Оно означает вызов функции `f` с двумя аргументами, значение первого уже дано в готовом виде, тогда как второй аргумент задан, в свою очередь, выражением, требующим вычисления. При аппликативном порядке вычислений (например, в языке C++) будет сначала вычислено значение выражения `g(1)` и сохранено в некоторой области памяти (говоря упрощённо, во вспомогательной переменной), затем будет вызвана функция `f` с передачей двух уже имеющихся в наличии значений аргументов. При нормальном же порядке вычислений всё, что фактически передаётся функции `f` в качестве второго аргумента, – это информация о том, что значение данного аргумента *можно будет вычислить в любой момент, если оно понадобится* (а именно вызвав функцию `g` с передачей значения 1). При этом вполне



в частности Haskell, F# и ML, основываются на нормальном порядке вычислений. Для языков функционального программирования характерна референциальная прозрачность, означающая в том числе и отсутствие побочных эффектов у вызова функций. Поэтому, в принципе, допустимо любое выражение, встретившееся в ходе выполнения программы, вычислять лишь один раз для каждого значения аргумента и помещать результат в таблицу; если в будущем встретится то же выражение с теми же значениями входящих в него аргументов, можно просто отыскать в таблице уже вычисленное значение и использовать его повторно. Таким образом, принцип ленивых вычислений (не вычислять выражение до тех пор, пока его значение не понадобится) тесно переплетён с референциальной прозрачностью и функциональной чистотой (не имеет значения, когда и сколько раз вычислять значение функции при одних и тех же аргументах). В языке C++ нет встроенной поддержки ленивого порядка вычислений, но её можно реализовать стандартными средствами языка при известной изобретательности.

Пример программы для обработки потока данных

Для первого знакомства с библиотекой `Streams` напишем небольшую программу, которая генерирует поток чисел и вычисляет сумму квадратов первых десяти из них.

```
#include "Stream.h"
using namespace std;
using namespace stream;
using namespace stream::op;

int main()
{
    int total = MakeStream::counter(1)
        | map_([] (int x) { return x * x; })
        | limit(10)
        | sum();

    cout << total << endl;
    return 0;
}
```



Из этого кода видно, что источник данных, их дальнейшие обработчики и приёмник окончательного результата соединены в конвейер. В начале кон-

может оказаться, что машине вообще не придётся выполнять функцию `g` – если при данном значении первого аргумента значение второго не требуется для вычисления значения функции `f`. Это фундаментальное отличие между двумя порядками вычислений способно кардинальным образом изменить само понятие программы и подход к программированию. Так, нормальный порядок вычислений резко расширяет возможности рекурсии и позволяет работать с бесконечными структурами данных, которые строятся бесконечной рекурсией. – *Прим. перев.*

вейера стоит генератор бесконечной последовательности целых чисел, начиная с 1, далее каждое число возводится в квадрат, из полученной бесконечной последовательности квадратов берутся первые десять значений, которые подаются на вход сумматора.

Агрегирование значений в парадигме потоков данных

Теперь, когда мы разобрали элементарные основы программирования потоков данных на примере библиотеки Streams, разберём более сложный пример – программу, которая вычисляет среднее арифметическое значений, хранящихся в контейнере `std::vector`.

```
#include "Stream.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
using namespace stream;
using namespace stream::op;

int main() {
    std::vector<double> a = { 10,20,30,40,50 };
    // преобразовать контейнер в поток данных и просуммировать
    auto val = MakeStream::from(a)
        | reduce(std::plus<void>());
    // вычислить среднее арифметическое
    cout << val/a.size() << endl;
    return 0;
}
```



В этой программе сначала создаётся стандартный вектор чисел, затем на его основе создаётся поток значений, над которым выполняется свёртка по операции сложения (с помощью стандартного функционального объекта `std::plus`). В конце алгоритма накопленная сумма элементов делится на число элементов в исходном векторе.

Погружение стандартных контейнеров в парадигму потоков данных

Библиотека Streams позволяет прозрачно работать с контейнерами библиотеки STL. Следующий пример демонстрирует, как вектор исходных данных преобразуется в поток, затем к каждому элементу потока данных применяется функция-преобразователь, а полученный в результате этого поток значений снова преобразуется в вектор. Вектор, построенный из потока данных, можно обрабатывать обычным способом, с помощью итератора.

```
#include "Stream.h"
#include <iostream>
```



```

#include <vector>
#include <algorithm>
#include <functional>
#include <cmath>
using namespace std;
using namespace stream;
using namespace stream::op;

double square(double a) { return a*a; }

int main() {
    std::vector<double> values = { 1,2,3,4,5 };

    std::vector<double> outputs = MakeStream::from(values)
        | map_([&](double a) { return a*a; })
        | to_vector();

    for(auto pn : outputs )
        cout << pn << endl;

    return 0;
}

```



Библиотека Streams снабжена весьма подробной документацией и многочисленными примерами, что позволяет любому пользователю создавать с её помощью высококачественные прикладные программы. Читателю стоит ознакомиться со справочным руководством по адресу <https://jscheiny.github.io/Streams/api.html>.

Несколько слов о библиотеке Streams

Библиотека Streams в целом представляет собой тщательно спроектированный программный продукт с чёткой и понятной системой абстракций. Любой программист, привычный к функциональному программированию и потоку данных, сможет уверенно освоить её за считанные часы. Читатели, знакомые лишь с библиотекой STL, также сочтут библиотеку Streams интуитивно понятной. Что касается положенной в основу библиотеки программной модели, то функции, составляющие её программный интерфейс, можно разделить на следующие категории:

- ядро (инициализация потоков данных);
- генераторы (создание потоков данных);
- операции преобразования потоков данных, обладающие внутренним состоянием;
- преобразователи потоков данных, не имеющие состояния;
- завершающие операции.

В документации, ссылка на которую приведена выше, каждая из этих категорий освещена подробно.



ПРОГРАММИРОВАНИЕ ПОТОКОВ СОБЫТИЙ

Из предыдущего раздела читатель составил представление о модели программирования, основанной на понятии потока данных. Если же данные, объединённые в поток, представляют собой события, можно говорить о программировании потоков событий как об особом частном случае. В сообществе программистов событийно-управляемые архитектуры считаются наиболее подходящей моделью для создания современных программ для множества различных предметных областей. Хорошим примером программной системы, основанной на модели потоков событий, может служить система управления версиями. Системы управления версиями имеют дело с событиями множества разновидностей: это может быть выгрузка кода в локальную копию, запись изменений, откат изменения, создание ветки и др.

Преимущества программирования на основе потоков событий

Объединение событий в потоки и их обработка каскадом преобразователей и фильтров даёт ряд преимуществ по сравнению с традиционными подходами к событийно-управляемому программированию, некоторые из которых перечислены ниже:

- источники и приёмники событий изолированы друг от друга;
- приёмники могут обрабатывать события, не обременяя себя подробностями их источников;
- к потокам событий можно применять высокоуровневые операции преобразования и фильтрации;
- преобразование и фильтрация могут применяться к результатам агрегирования сообщений;
- события можно передавать для обработки по сети;
- обработку событий можно легко сделать параллельной.

Библиотека Streamulus и её программная модель

Библиотека Streamulus, разработанная Ирит Катриэль (Irit Katriel), значительно упрощает программирование потоков событий благодаря положенной в её основу модели, включающей предметно-ориентированный встроенный язык (англ. domain-specific embedded language, DSEL). Чтобы понять суть этой программной модели, рассмотрим пример программы, которая направляет поток данных в объект пользовательского класса, агрегирующий полученные значения.

```
#include "streamulus.h"
#include <iostream>
using namespace std;
using namespace streamulus;

struct print {
    static double temp;
```

```

print() { }
template<typename T>
T operator()(const T& value) const
{
    print::temp += value;
    std::cout << print::temp << std::endl;
    return value;
}
};

double print::temp = 0;

```

Этот функциональный объект накапливает в статической переменной¹ сумму переданных ему чисел. При каждом обращении к объекту данного класса как к функции (этот вызов осуществляется через промежуточный объект типа `Streamify<print>`, см. ниже) текущее накопленное значение выводится на консоль. Некоторые подробности станут яснее из следующего кода:

```

void hello_stream()
{
    using namespace streamulus;
    // создать входной поток данных с именем "Input Stream"
    InputStream<double> s = NewInputStream<double>(
        "Input Stream",
        true /* выводить подробную информацию: да */);
    // создать ядро обработки потоков событий
    Streamulus streamulus_engine;

```

С помощью функции-шаблона `NewInputStream<T>` создаётся поток действительных чисел. Второй аргумент логического типа определяет, будет ли система выводить на консоль подробные сведения о движении данных из этого потока: передав в этот аргумент значение `false`, можно отключить данный режим. Чтобы привести в действие всю систему в целом, нужно создать и запустить ядро обработки потоков данных. В частности, ядро само осуществляет топологическую сортировку объектов-потоков, чтобы определить, в каком направлении и в каком порядке должны передаваться изменения, происходящие в потоках данных.

```

// Для каждого значения из входного потока данных:
// прибавить к накопителю суммы и напечатать текущую сумму
streamulus_engine.Subscribe(Streamify<print>(s));

```

Функция-шаблон `Streamify<f>` превращает определённый выше функциональный класс `print`, способный обработать отдельно взятое число, в операцию по обработке потока данных. Операции над потоками данных иногда называ-

¹ Вряд ли нужно обосновывать, насколько неудачно это решение. Наличие глобального состояния, разделяемого всеми объектами класса, делает невозможной параллельную обработку нескольких потоков данных разными объектами этого класса. — *Прим. перев.*

ют *стропами*, от английского *stream operator*. Программист может объявлять собственные стропы, хотя в большинстве случаев вполне хватает возможностей функции *Streamify*. Внутри себя она создаёт один функциональный объект функционального класса и оборачивает его в строп. Затем эта операция подключается в качестве обработчика к созданному ранее потоку данных *s*.

```
// Поместить значения во входной поток данных
InputStreamPut<double>(s, 10);
InputStreamPut<double>(s, 20);
InputStreamPut<double>(s, 30);
}

int main()
{
    hello_stream();
    return 0;
}
```

Этот фрагмент кода помещает в поток данных некоторые начальные значения. По мере появления данных во входном потоке система будет автоматически вызывать подключённые к нему обработчики. В нашем случае обработчик будет печатать накопленную сумму полученных из потока значений.

Этот элементарный пример позволяет составить общее представление об устройстве и принципе действия программ, основанных на системе *Streamulus*. Рассмотрим теперь более сложную программу, лучше иллюстрирующую возможности библиотеки. Следующая программа пропускает поток данных сквозь каскад функций одного аргумента. Этот пример также демонстрирует, насколько свободно можно обращаться с операциями над потоками данных.

```
#include "streamulus.h"
#include <iostream>
using namespace std;
using namespace streamulus;

// функциональные объекты для преобразования числовых значений
struct twice {
    template<typename T>
    T operator()(const T& value) const { return value*2; }
};

struct neg {
    template<typename T>
    T operator()(const T& value) const { return -value; }
};

struct half{
    template<typename T>
    T operator()(const T& value) const { return 0.5*value; }
};
```

Эти классы служат обёртками над чисто арифметическими операциями. Так, функциональный объект класса `twice` удваивает аргумент, объект класса `neg` меняет знак аргумента на противоположный, а объект класса `half` уменьшает значение аргумента наполовину.

```
struct print{
    template<typename T>
    T operator()(const T& value) const{
        std::cout << value << std::endl;
        return value;
    }
};

struct as_string
{
    template<typename T>
    std::string operator()(const T& value) const {
        std::stringstream ss;
        ss << value;
        return ss.str();
    }
};
```



Как работают эти два функциональных объекта, вполне очевидно. Первый из них выводит значение аргумента на консоль и возвращает свой аргумент неизменным. Второй переводит значение своего аргумента, какого бы типа оно ни было, в текстовую строку, используя для преобразования стандартный класс строкового потока `std::stringstream`.

```
void DataFlowGraph(){
    // создать именованный поток чисел
    InputStream<double> s = NewInputStream<double>(
        "Input Stream",
        false /* печатать отладочную информацию: нет */);
    Streamulus streamulus_engine;
    // определить граф обработки событий
    Subscription<double>::type val2 = streamulus_engine.Subscribe(
        Streamify<neg>(
            Streamify<half>(2*s)));
    Subscription<double>::type val3 = streamulus_engine.Subscribe(
        Streamify<twice>(val2*0.5));
    streamulus_engine.Subscribe(
        Streamify<print>(
            Streamify<as_string>(val3*2)));
    // послать данные во входной поток данных
    for (int i=0; i<5; i++)
        InputStreamPut(s, (double)i);
}

int main(){
```

```

    DataFlowGraph();
    return 0;
}

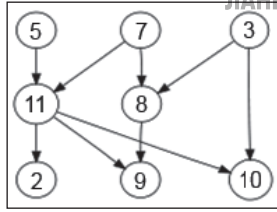
```

В функции `DataFlowGraph` в первую очередь создаётся поток исходных данных типа действительных чисел двойной точности. После инициализации ядра системы строится длинный конвейер операций по обработке потоков данных, которые, в свою очередь, получены из операций над отдельными значениями с помощью функции `Streamify<f>`. Последнюю можно считать разновидностью операции композиции функций одного аргумента. Когда все детали механизма настроены, можно привести его в действие, отправив данные во входной поток данных посредством функции `InputStreamPut`.

Устройство библиотеки *Streamulus*: взгляд изнутри

Для того чтобы распространять изменения от одних потоков данных к другим, внутренние механизмы библиотеки *Streamulus* используют графы. Вершинами этого графа являются обработчики сообщений, а рёбра – буферы, в которых хранятся сообщения на пути от одного обработчика к другому. Библиотека *Streamulus* берёт на себя построение графа зависимостей между переменными, моделирующими потоки сообщений. Порядок, в котором переменные извещаются об обновлениях, определяется топологической сортировкой этого графа.

С точки зрения математики, граф – это множество *вершин* (или *узлов*), соединённых *рёбрами*. В прикладных задачах вершины служат моделями некоторых сущностей (городов, людей, дел и т. д.), а рёбра – моделями некоторых отношений между ними (соответственно, наличие транспортного сообщения между городами, дружбы или подчинения между людьми, зависимости дела от результатов другого дела). В кибернетике, особенно там, где речь идёт о всевозможных расписаниях и об анализе зависимостей между разными сущностями, особенно удобны графы специального вида – *направленные ациклические графы* (англ. *directed acyclic graph*, DAG), также называемые *гамаками*. Гамак представляет собой орграф (т. е. граф, у которого каждое ребро имеет определённое направление: ведёт от одной вершины к другой, но не наоборот) без циклов (начиная свой путь от какой угодно вершины и следуя только по рёбрам, невозможно вернуться в исходную вершину). Замечательная особенность гамаков – возможность топологической сортировки вершин. Вершины гамака можно выстроить в линейной последовательности так, чтобы для любой вершины *a* все вершины, достижимые из *a*, стояли после неё. Из всех графов топологическая сортировка возможна только для гамаков. В общем случае для одного гамака топологическую сортировку можно выполнить более, чем одним способом. Так, для представленного на рисунке графа можно построить следующие топологические сортировки:



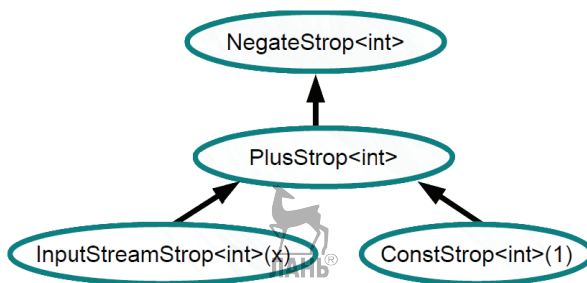
- 5, 7, 3, 11, 8, 2, 9, 10 – слева направо, сверху вниз;
- 3, 5, 7, 8, 11, 2, 9, 10 – сначала вершины с наименьшими возможными номерами;
- 5, 7, 3, 8, 11, 10, 9, 2 – в первую очередь без рёбер;
- 7, 5, 11, 3, 10, 8, 9, 2 – сначала вершины с наибольшими возможными номерами;
- 5, 7, 11, 2, 3, 8, 9, 10 – в первую очередь сверху вниз, затем слева направо;
- 3, 7, 8, 5, 11, 10, 2, 9 – без определённого правила.

Обработка выражений в библиотеке *Streamulus*

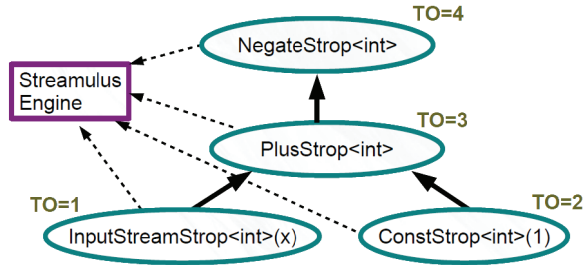
Рассмотрим, как библиотека *Streamulus* обрабатывает выражения, на примере следующего простого потока данных:

```
InputStream<int>::type x = NewInputStream<int>("X");
Engine.Subscribe(-(x+1));
```

Выражение $-(x+1)$ задаёт поток данных и, с точки зрения внутренних механизмов библиотеки, преобразуется в показанный ниже граф. Напомним, что термин «строп» (англ. *strop*) означает «операция над потоком данных» (*stream operation*). Каждая вершина графа представляет один строп.



Когда вершины и рёбра графа созданы, производится топологическая сортировка с целью определить порядок вычисления компонентов выражения. Результат сортировки показан на следующем рисунке (пметка «ТО» означает «topological order», т. е. топологический порядок):



Ядро системы Streamulus обходит вершины и рёбра графа, чтобы определить правильный порядок выполнения строп и тем самым порядок распространения данных по графу. Затем стропы размещаются ядром в линейной последовательности согласно топологическому порядку. В дальнейшем они будут выполняться именно в этой линейной последовательности.



Ядро системы Streamulus использует для этих манипуляций библиотеку `proto` из коллекции Boost. Вся обработка деревьев выражений возлагается на неё. Чтобы полностью разобраться в исходном коде библиотеки Streamulus, читателю понадобится основательное знание метапрограммирования на шаблонах, особенно техники, известной под названием *шаблон выражения*. Напомним, что метапрограммирование – это создание кода, который на этапе компиляции генерирует код, предназначенный для выполнения. Возможность метапрограммирования первоначально не была предусмотрена авторами языка C++, но в 1994 г. Эрвином Унру (Erwin Unruh) было обнаружено, что имеющийся в языке механизм шаблонов полон по Тьюрингу, т.е., в принципе, позволяет на этапе компиляции выполнять любые вычисления – в том числе и генерировать код.

Библиотека Spreadsheet для оповещения об изменениях данных

Электронные таблицы (англ. *spreadsheet*) часто называют наиболее характерным примером реактивных систем. Лист электронной таблицы представляет собой матрицу ячеек. Значения некоторых ячеек просто вводятся пользователем, но у некоторых иных могут вычисляться по формулам и зависеть от значений других ячеек. Всякий раз, когда значение какой-либо ячейки изменяется, это изменение должно распространиться на все ячейки, зависящие от неё. Но когда меняется значение какой-либо зависимой ячейки, от неё могут зависеть, в свою очередь, другие ячейки – таким образом, изменение должно распространиться дальше. Программирование электронных таблиц становится довольно простым делом, если в распоряжении программиста имеется такая библиотека, как Streamulus. К счастью, автор библиотеки Streamulus разработала ещё одну библиотеку, надстроенную над ней и предназначенную именно для работы с каскадами изменений, подобных тем, что имеют место в электронных таблицах.

i Предназначенная для языка C++ библиотека Spreadsheet позволяет программировать в стиле электронных таблиц, т. е. создавать в своей программе переменные, ведущие себя подобно ячейкам таблицы. Каждой такой переменной-ячейке можно назначить выражение, зависящее от других переменных-ячеек. Изменения распространяются по цепочкам зависимости ячеек, как в настоящих электронных таблицах. Библиотека Spreadsheet была создана для демонстрации богатых возможностей библиотеки Streamulus и состоит исключительно из заголовочных файлов. В ней также используются библиотеки из коллекции Boost. Более подробную информацию читатель может найти по адресу <https://github.com/iritkatriel/spreadsheet>.

Разберём пример программы, прилагающийся к библиотеке Spreadsheet и включённый в репозиторий с примерами кода к данной книге.

```
#include "spreadsheet.hpp"
#include <iostream>
int main (int argc, const char * argv[]) {
    using namespace spreadsheet;
    Spreadsheet sheet;

    Cell<double> a = sheet.NewCell<double>();
    Cell<double> b = sheet.NewCell<double>();
    Cell<double> c = sheet.NewCell<double>();
    Cell<double> d = sheet.NewCell<double>();
    Cell<double> e = sheet.NewCell<double>();
    Cell<double> f = sheet.NewCell<double>();
```

В главной функции программы сначала создаётся объект «таблица» (Spreadsheet), который управляет жизнью и функционированием своих объектов-ячеек. Затем в контексте этой таблицы создаётся несколько объектов типа «ячейка» (Cell), содержащих действительные числа с двойной точностью. После создания ячеек можно заняться вписыванием в них как значений, так и выражений, содержащих ссылки на другие ячейки.

```
c.Set(SQRT(a()*a() + b()*b()));
a.Set(3.0);
b.Set(4.0);
d.Set(c()+b());
e.Set(d()+c());
```

После каждого изменения ячейки методом Set автоматически происходит распространение этого изменения по всем ячейкам, зависящим от неё. За организацию потоков данных отвечает библиотека Streamulus.

```
std::cout << " a=" << a.Value()
            << " b=" << b.Value()
            << " c=" << c.Value()
            << " d=" << d.Value()
            << " e=" << e.Value()
            << std::endl;
```



Этот фрагмент кода печатает на консоль получившиеся в итоге значения ячеек. Теперь можно назначить ячейкам новые выражения и значения, тем

самым перестроив граф потоков данных и заново запустив каскад изменений в зависимых ячейках:

```
c.Set(2*(a()+b()));
c.Set(4*(a()+b()));
c.Set(5*(a()+b()));
c.Set(6*(a()+b()));
c.Set(7*(a()+b()));
c.Set(8*(a()+b()));
c.Set(a());
```

```
std::cout << " a=" << a.Value()
           << " b=" << b.Value()
           << " c=" << c.Value()
           << " d=" << d.Value()
           << " e=" << e.Value()
           << std::endl;
```

```
std::cout << "Всего хорошего!\n";
return 0;
```

```
}
```

Исходный код библиотеки Spreadsheet стоит того, чтобы его внимательно изучить и разобраться в его внутреннем устройстве. Эта библиотека – превосходный пример применения библиотеки Streamulus надёжного и изящного программирования прикладных задач.

Библиотека RaftLib – ещё один инструмент обработки потоков данных

С библиотекой RaftLib стоит ознакомиться любому, кто интересуется параллельным программированием или программированием в модели потоков данных. Исходный код этой библиотеки находится в открытом доступе по адресу <https://github.com/RaftLib/RaftLib>. На странице библиотеки дано следующее описание:

i Библиотека RaftLib для языка C++ поддерживает параллельные вычисления над потоками данных. С помощью перегруженной операции сдвига вправо (наподобие операции записи в потоки стандартной библиотеки C++, которыми обычно пользуются для манипуляций со строками) можно связывать между собой параллельно работающие ядра¹. Библиотека RaftLib позволяет избежать явного управления потоками посредством механизмов pthread, std::thread, OpenMP и любых других библиотек для работы с потоками. Последние часто используются недостаточно правильно, что приводит к недетермини-

¹ Ядрами, согласно принятой в данной библиотеке терминологии, называются относительно независимые программные единицы, осуществляющие генерацию или обработку потоков данных. Каждое ядро отвечает за свою небольшую операцию над потоком данных. Таким образом, создать программу на основе библиотеки RaftLib – значит создать множество ядер и соединить их между собой каналами передачи данных. – *Прим. перев.*

рованному поведению программ. Модель, положенная в основу библиотеки RaftLib, позволяет использовать неблокирующий доступ по принципу очереди к каналам передачи данных, соединяющим ядра. Система в целом содержит многочисленные средства для автоматического распараллеливания вычислений, оптимизации и для удобства пользования, позволяющие относительно легко создавать высокопроизводительные прикладные программы.

В этой книге мы не будем освещать все подробности библиотеки RaftLib из-за недостатка места. Прекрасный доклад автора библиотеки Джонатана Берда (Jonathan Beard) можно найти по адресу <https://www.youtube.com/watch?v=liQ787fJgmU>. Рассмотрим небольшой пример кода, демонстрирующий библиотеку в действии.

```
#include <raft>
#include <raftio>
#include <cstdlib>
#include <string>

class hi : public raft::kernel
{
public:
    hi() : raft::kernel()
    {
        output.addPort<std::string>("0");
    }

    virtual raft::kstatus run()
    {
        output["0"].push(std::string("Hello World\n"));
        return raft::stop;
    }
};

int main()
{
    /** создать ядро для вывода строк на печать **/
    raft::print<std::string> p;
    /** создать ядро, генерирующее приветствие **/
    hi hello;
    /** создать карту соединений ядер **/
    raft::map m;
    /** связать два параллельно работающих ядра **/
    m += hello >> p;
    /** запустить систему **/
    m.exe();
    return EXIT_SUCCESS;
}
```

От программиста требуется создать свои ядра для вычислений, нужных для решения прикладной задачи, затем использовать перегруженную операцию >>

для организации потоков данных между ядрами. Чтобы узнать больше об этой замечательной библиотеке, читателю рекомендуется обратиться к документации (её можно найти на странице по приведённому выше адресу) и к примерам программ, включённым в состав библиотеки.

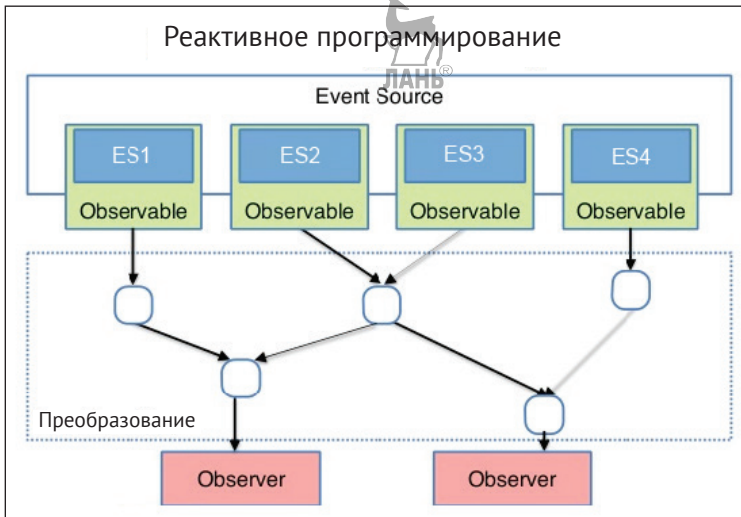
Потоки данных и реактивное программирование

Для нашего рассмотрения важно, что в модели реактивного программирования события образуются как потоки данных, распространяющиеся от обработчика к обработчику по определённому графу. Для этого объекты данных, характеризующие события, нужно накапливать в каких-то контейнерных структурах данных, из которых можно сделать потоки данных. Иногда, особенно если данных очень много, для сортировки и отбора событий применяются вероятностные и статистические методы. Сгенерированный поток данных можно фильтровать по некоторым критериям и подвергать преобразованиям на стороне источника, т. е. до того, как доставлять оповещения наблюдателям, ожидающим данных. От источников событий требуется, чтобы они следовали принципу «отправь и забудь» в отношении объектов-событий, чтобы избежать взаимного влияния источников и приёмников. За выбор моментов времени для доставки данных о событиях отвечает планировщик, который приводит в движение весь конвейер обработки событий асинхронным образом. Таким образом, ключевые элементы реактивного программирования – это:

- наблюдаемые источники данных, они же – потоки данных, в которых заинтересованы обработчики;
- наблюдатели, т. е. сущности, которые заинтересованы в данных от наблюдаемых источников и подписываются на уведомления от них;
- планировщик – сущность, которая решает, когда тот или иной поток данных может продвигаться дальше по графу обработчиков;
- операции в стиле функционального программирования для фильтрации и преобразования событий.

Если описать принцип действия реактивной системы предельно сжато, то планировщик как центральная часть этой системы приводит в действие наблюдаемый источник данных, управляет асинхронной фильтрацией и преобразованием событий и оповещает подписчиков о новых событиях, как показано на схеме.





Итоги

В этой главе мы рассмотрели программирование потоков событий. Подход к событиям как к потокам обладает рядом преимуществ по сравнению с традиционной моделью обработки событий. Открывал главу рассказ о библиотеке Streams и лежащей в её основе модели программирования. Далее рассматривалось несколько программ, помогающих ближе познакомиться с библиотекой и её семантикой. Библиотека Streams снабжена превосходной документацией, и читателю рекомендуется изучить её, чтобы побольше узнать о возможностях библиотеки. После библиотеки Streams было рассказано о библиотеке Streamulus, которая предоставляет программисту встраиваемый предметно-ориентированный язык для управления потоками данных. Проанализировано несколько примеров программ, прилагающихся к библиотеке. Также дана краткая характеристика библиотеки RaftLib – ещё одной удобной библиотеки для программирования потоков данных. Темой обработки потоков данных как модели программирования завершается изучение предварительного материала, необходимого для понимания реактивной модели программирования в целом и библиотеки RxCpp в частности. В следующей главе начнётся изучение библиотеки RxCpp и, на её примере, знакомство с разработкой реактивных систем.

Глава 7

.....

Знакомство с моделью маршрутов данных и библиотекой RxCpp



С этой главы начинается полное погружение в модель реактивного программирования. Предшествующие главы можно считать подготовительным материалом, приближающим к пониманию парадигмы реактивного программирования, в первую очередь функционального. Оглядываясь назад, можно вспомнить рассмотренные ранее темы:

- модели обработки событий, лежащие в основе различных платформ пользовательского графического интерфейса;
- новые средства, появившиеся в современном стандарте языка C++, включая элементы функционального программирования;
- средства поддержки многопоточного параллельного программирования, включённые в стандарт языка;
- модели неблокирующего программирования как шаг в направлении декларативного стиля;
- шаблоны разработки и их современные версии, модель наблюдаемых источников;
- модель потоков событий.

Парадигма **функционального реактивного программирования** (ФРП) зиждется на всех этих элементах, объединённых в строгую систему.

Говоря упрощённо, реактивное программирование – это не что иное, как программирование асинхронных потоков данных. Применяя к потокам различные операции, можно решать различные вычислительные задачи. Первый вопрос при разработке реактивной программы – как представить исходные данные в виде потоков, каким бы ни был источник этих данных. Потоки событий также называют наблюдаемыми источниками, а подписчиков называют ещё наблюдателями и обработчиками. Между источниками и наблюдателями располагаются операции над потоками: фильтры и преобразователи.

Поскольку, как правило, предполагается, что источник данных не претерпевает изменений, по мере того как данные проходят сквозь операции, одни и те же данные могут проходить от источника к наблюдателю различными путями. Неизменность источников открывает возможности для произвольного, асинхронного порядка выполнения операций, а координация работы системы во времени может возлагаться на отдельную программную сущность, называемую планировщиком. Таким образом, наблюдаемые источники, наблюдатели, операции над потоками и планировщики составляют скелет программы в модели ФРП.

В этой главе будут рассмотрены следующие темы:

- краткое введение в вычислительную парадигму, основанную на маршрутах данных;
- введение в библиотеку RxCpp и присущую ей модель программирования;
- простейшие примеры программ с использованием библиотеки RxCpp;
- операции над потоками в библиотеке RxCpp;
- графическое изображение реактивных операций с помощью цветных шариков;
- планировщики;
- аномалии операций разглаживания (flat) и сцепления (concat);
- различные операции над потоками.

ПАРАДИГМА МАРШРУТОВ ДАННЫХ¹

Обычно программисты при создании программ мыслят в терминах маршрутов управления. Это означает составлять программу из множества небольших операторов (присваиваний, их последовательного сочленения, условий и циклов) и подпрограмм (в том числе рекурсивных), работающих над состоянием памяти. Для управления ходом вычислений используются такие конструкции, как условный оператор if-else, операторы цикла while или for и рекурсия. Если программа, написанная в этой парадигме, предполагает параллельное выполнение нескольких потоков, доступ к общим данным требует особой тщательности и часто подвержен трудноуловимым ошибкам. Доступ к изменяемым данным, видимым из нескольких потоков, приходится окружать двоичными семафорами и другими примитивами синхронизации.

¹ В русском языке термин «поток» в контексте программирования обладает, по меньшей мере, тремя значениями: он соответствует английским терминам thread (поток выполнения в параллельном программировании, предмет главы 3), stream (поток как абстрагированный источник данных) и flow («русло», по которому протекают данные). При переводе работ, посвящённых одной узкой теме, омонимия не доставляет хлопот, однако в данной книге активно используются все три понятия. Потоки выполнения (thread) от потоков-источников (stream) легко отличить по контексту, а для термина «flow» пришлось выбрать альтернативный перевод: «маршрут». — *Прим. перев.*



С точки зрения компилятора результатом синтаксического разбора исходного текста становится построение абстрактного синтаксического дерева, которое используется в дальнейшем для проверки типов и генерации кода. В сущности, абстрактное синтаксическое дерево – это структурная модель программы, позволяющая проводить как анализ маршрутов данных, необходимый для оптимизации размещения данных в регистрах процессора, так и анализ хода выполнения, необходимый для оптимизации последовательности машинных команд. Хотя программист размышляет о своей программе в терминах хода выполнения, компилятор (по крайней мере, некоторая его часть) рассматривает программу также и сквозь призму маршрута данных. Таким образом, приходим к выводу, что граф маршрутизации данных скрыто присутствует в любой программе.

В парадигме маршрутов данных вычисление в явном виде построено на основе графа, вершины которого соответствуют вычислительным операциям, а рёбра – путям передачи данных между вершинами. Если на расположенные в вершинах графа вычислительные операции наложить дополнительные ограничения (например, потребовать, чтобы поступивший на вход операции оригинал данных сохранялся неизменным – вместо внесения изменений в этот объект операция может лишь создавать на его основе новые объекты), можно использовать все возможности параллельной обработки. Планировщик должен искать возможности для параллельного выполнения операций с помощью топологической сортировки графа обработки данных. При построении графа программист пользуется потоками данных (*stream*) для представления рёбер и потоковыми операциями для моделирования вершин. Это можно делать в декларативном стиле, в частности оформляя операции над потоками в виде лямбда-выражений, работающих исключительно со своими аргументами. В мире функционального программирования есть свой общепринятый набор простейших операций над потоками: например, операции *map* (поэлементное преобразование), *reduce* (свёртка по операции), *filter* (отбор элементов, удовлетворяющих условию), *take* (взятие заданного числа элементов). Системы, реализующие данную вычислительную модель, опираются на важное требование – данные должны быть представлены в виде потоков. На этой парадигме, в частности, основана библиотека TensorFlow, предназначенная для машинного обучения. Библиотеку RxCpp также можно считать средством организации вычислений через маршрутизацию данных, хотя в ней создание графа обработки данных выглядит не столь явным, как в библиотеке TensorFlow. Поскольку для функционального программирования характерен ленивый способ вычисления, граф обработки данных создаётся путём конструирования конвейера из потоков данных и асинхронных операций над потоками.

Знакомство с библиотекой RxCpp

В оставшейся части книги для написания реактивных программ будет использоваться библиотека RxCpp. Эта библиотека, предназначенная для програм-

мирования на языке C++, состоит исключительно из заголовочных файлов. Её исходный код можно загрузить из репозитория системы GitHub по адресу <https://github.com/ReactiveX/RxCpp>. В библиотеке RxCpp широко используются новшества современного стандарта языка, такие как классы для управления потоками, лямбда-выражения, композиции и преобразования функций и прочие, помогающие реализовать структуры реактивного программирования. Библиотека RxCpp выстроена по той же схеме, что и широко известные библиотеки Rx.net и RxJava.

Как и в других каркасах для реактивного программирования, здесь есть ряд основополагающих понятий, с которыми нужно хорошо разобраться, прежде чем браться за написание своей первой строчки кода. К ним относятся:

- наблюдаемые потоки данных;
- наблюдатели (или подписчики);
- операции над потоками;
- планировщики.

В библиотеке RxCpp большая часть вычислений выполняется посредством наблюдаемых потоков данных. Библиотека предоставляет множество средств для создания наблюдаемых потоков на основе разнообразных источников данных. В роли источников могут выступать диапазоны значений, контейнеры библиотеки STL и многие другие сущности. Между наблюдаемыми потоками и присоединёнными к ним приёмниками данных (наблюдателями) можно размещать операции. Поскольку функциональный стиль программирования поддерживает композицию функций, между наблюдаемым потоком и подписчиком можно поместить сколь угодно длинную цепочку операций, которая при этом выглядит как единая операция. Используемый в библиотеке планировщик заботится о том, что всякий раз, когда данные появляются в каком-то из наблюдаемых потоков, они немедленно будут пропущены через нужные операции, а подписчикам будет разослано оповещение (если после всех преобразований и фильтров по-прежнему есть, о чём оповещать). Работа наблюдателя начинается лишь тогда, когда с приходом оповещения вызывается нужный метод-обработчик (в частности, лямбда-выражение). Таким образом, наблюдатели могут заниматься исключительно основной задачей, за которую они и отвечают.

Библиотека RxCpp и её модель программирования

В этом разделе мы создадим несколько программ, которые помогут читателю понять модель программирования, лежащую в основе библиотеки RxCpp. Цель этих примеров – проиллюстрировать основные понятия реактивного программирования, поэтому программы будут довольно очевидными. Код примеров вполне достаточен, для того чтобы программисты включали его в свои реальные проекты с минимальными изменениями. В приведённых ниже примерах источники данных будут основываться на диапазонах значений, контейнерах библиотеки STL и др.

Простой пример взаимодействия источника с наблюдателем

Напишем программу, которая поможет понять модель программирования, заложенную в библиотеке RxCpp. В этой программе будет один поток данных в качестве наблюдаемого источника и один подписанный на него наблюдатель. Программа будет генерировать последовательность чисел от 1 до 12 с помощью специального класса `range`. После создания диапазона значений и надстроенного над ним наблюдаемого источника данных к последнему присоединяется подписчик. В результате своего выполнения эта программа напечатает на консоль последовательность чисел с дополнительным текстом.

```
#include "rxcpp/rx.hpp"
#include <iostream>

int main() {
    // создать поток чисел
    auto observable = rxcpp::observable<>::range(1, 12);

    // подписка на события OnNext и OnCompleted
    observable.subscribe(
        [](int v){ printf("OnNext: %d\n", v); },
        [](){ printf("OnCompleted\n"); });

    return 0;
}
```

Приведённая выше программа напечатает на консоль по одному сообщению для каждого числа и затем ещё одно сообщение «OnCompleted». Этот пример демонстрирует, как создавать наблюдаемый поток и как с помощью метода `subscribe` присоединять к нему наблюдателей.

Фильтрация и преобразование потоков данных

Следующий пример поможет читателю понять, как работают операции `filter` и `map` над потоками. Метод `filter` вычисляет предикат на каждом элементе входного потока, и если при этом получается значение «истина», элемент передаётся в выходной поток. Метод `map` применяет функцию-преобразователь к каждому элементу входного потока и полученные при этом значения отправляет в выходной поток.

```
#include "rxcpp/rx.hpp"
#include <iostream>

int main() {
    auto values = rxcpp::observable<>::range(1, 12)
        .filter([](int v){ return v % 2 == 0; })
        .map([](int x) { return x*x; });
    values.subscribe(
        [](int v){ printf("OnNext: %d\n", v); },
        [](){ printf("OnCompleted\n"); });
    return 0;
}
```

Эта программа генерирует поток целых чисел и пропускает этот поток через фильтр: операция `filter` отбирает из потока только чётные числа. Получившийся в результате фильтрации поток подаётся на преобразователь `map`, который возводит в квадрат каждый его элемент. В завершение поток результатов печатается на консоль.

Создание потока из контейнера

Хотя модель реактивного программирования в первую очередь предназначена для обработки данных, поступающих в непредсказуемые моменты на протяжении длительных промежутков времени, реактивный поток данных можно построить и из контейнера стандартного типа. Это может пригодиться при интеграции имеющегося кода, основанного на библиотеке STL, в реактивную систему. Для преобразования контейнера в поток служит метод `iterate`.

```
#include "rxcpp/rx.hpp"
#include <iostream>
#include <array>

int main() {
    std::array<int, 3> a={{1, 2, 3}};
    auto values = rxcpp::observable<>::iterate(a);
    values.subscribe(
        [](int v){ printf("OnNext: %d\n", v); },
        [ ](){ printf("OnCompleted\n"); });
    return 0;
}
```

СОЗДАНИЕ СОБСТВЕННЫХ НАБЛЮДАЕМЫХ ИСТОЧНИКОВ

В разобранных ранее примерах программ наблюдаемые потоки данных создавались либо из диапазона значений, либо из контейнера, заранее наполненного значениями. Теперь нужно рассмотреть, как создать свой наблюдаемый поток данных с нуля. Или почти с нуля.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
int main() {
    auto ints = rxcpp::observable<>::create<int>(
        [](rxcpp::subscriber<int> s){
            s.on_next(1);
            s.on_next(4);
            s.on_next(9);
            s.on_completed();
        });
    ints.subscribe(
        [](int v){printf("OnNext: %d\n", v);},
        [ ](){printf("OnCompleted\n");});
    return 0;
}
```



В этом примере объект `ints` представляет собой наблюдаемый источник данных целого типа, к которому могут подключаться наблюдатели или подписчики. Поведение источника `ints` задано в коде следующим образом: каков бы ни был подписчик, сообщить ему (путём вызова его метода `on_next`) три числа, затем (посредством метода `on_completed`) известить об окончании потока.

Конкатенация потоков

Два имеющихся потока данных можно сцепить между собой, образовав новый поток, что может быть удобно в некоторых случаях. Разберём, как работает конкатенация, с помощью примера:

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>

int main() {
    auto o1 = rxcpp::observable<>::range(1, 3);
    auto o2 = rxcpp::observable<>::range(4, 6);
    auto values = o1.concat(o2);
    values.subscribe(
        [](int v){ printf("OnNext: %d\n", v); },
        []() { printf("OnCompleted\n"); });
    return 0;
}
```



Операция `concat` присоединяет один поток к концу другого, сохраняя порядок элементов в каждом из них.



Отписка от потока данных

Следующий пример показывает, как отсоединить наблюдателя от наблюдаемого источника, тем самым прекратив отсылку ему оповещений от данного источника. В этой программе показан лишь один из возможных способов, за более подробными сведениями следует обратиться к документации.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>

int main() {
    auto subs = rxcpp::composite_subscription();
    auto values = rxcpp::observable<>::range(1, 10);
    values.subscribe(
        subs,
        [&subs](int v){
            printf("OnNext: %d\n", v);
            if (v == 6)
                subs.unsubscribe(); // отписаться
        },
        []() { printf("OnCompleted\n"); });
    return 0;
}
```

ВИЗУАЛЬНОЕ ПРЕДСТАВЛЕНИЕ ПОТОКОВ ДАННЫХ

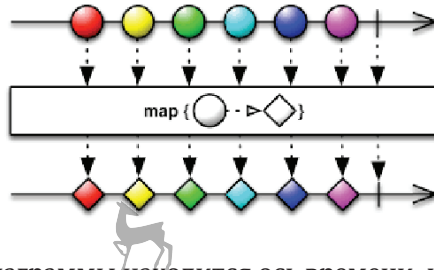
Реактивные потоки данных непросто изобразить графически, так как данные проходят по ним асинхронно. И всё же разработчикам реактивных систем удалось придумать подходящий способ визуализации потоков данных – диаграммы из разноцветных шариков¹. Начнём с небольшой программы, основанной на операции `map`.

```
#include "rxcpp/rx.hpp"
#include <iostream>

int main() {
    auto values = rxcpp::observable<>::range(1, 10)
        .map([](int x) { return x*x; });
    values.subscribe(
        [](int v){ printf("OnNext: %d\n", v); },
        [ ](){ printf("OnCompleted\n"); });
    return 0;
}
```



Вместо того чтобы долго описывать правила рисования диаграмм из разноцветных шариков, просто покажем пример диаграммы, изображающей работу операции `map`.



В верхней части диаграммы находится ось времени, на которой расположены элементы исходного потока данных. Ниже изображена операция, которая всякому шарообразному элементу данных ставит в соответствие ромбовидный. В самом низу расположена ось времени, представляющая собой как бы проекцию верхнего потока данных сквозь операцию.


ОПЕРАЦИИ НАД ПОТОКАМИ ДАННЫХ

Одно из главных преимуществ потоко-ориентированного подхода к обработке данных состоит в том, что к потокам можно применять преобразования в стиле функционального программирования. Если воспользоваться терминологи-

¹ В англоязычной литературе – marble diagrams. Слово «marble» (в прямом значении – мрамор) в данном случае означает шарик из цветного стекла с прожилками. – Прим. перев.

ей библиотеки RxCpp, обработка осуществляется посредством операций. Под операциями понимают не что иное, как фильтрацию, отображение, агрегацию и свёртку потоков. В предыдущих разделах было показано на примерах, как работают операции `map`, `filter` и `take`.

Операция `average`


Операция `average` вычисляет среднее арифметическое значений, полученных от источника данных. Поддерживаются и другие статистические операции, на-


- `min` (наименьшее значение);
- `max` (наибольшее значение);
- `count` (число элементов);
- `sum` (сумма значений).

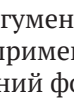
Ниже показан пример кода, иллюстрирующий применение операции `average`. Эта же схема подходит и для прочих операций из приведённого выше списка.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>

int main() {
    auto values = rxcpp::observable<>::range(1, 20).average();
    values.subscribe(
        [](double v){ printf("Среднее: %lf\n", v); },
        [ ](){ printf("OnCompleted\n"); });
    return 0;
}
```

Данная программа выведет на печать одно значение – среднее арифметическое чисел от 1 до 19.


Операция `scan`

Данная операция применяет функцию двух аргументов к очередному элементу входного потока и предыдущему результату применения этой функции, запо-


```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>

int main() {
    int count = 0;
    auto values = rxcpp::observable<>::range(1, 20).
        scan(
            0,
            [&count](int seed, int v){
                count++;
            }
        );
}
```

```

        return seed + v;
    });
    values.subscribe(
        [&](int v){ printf("Среднее: %f\n", (double)v/count); },
        [](){ printf("OnCompleted\n"); });
    return 0;
}

```



Эта программа выводит на экран последовательность из девятнадцати бегущих средних: для каждого целого числа из диапазона от 1 до 19 выводится среднее арифметическое чисел от 1 до этого числа.

Соединение операций в конвейер

Библиотека RxCpp позволяет соединять потоковые операции между собой, строя сколь угодно длинные цепочки. Для этого служит операция композиции (`|>`), благодаря ей программисты могут пользоваться привычным синтаксисом командной оболочки Unix. Использование этой операции позволяет писать код так, чтобы его смысл был понятен всякому. В следующем примере показана композиция из двух операций: генерации диапазона и поэлементного отображения значений.

```

#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
namespace Rx {
using namespace rxcpp;
using namespace rxcpp::sources;
using namespace rxcpp::operators;
using namespace rxcpp::util;
}
using namespace Rx;
#include <iostream>

int main() {
    auto ints = rxcpp::observable<>::range(1,10) |
        map( [] ( int n ) {return n*n; });

    ints.subscribe(
        [](int v){ printf("OnNext: %d\n", v); },
        [](){ printf("OnCompleted\n"); });
    return 0;
}

```



Работа с планировщиками

Из предыдущих разделов читатель узнал многое о наблюдаемых потоках данных, операциях над потоками и наблюдателях. В частности, о том, что между наблюдаемым источником и наблюдателем можно поместить реактивные операции для преобразования или фильтрации потоков. Парадигма функционального программирования велит писать чистые функции (т. е. функции без

побочных эффектов и глобального изменяемого состояния), следствием чего становится возможность обрабатывать данные в произвольном порядке. Порядок, в котором выполняются операции, не важен, если гарантируется, что данные на входе операций не могут быть изменены. Поскольку типичная реактивная программа состоит из многих наблюдателей и источников, управление очередностью операций можно поручить отдельному модулю – планировщику. Библиотека RxCpp по умолчанию ставит все операции в тот поток выполнения, из которого вызывался метод `subscribe` для подписки на события. Библиотека также позволяет указать другой поток выполнения, для этого служат методы `observe_on` и `subscribe_on`. Кроме того, некоторые операции над наблюдаемыми источниками принимают планировщик в качестве одного из своих аргументов – если эти операции могут выполняться в потоке, которым управляет планировщик.

В библиотеке RxCpp поддерживаются два вида планировщиков:

- непосредственный планировщик `ImmediateScheduler`;
- планировщик с циклом обработки событий `ImmediateScheduler`.

По умолчанию библиотека RxCpp однопоточна. Однако программист может по своему желанию настроить многопоточный режим работы. Основы управления планированием операций иллюстрирует следующий пример.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>
#include <thread>

int main(){
    // Генерировать последовательность значений
    auto values = rxcpp::observable<>::range(1,4).
        map([](int v){ return v*v; });
    // показать дескриптор главного потока
    std::cout << "Главный поток => "
        << std::this_thread::get_id()
        << std::endl;
    // наблюдатель в другом потоке выполнения
    values.observe_on(rxcpp::synchronize_new_thread()).
        as_blocking().subscribe(
            [](int v) {
                std::cout << "Поток наблюдателя => "
                    << std::this_thread::get_id()
                    << " " << v << std::endl;
            },
            [](){ std::cout << "OnCompleted" << std::endl; });

    // показать дескриптор главного потока
    std::cout << "Главный поток => "
        << std::this_thread::get_id()
        << std::endl;
    return 0;
}
```


Чтобы видеть, в каком потоке выполняется та или иная операция, программа выводит на консоль числовой идентификатор, который есть у каждого потока согласно стандарту языка. Эта программа выведет на консоль следующий текст, из которого очевидно, что подписчик работает в потоке, отличном от главного потока программы:

```
Главный поток => 1
Поток наблюдателя => 2 1
Поток наблюдателя => 2 4
Поток наблюдателя => 2 9
Поток наблюдателя => 2 16
OnCompleted
Главный поток => 1
```

Следующая программа иллюстрирует использование метода `subscribe_on`. В поведении методов `observe_on` и `subscribe_on` есть тонкое различие, которое разберём в следующей главе. Следующий пример кода призван продемонстрировать декларативные средства управления планировщиком.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>
#include <thread>
#include <mutex>
// Для глобальной синхронизации вывода.
std::mutex console_mutex;

// вывести идентификатор текущего потока
void CTDetails() {
    console_mutex.lock();
    std::cout << "Поток => "
              << std::this_thread::get_id()
              << std::endl;
    console_mutex.unlock();
}

// отдать управление другим потокам
void Yield( bool y ) {
    if (y) { std::this_thread::yield(); }
}

int main(){
    auto threads = rxcpp::observe_on_event_loop();
    auto values = rxcpp::observable<>::range(1);

    // запланировать на выполнение в другом потоке
    auto s1 = values.
        subscribe_on(threads).
        map([](int prime) {
            CTDetails();
            Yield(true);
            return std::make_tuple("1:", prime);
```



```

    });
    // запланировать в ещё одном потоке
    auto s2 = values.
        subscribe_on(threads).
        map([](int prime) {
            CTDetails();
            Yield(true);
            return std::make_tuple("2:", prime);
        });

    s1.merge(s2).
        take(6).as_blocking().
        subscribe(rxcpp::util::apply_to(
            [](const char* s, int p) {
                CTDetails();
                console_mutex.lock();
                printf("%s %d\n", s, p);
                console_mutex.unlock();
            }));
    return 0;
}

```



На один источник данных `values` подписаны два наблюдателя, причём каждый ставится на выполнение в своём потоке. В результате выполнения этой программы на консоль выводится текст, свидетельствующий о том, что подписчики работают параллельно.

Сага о двух операциях: как разглаживать потоки потоков

Часто источником недоразумений для программистов оказываются операции `flat_map` и `concat_map`. Различие между ними впрямь довольно тонко, его мы и разберём в этом разделе. Сначала рассмотрим пример программы, в которой использована операция `flat_map`.

```

#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>
namespace rxu = rxcpp::util;
#include <array>

int main() {
    std::array<std::string, 4> a =
        {"Praseed", "Peter", "Sanjay", "Raju"};
    auto values = rxcpp::observable<>::iterate(a).flat_map(
        [](std::string v) {
            std::array<std::string, 3> salutation =
                {"Mr.", "Monsieur", "Sri"};
            return rxcpp::observable<>::iterate(salutation);
        },
        [](std::string f, std::string s) { return s + " " + f; });
    values.subscribe(

```



```

    [] (std::string f) { std::cout << f << std::endl; },
    []() { std::cout << "Здравствуй, мир" << std::endl; });
return 0;
}

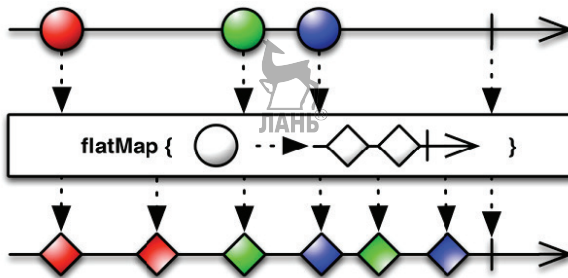
```

Эта программа «перемножает» содержимое двух контейнеров и выводит результат в произвольном порядке. Ниже приведён результат её работы. Одна из выгод от этой операции состоит в возможности дополнительно обрабатывать поток после применения операции преобразования элементов.

Mr. Praseed
 Monsieur Praseed
 Mr. Peter
 Sri Praseed
 Monsieur Peter
 Mr. Sanjay
 Sri Peter
 Monsieur Sanjay
 Mr. Raju
 Sri Sanjay
 Monsieur Raju
 Sri Raju
 Здравствуй, мир



На следующей диаграмме изображен принцип работы этой операции. Операция `flat_map` к каждому элементу потока данных применяет лямбда-функцию, которая одному входному значению ставит в соответствие целый поток результатов. Затем производится слияние полученных потоков в один выходной поток. На диаграмме показано, что красный шарик преобразуется в два ромба того же цвета, а ромбы, полученные из зелёного и синего шаров, оказываются перемешанными в потоке результатов.



Теперь рассмотрим программный код с операцией `concat_map`. Этот код почти неотличим от предыдущего. Единственное изменение состоит в замене операции `flat_map` на `concat_map`.

```

#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"

```

```

#include <iostream>
namespace rxu = rxcpp::util;
#include <array>

int main() {
    std::array<std::string, 4> a =
        {{ "Praseed", "Peter", "Sanjay", "Raju" }};
    auto values = rxcpp::observable<>::iterate(a).concat_map(
        [] (std::string v) {
            std::array<std::string, 3> salutation =
                {{ "Mr.", "Monsieur", "Sri" }};
            return rxcpp::observable<>::iterate(salutation);
        },
        [] (std::string f, std::string s) { return s + " " + f; });
    values.subscribe(
        [] (std::string f) { std::cout << f << std::endl; },
        [] () { std::cout << "Здравствуй, мир" << std::endl; });
    return 0;
}

```

Вот результат работы этой программы. Легко видеть, что данные в потоке результатов не перемешаны, а расположены в строгом порядке.

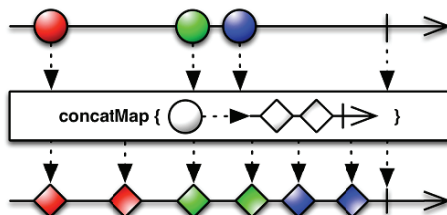
```

Mr. Praseed
Monsieur Praseed
Sri Praseed
Mr. Peter
Monsieur Peter
Sri Peter
Mr. Sanjay
Monsieur Sanjay
Sri Sanjay
Mr. Raju
Monsieur Raju
Sri Raju
Здравствуй, мир

```



На следующей диаграмме показано, как работает операция `concat_map`. В отличие от предыдущего случая, обработка теперь синхронизирована: сначала в потоке результатов стоят все красные элементы, затем зелёные, после них – синие.



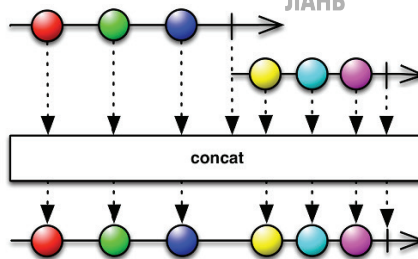
Итак, операция `flat_map` вырабатывает результаты в перемешанном порядке, тогда как операция `concat_map` строит выходной поток в точно том же порядке, что ожидалось. В чём же настоящее различие между ними? Чтобы сделать разницу более очевидной, рассмотрим две операции: `concat` и `merge`. А именно разберёмся, каким образом работает конкатенация (склеивание) потоков. Обычно эта операция присоединяет всё содержимое второго потока после элементов первого потока, сохраняя исходный порядок элементов, как явствует из следующего примера.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>
#include <array>

int main() {
    auto o1 = rxcpp::observable<>::range(1, 3);
    auto o2 = rxcpp::observable<>::range(4, 6);
    auto values = o1.concat(o2);

    values.subscribe(
        [](int v) { printf("OnNext: %d\n", v); },
        []() { printf("OnCompleted\n"); });
    return 0;
}
```

На следующей диаграмме показано, что происходит, когда к двум потокам данных применяется операция `concat`. Новый поток создаётся путём прибавления содержимого второго потока к первому. Порядок элементов при этом сохраняется.



Теперь посмотрим, что получается, если к потокам применить операцию `merge`. Следующая программа демонстрирует эту операцию в действии.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>
namespace rxu=rxcpp::util;
#include <array>
```

```

int main() {
    auto o1 = rxcpp::observable<>
        ::timer(std::chrono::milliseconds(15))
        .map([](int) { return 1; });
    auto o2 = rxcpp::observable<>::error<int>(
        std::runtime_error("Error from source\n"));

    auto o3 = rxcpp::observable<>
        ::timer(std::chrono::milliseconds(5))
        .map([](int) { return 3; });

    auto base = rxcpp::observable<>
        ::from(o1.as_dynamic(), o2, o3);

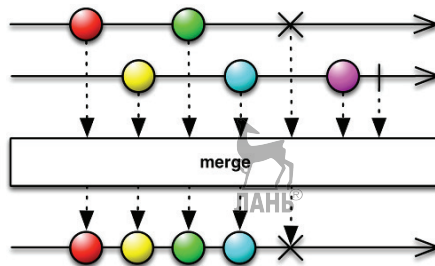
    auto values = base.merge();

    values.subscribe(
        [](int v){ printf("OnNext: %d\n", v); },
        [](std::exception_ptr eptr) {
            printf("OnError %s\n", rxu::what(eptr).c_str());
        },
        [ ](){ printf("OnCompleted\n"); });

    return 0;
}

```

На следующей диаграмме показано, как работает операция `merge` над двумя потоками данных. Содержимое выходного потока представляет собой чередование элементов двух входных потоков в произвольном порядке.



Теперь можно описать различие между операциями `flat_map` и `concat_map`: оно состоит в том, каким способом комбинируются потоки значений. Операция `flat_map` реализована на основе операции `merge`, тогда как операция `concat_map` основана на операции `concat`. В случае операции `merge` порядок элементов в выходном потоке не имеет значения, а операция `concat` всегда помещает элементы второго входного потока после элементов первого – отсюда и различный порядок результатов у программ, иллюстрирующих работу операций `flat_map` и `concat_map`.

Прочие важные операции

Из предыдущих разделов должна уже быть понятна суть реактивной модели программирования, так как разобраны все основные темы: потоки данных, наблюдатели, операции над потоками и планировщики. В библиотеке RxCpp поддерживается ещё ряд операций, о которых стоит знать, чтобы лучше выражать логику работы программ. В этом разделе будут рассмотрены операции `tap` и `buffer`. Начнём с операции `tap`, которая позволяет заглянуть в содержимое потока данных.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>
```



```
int main() {
    // создать поток с помощью операции map
    auto ints = rxcpp::observable<>::range(1,3).
        map([] (int n) { return n*n; });
    // применить операцию tap для трассировки потока
    auto values = ints.tap(
        [](int v) { printf("Tap -      OnNext: %d\n", v); },
        []() { printf("Tap -      OnCompleted\n"); });
    // выполнить действия с потоком
    values.subscribe(
        [](int v){ printf("Subscribe - OnNext: %d\n", v); },
        [](){ printf("Subscribe - OnCompleted\n"); });
    return 0;
}
```

Теперь рассмотрим операцию `defer`. В качестве аргумента она принимает фабрику потоков – то есть функцию (или функциональный объект), которая создаёт поток данных только тогда, когда какой-то наблюдатель подписывается на эти данные. Работу этой функции иллюстрирует следующая программа.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>
```



```
int main() {
    auto observable_factory = [] () {
        return rxcpp::observable<>::range(1,3)
            .map([] (int n) { return n*n; });
    };
    auto ints = rxcpp::observable<>::defer(observable_factory);
    ints.subscribe(
        [](int v) { printf("OnNext: %d\n", v); },
        [] () { printf("OnCompleted\n"); });

    ints.subscribe(
        [](int v) { printf("2nd OnNext: %d\n", v); },
        [] () { printf("2nd OnCompleted\n"); });
}
```

Метод `buffer`, поддерживаемый классами потоков данных, принимает в качестве аргумента целое число и создаёт новый поток данных, чьи элементы – это контейнеры значений, взятых из исходного потока, причём размер этих контейнеров ограничен заданным числом. Это позволяет наблюдателям обрабатывать элементы не по одному, а «пачками», как показано в следующем примере.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>

int main() {
    auto values = rxcpp::observable<>::range(1, 10).buffer(2);
    values.subscribe(
        [](std::vector<int> v){
            printf("OnNext:{"");
            std::for_each(
                v.begin(),
                v.end(),
                [](int a){ printf(" %d", a); });
            printf("}\n");
        },
        [](){ printf("OnCompleted\n"); });
    return 0;
}
```

Операция `timer` принимает в качестве аргумента интервал времени и, возможно, необязательный аргумент – планировщик и создаёт поток данных. Библиотека содержит несколько вариантов этой функции. В следующем примере показан один из них.

```
#include "rxcpp/rx.hpp"
#include "rxcpp/rx-test.hpp"
#include <iostream>

int main() {
    auto scheduler = rxcpp::observe_on_new_thread();
    auto period = std::chrono::milliseconds(1);
    auto values = rxcpp::observable<>::timer(period, scheduler)
        .finally([](){ printf("Конец\n"); });

    values.as_blocking().subscribe(
        [](int v){ printf("OnNext: %d\n", v); },
        [](){ printf("OnCompleted\n"); });
    return 0;
}
```

БЕГЛЫЙ ВЗГЛЯД НА ЕЩЁ НЕ ИЗУЧЕННОЕ

Реактивную модель программирования можно считать результатом слияния следующих компонентов:

- вычисления на потоках данных;

- декларативный стиль программирования;
- функциональный стиль программирования;
- параллельная обработка;
- программирование в терминах потоков событий;
- применение идиом и шаблонов проектирования.

Чтобы понять данный предмет во всей его полноте, нужно много практиковаться с описанной здесь моделью программирования. Поначалу упражнения могут показаться не слишком осмысленными, но рано или поздно произойдёт качественный скачок, после которого сложится понимание всей модели целиком. До сих пор были рассмотрены следующие вопросы:

- наблюдаемые источники (потоки) данных и наблюдатели;
- базовые и вспомогательные операции над потоками данных;
- средства планирования операций: базовые и средней сложности.

Это лишь начало, для по-настоящему глубокого знакомства с реактивной моделью программирования предстоит изучить ещё множество вопросов:

- горячие и холодные источники данных (глава 9);
- подробности реактивных компонентов (глава 9);
- усложнённые средства управления планировщиками (глава 9);
- особенности программирования графических интерфейсов (глава 9);
- усложнённые операции над потоками данных (глава 9);
- реактивные шаблоны проектирования (глава 10);
- обеспечение надёжности программ (глава 12).

Итоги

В этой главе освещен довольно обширный круг вопросов, относящихся как к основаниям реактивной модели программирования вообще, так и к библиотеке RxCpp в частности. Открывал главу концептуальный обзор вычислительной парадигмы, основанной на маршрутизации данных, следующие разделы были посвящены написанию простейших реактивных программ. После знакомства с графическим языком, позволяющим описывать функционирование реактивных систем, было рассказано об основных операциях, поддерживаемых библиотекой RxCpp. Затем было введено важное для дальнейшего изучения понятие планировщика. Завершал главу рассказ о различии между операциями `flat_map` и `concat_map` и краткий обзор ряда других полезных операций над потоками данных. В следующей главе будет рассказано о горячих и холодных источниках данных, об усложнённых средствах планирования операций и о многом другом, не затронутом в этой главе.

Ключевые элементы библиотеки RxCpp

В предыдущей главе началось знакомство читателя с библиотекой RxCpp и её программной моделью. Ряд примеров был призван пояснить, как работает эта библиотека. Описаны наиболее важные её части. В этой главе читателю предлагается более глубоко изучить основные механизмы библиотеки RxCpp и реактивной модели программирования в целом, включая следующие вопросы:

- наблюдаемые источники данных (Observable);
- наблюдатели и их разновидность – подписчики;
- темы (Subject) как дальнейшее развитие идеи наблюдаемых источников;
- планировщики;
- операции над потоками.

В сущности, главные принципы реактивного программирования сводятся к следующим.

- Наблюдаемые источники – это потоки данных, на оповещения от которых могут подписываться наблюдатели.
- Тема – это комбинация наблюдаемых источников данных и наблюдателей.
- Планировщики запускают на выполнение действия, связанные с операциями, и заставляют данные продвигаться от источников к наблюдателям.
- Операции над потоками – это функции, которые преобразовывают одни наблюдаемые источники данных в другие наблюдаемые источники.

Наблюдаемые источники данных

В предыдущей главе были показаны многочисленные примеры наблюдаемых источников данных и присоединённых к ним подписчиков. Приоткроем теперь их внутреннее устройство: во всех разобранных ранее примерах наблюдаемый источник данных (observable) создавал вспомогательный объект – производитель данных (producer). Задача производителя – производить поток

событий. При этом задача наблюдаемого источника самого по себе состоит в том, чтобы подключить подписчиков к производителю. Прежде чем двигаться дальше, остановимся ещё раз на анатомии наблюдаемого источника и его функционировании.

- Наблюдаемый источник данных (observable) ведёт себя подобно функции, которая принимает объект-наблюдатель (observer) в качестве аргумента и возвращает новую функцию.
- Наблюдаемый источник данных присоединяет объект-наблюдатель к объекту-производителю; сам же производитель непосредственно для наблюдателя невидим.
- Производитель вырабатывает данные для наблюдаемого источника.
- Наблюдатель (observer) – это объект, обладающий методами `on_next`, `on_error` и `on_completed`.

Что такое объект-производитель

Из производителя (producer) появляются данные наблюдаемого источника. В роли производителей могут выступать окна, таймеры, соединения Web-Socket, древовидные документы, итераторы по коллекциям или контейнерам и многие другие сущности. Производителем способно быть всё, из чего можно получать данные, которые в дальнейшем предполагается передавать наблюдателю через метод `on_next`.

Горячие и холодные источники данных

В большинстве примеров из предыдущей главы объекты-производители создавались в процессе работы операций над наблюдаемыми источниками. Однако производитель может быть создан и отдельно от такой операции, тогда последней передаётся лишь ссылка на уже имеющийся объект-производитель. Наблюдаемый источник данных (observable), который содержит лишь ссылку на объект-производитель, называется *горячим*. Напротив, наблюдаемый источник, который сам создаёт себе производителя и владеет им, называется *холодным*. Чтобы пояснить эти понятия, покажем сначала пример холодного источника.

```
#include <rxcpp/rx.hpp>
#include <memory>

int main(int argc, char *argv[]) {
    // планировщик
    auto eventloop = rxcpp::observe_on_event_loop();
    // холодный источник
    auto values = rxcpp::observable<>
        ::interval(std::chrono::seconds(2)).take(2);
```

Функция `interval` возвращает холодный источник данных, поскольку именно эта функция создаёт объект-производитель, генерирующий поток событий.

Холодный источник начинает генерировать данные только тогда, когда к нему подключается подписчик или наблюдатель. Даже если наблюдатель подключится с задержкой, это не повлияет на результат – он всё равно получит от источника все данные. Это иллюстрирует следующий фрагмент кода:

```
// подписаться дважды
values.subscribe_on(eventloop).subscribe(
    [](int v){ printf("[1] onNext: %d\n", v); },
    [](){ printf("[1] onCompleted\n"); });
values.subscribe_on(eventloop).subscribe(
    [](int v){ printf("[2] onNext: %d\n", v); },
    [](){ printf("[2] onCompleted\n"); });

// пустая блокирующая подписка, чтобы увидеть результат
values.as_blocking().subscribe();

// подождать две секунды
rxcpp::observable<>::timer(std::chrono::milliseconds(2000))
    .subscribe([&](long){ });

return 0;
}
```

Результат работы этой программы показан ниже. Порядок сообщений может различаться от запуска к запуску, поскольку порядок выполнения операций в параллельных потоках недетерминирован. Однако задержка между первой и второй подписками не может привести к потере данных.

```
[1] onNext: 1
[2] onNext: 1
[2] onNext: 2
[1] onNext: 2
[2] onCompleted
[1] onCompleted
```

Горячие источники данных

Холодный источник данных можно превратить в горячий с помощью метода `publish`. Вследствие такого преобразования подписчики, подключившиеся к источнику слишком поздно, пропустят сообщения, отправленные источником до этого. Горячий источник генерирует данные независимо от того, подписаны ли на него хоть один наблюдатель. Эту особенность демонстрирует следующая программа:

```
#include <rxcpp/rx.hpp>
#include <memory>

int main(int argc, char *argv[]) {
    auto eventloop = rxcpp::observe_on_event_loop();
    // создать холодный источник и превратить его в горячий
    auto values = rxcpp::observable<>
```

```

        ::interval(std::chrono::seconds(2))
        .take(2)
        .publish();

// подписаться дважды
values.subscribe_on(eventloop).subscribe(
    [](int v){ printf("[1] onNext: %d\n", v); },
    [](){ printf("[1] onCompleted\n"); });

values.subscribe_on(eventloop).subscribe(
    [](int v){ printf("[2] onNext: %d\n", v); },
    [](){ printf("[2] onCompleted\n"); });

// запустить генерацию событий
values.connect();
// блокирующая подписка
values.as_blocking().subscribe();

// подождать две секунды
rxcpp::observable<>::timer(
    std::chrono::milliseconds(2000))
    .subscribe([&](long) { });

return 0;
}

```



Следующий пример показывает ещё один механизм, поддерживаемый библиотекой RxCpp, а именно метод `publish_synchronized`. С точки зрения программного интерфейса, отличие от предыдущего случая совсем невелико. Рассмотрим текст программы:

```

#include <rxcpp/rx.hpp>
#include <memory>

int main(int argc, char *argv[]) {
    auto eventloop = rxcpp::observe_on_event_loop();
    // создать холодный источник и превратить его в горячий
    auto values = rxcpp::observable<>
        ::interval(std::chrono::seconds(2))
        .take(5)
        .publish_synchronized(eventloop);

// подписаться дважды
values.subscribe(
    [](int v){ printf("[1] onNext: %d\n", v); },
    [](){ printf("[1] onCompleted\n"); });

values.subscribe(
    [](int v){ printf("[2] onNext: %d\n", v); },
    [](){ printf("[2] onCompleted\n"); });

// запустить генерацию событий

```



```

values.connect();
// блокирующая подписка
values.as_blocking().subscribe();

// подождать две секунды
rxcpp::observable<>::timer(
    std::chrono::milliseconds(2000))
    .subscribe([&](long){ });

return 0;
}

```



Программа напечатает на консоль следующий текст. Легко убедиться, что операции над потоком вполне синхронизированы, сообщения выводятся в строгом порядке.

```

[1] onNext: 1
[2] onNext: 1
[1] onNext: 2
[2] onNext: 2
[1] onNext: 3
[2] onNext: 3
[1] onNext: 4
[2] onNext: 4
[1] onNext: 5
[2] onNext: 5
[1] onCompleted
[2] onCompleted

```



Горячие источники данных и механизм повтора

Горячий источник испускает данные независимо от того, получают ли их подписчики. Иногда это бывает нежелательно. В реактивном программировании существует механизм, позволяющий запоминать невостребованные данные в буфере, чтобы опоздавшие подписчики получили о них оповещения. Чтобы сделать горячий источник буферизированным, следует воспользоваться методом `replay`. Разберём программу, которая демонстрирует в действии полезный приём работы с горячими источниками данных – повтор отложенных в буфере сообщений.

```

//----- ReplayAll.cpp
#include <rxcpp/rx.hpp>
#include <memory>

int main(int argc, char *argv[]) {
    auto values = rxcpp::observable<>::interval(
        std::chrono::milliseconds(50),
        rxcpp::observe_on_new_thread())
        .take(5).replay();

    // подписаться сразу

```

```

values.subscribe(
    [](long v){printf("[1] OnNext: %ld\n", v);},
    [](){printf("[1] OnCompleted\n");});
// запуск
values.connect();
// выдержать паузу перед подпиской
rxcpp::observable<>::timer(std::chrono::milliseconds(125))
    .subscribe([&](long) {
        values.as_blocking().subscribe(
            [](long v){ printf("[2] OnNext: %ld\n", v); },
            [](){ printf("[2] OnCompleted\n"); });
    });

// подождать две секунды
rxcpp::observable<>::timer(
    std::chrono::milliseconds(2000)).
    subscribe([&](long){ });

return 0;
}

```

Чтобы создавать реактивные программы, программисту нужно хорошо понимать семантическое различие между горячими и холодными источниками данных. В этом разделе затронуты лишь некоторые аспекты данного различия. Более подробные сведения можно найти в документации к библиотеке RxCpp и, шире, к системе ReactiveX. В сети имеется множество статей, посвящённых данной теме.

Наблюдатели и подписчики

Наблюдатель (observer) подписывается на оповещения от определённого наблюдаемого источника данных (observable). О наблюдателях речь шла в предыдущей главе. Теперь в центре нашего внимания будет подписчик – как комбинация наблюдателя и подписки. Подписчик обладает способностью отписываться от оповещений, тогда как наблюдатель в строгом смысле умеет лишь подписываться. Следующая программа поможет пояснить разницу между этими понятиями.

```

#include "rxcpp/rx.hpp"
int main() {
    // создать объект-подписку
    auto subscription = rxcpp::composite_subscription();
    // создать объект-подписчик
    auto subscriber = rxcpp::make_subscriber<int>(
        subscription,
        [&subscription] (int v) {
            printf("OnNext: --%d\n", v);
            if (v == 3)
                subscription.unsubscribe();
        },

```

```

    [](){ printf("OnCompleted\n"); });

    rxcpp::observable<>::create<int>(
        [] (rxcpp::subscriber<int> s) {
            for (int i = 0; i < 5; ++i) {
                if (!s.is_subscribed())
                    break;
                s.on_next(i);
            }
            s.on_completed();
        }).subscribe(subscriber);

    return 0;
}

```

Чтобы создавать сложные программы со множеством параллельных потоков и гибким поведением, возможность отказываться от подписки весьма удобна. Читателю стоит глубже изучить эту тему, обратившись к документации по библиотеке RxCpp.

ЕДИНСТВО НАБЛЮДАЕМОГО И НАБЛЮДАТЕЛЯ

Темой (Subject) называют программную сущность, которая является одновременно наблюдателем (observer) и наблюдаемым источником данных (observable). Такой объект помогает рассылать оповещения от одного источника множеству наблюдателей. С его помощью можно реализовать такие усложнённые техники, как буферизация данных. Объект-тему можно использовать для преобразования горячего источника данных в холодный. В библиотеке RxCpp реализованы четыре разновидности объектов-тем:

- SimpleSubject;
- BehaviorSubject;
- ReplaySubject;
- SynchronizeSubject.

Рассмотрим простую программу, в которой объект-тема подписывается на получение данных от источника и сам выступает в качестве источника для других наблюдателей.

```

#include <rxcpp/rx.hpp>
#include <memory>
int main() {
    // создать объект-тему
    rxcpp::subjects::subject<int> subject;

    // получить интерфейс источника
    auto observable = subject.get_observable();
    // подписаться дважды
    observable.subscribe([](int v) { printf("1---%d\n",v ); });
    observable.subscribe([](int v) { printf("2---%d\n",v ); });

    // получить интерфейс подписчика
}

```



```

auto subscriber = subject.get_subscriber();
// оповестить о нескольких значениях
subscriber.on_next(1);
subscriber.on_next(4);
subscriber.on_next(9);
subscriber.on_next(16);

// выждать две секунды
rxcpp::observable<>::timer(std::chrono::milliseconds(2000))
    .subscribe([](long){});
return 0;
}

```



Тема разновидности BehaviorSubject представляет собой разновидность объекта-темы, который хранит текущее (т. е. последнее сгенерированное) значение. Всякий новый подписчик немедленно получает оповещение о текущем значении. Во всех прочих отношениях этот объект ведёт себя так же, как и тема разновидности SimpleSubject. Темы вида BehaviorSubject называют также свойствами (property) и ячейками (cell). Это может быть полезно в случаях, когда определённая область памяти подвергается последовательным обновлениям, как при транзакциях. Следующая программа демонстрирует такой объект в действии.

```

#include <rxcpp/rx.hpp>
#include <memory>
int main() {
    rxcpp::subjects::behavior<int> behsubject(0);
    auto observable = behsubject.get_observable();
    observable.subscribe([](int v) { printf("1----%d\n", v ); });
    observable.subscribe([](int v) { printf("2----%d\n", v ); });
    auto subscriber = behsubject.get_subscriber();
    subscriber.on_next(1);
    subscriber.on_next(2);

    int n = behsubject.get_value();
    printf ("Последнее значение: %d\n", n);
    return 0;
}

```

Разновидность тем ReplaySubject обладает способностью хранить ранее сгенерированные значения. При создании такого объекта можно настроить, сколько именно последних значений он должен запоминать. Это очень удобно при работе с горячими источниками данных. Ниже показаны прототипы различных вариантов конструктора replay:


```

replay(
    Coordination cn,
    composite_subscription cs=composite_subscription());

replay(
    std::size_t count,

```





```

    Coordination cn,
    composite_subscription cs=composite_subscription());
replay(
    rxsc::scheduler::clock_type::duration period,
    Coordination cn,
    composite_subscription cs=composite_subscription());
replay(
    std::size_t count,
    rxsc::scheduler::clock_type::duration period,
    Coordination cn,
    composite_subscription cs=composite_subscription());

```

Ниже приведён пример, поясняющий семантику тем ReplaySubject.

```

#include <rxcpp/rx.hpp>
#include <memory>
int main(int argc, char *argv[]) {

    // создать объект-тему ReplaySubject
    rxcpp::subjects::replay<int, rxcpp::observe_on_one_worker>
        replay_subject(10, rxcpp::observe_on_new_thread());


    // получить интерфейс источника
    auto observable = replay_subject.get_observable();
    // подписаться
    observable.subscribe([](int v) { printf("1---%d\n", v ); });

    // получить интерфейс подписчика и оповестить о данных
    subscriber.on_next(1);
    subscriber.on_next(2);

    // добавить подписчика:
    // обычная тема потеряла бы старые значения,
    // но этот объект хранит их в буфере
    observable.subscribe([](int v) { printf("2---%d\n", v ); });

    // выждать две секунты
    rxcpp::observable<>::timer(std::chrono::milliseconds(2000))
        .subscribe([](long){ });
    return 0;
}

```



Таким образом, в этом разделе рассмотрены три варианта объектов-тем. Основное их предназначение состоит в том, чтобы через интерфейс подписчика объединять события, получаемые от различных источников, и через интерфейс источника предоставлять их группе других подписчиков. Темы типа SimpleSubject делают только это: служат одновременно источниками и подписчиками и работают как промежуточные узлы передачи потока данных. Темы типа BehaviorSubject позволяют отслеживать изменение какого-либо свойства или переменной на протяжении некоторого времени. Темы типа ReplaySubject помогают избежать потери данных вследствие поздней подписки. Наконец,

темы типа `SynchronizeSubject` обладают встроенными механизмами синхронизации, но их мы здесь рассматривать не будем.

ПЛАНИРОВЩИКИ

Библиотека `RxCpp` поддерживает декларативный механизм управления параллельными потоками выполнения благодаря входящей в её состав надёжной подсистеме – планировщику вычислений. Данные, поступающие от наблюдаемого источника, отправляются по графу распространения изменений множеством различных маршрутов. Давая подсказки конвейеру обработки данных, можно сделать так, чтобы вычислительные операции ставились на выполнение в один поток, в несколько параллельных потоков или в фоновый поток. Это позволяет выражать замысел программиста более отчётливо.

Декларативная модель планирования операций в библиотеке `RxCpp` возможна благодаря неизменности объектов данных, поступающих на вход вычислительных операций. Когда операция применяется к потоку данных, это означает, что на вход она принимает источник данных (`observable`), а результатом такого применения становится новый источник данных. Данные, приходящие в операцию, не модифицируются – вместо этого на выходе операции создаются новые объекты данных. Это обстоятельство позволяет обрабатывать элементы потока независимо друг от друга в параллельных потоках, не заботясь о синхронизации. Подсистема планировщика библиотеки `RxCpp` содержит следующие сущности:

- собственно планировщик (`scheduler`);
- рабочий поток (`worker`);
- координация (`coordination`);
- координатор (`coordinator`);
- единица планирования (`schedulable`);
- ось времени (`time line`);

Библиотека `RxCpp` версии 2 обязана своей архитектурой планировщика системе `RxJava`. В частности, отсюда позаимствована идиома рабочего потока. Работа планировщика основывается на следующих важных принципах:

- планировщик обладает осью времени;
- планировщик может создавать сколько угодно рабочих потоков, привязанных к этой оси времени;
- каждый рабочий поток содержит очередь единиц планирования;
- единица планирования обладает функцией (также называемой действием, `action`) и имеет определённое время жизни;
- координация работает как фабрика координаторов и обладает своим планировщиком;
- каждый координатор обладает своим рабочим потоком и выступает фабрикой для:
 - координированных единиц планирования;
 - координированных источников данных и подписчиков.

В примерах реактивных программ из предыдущих разделов на самом деле использовались планировщики, хотя их функционирование и сам факт существования не был виден сквозь программный интерфейс. Напишем теперь простую программу, которая поможет понять, как планировщик работает «под капотом».

```
#include "rxcpp/rx.hpp"
int main() {
    // получить координацию
    auto coordination = rxcpp::serialize_new_thread();

    // создать рабочий поток из координации
    auto worker = coordination.create_coordinator().get_worker();

    // создать действие
    auto sub_action = rxcpp::schedulers::make_action(
        [] (const rxcpp::schedulers::schedulable&) {
            printf("Action Executed in Thread # : %d\n",
                std::this_thread::get_id());
        });

    // создать единицу планирования -
    // привязать действие к рабочему потоку
    auto scheduled = rxcpp::schedulers
        ::make_schedulable(worker, sub_action);
    // запланировать единицу на выполнение
    scheduled.schedule();

    return 0;
}
```

В библиотеке RxCpp все операции, ожидающие на вход несколько потоков данных, а также операции, имеющие отношение к физическому времени, в качестве одного из аргументов принимают функцию координации. Вот некоторые из функций координации, поддерживаемых библиотекой:

- `identity_immediate();`
- `identity_current_thread();`
- `identity_same_worker(worker w);`
- `serialize_event_loop();`
- `serialize_new_thread();`
- `serialize_same_worker(worker w);`
- `observe_on_event_loop();`
- `observe_on_new_thread();`

В предыдущем примере мы своими руками создали и запланировали на выполнение некоторое действие (заданное лямбда-выражением), точно выписав все подробности этого. Теперь покажем более декларативные приёмы работы с планировщиком. Напишем программу, которая ставит задачи на выполнение, используя функцию координации.

```

#include "rxcpp/rx.hpp"
int main()
{
    // функция координации
    auto coordination = rxcpp::identity_current_thread();
    // получить объект-координатор и создать рабочий поток
    auto worker = coordination.create_coordinator().get_worker();
    // момент начала работы и периодичность операций
    auto start = coordination.now() +
        std::chrono::milliseconds(1);
    auto period = std::chrono::milliseconds(1);
    // создать наблюдаемый источник с возможностью повторения
    auto values = rxcpp::observable<>::interval(start, period)
        .take(5)
        .replay(2, coordination);
    // первая подписка с использованием рабочего потока
    worker.schedule([&] (const rxcpp::schedulers::schedulable&) {
        values.subscribe(
            [] (long v) {
                printf(
                    "#1 -- %d : %ld\n",
                    std::this_thread::get_id(),
                    v);
            },
            [] () { printf("#1 --- OnCompleted\n");});
    });

    worker.schedule([&] (const rxcpp::schedulers::schedulable&) {
        values.subscribe(
            [] (long v) {
                printf(
                    "#2 -- %d : %ld\n",
                    std::this_thread::get_id(),
                    v);
            },
            [] () { printf("#2 --- OnCompleted\n");});
    });

    // запустить генерацию значений
    worker.schedule([&] (const rxcpp::schedulers::schedulable&) {
        values.connect();
    });

    // блокирующая подписка, чтобы дождаться результатов
    values.as_blocking().subscribe();

    return 0;
}

```

В этой программе создаётся горячий источник данных с возможностью повторения сгенерированных данных для поздних подписчиков. Также создаётся рабочий поток, чтобы система могла ставить на выполнение обработку данных

по подпискам и присоединять наблюдателей к наблюдаемым источникам. Тем самым программа демонстрирует работу планировщиков в библиотеке RxCpp.

Методы `observe_on` и `subscribe_on`

Операции `observe_on` и `subscribe_on` ведут себя несколько различным образом, что часто становится источником недоразумений для тех, кто делает первые шаги в реактивном программировании. Операция `observe_on` влияет на поток, в котором выполняются операции и наблюдатели, расположенные «ниже по течению» в графе маршрутизации данных. Операция же `subscribe_on` влияет на обработчики данных как выше, так и ниже по течению. Следующая программа демонстрирует тонкое различие в поведении этих методов. Сначала напомним программу с использованием операции `observe_on`.

```
#include "rxcpp/rx.hpp"
int main(){
    // идентификатор главного потока
    printf("Главный поток: %d\n",
        std::this_thread::get_id());

    // операция map выполняется в главном потоке,
    // подписчик работает в новом потоке
    rxcpp::observable<>::range(0,15)
        .map([] (int i) {
            printf(
                "Map %d: %d\n",
                std::this_thread::get_id(),
                i);
            return i;
        })
        .take(5)
        .observe_on(rxcpp::synchronize_new_thread())
        .subscribe([&](int i){
            printf(
                "Sub %d : %d\n",
                std::this_thread::get_id(),
                i);
        });

    // выждать две секунды
    rxcpp::observable<>
        ::timer(std::chrono::milliseconds(2000))
        .subscribe([] (long) { });

    return 0;
}
```

Результат выполнения этой программы будет таким:

```
Главный поток: 1
Map 1: 0
Map 1: 1
```



```
Sub 2: 0
Map 1: 2
Sub 2: 1
Map 1: 3
Sub 2: 2
Map 1: 4
Sub 2: 3
Sub 2: 4
```

Этот текст ясно показывает, что операция `map` отрабатывает в главном потоке программы, тогда как подписанная на данные лямбда-функция ставится на выполнение в новом потоке. Отсюда очевидно, что применение метода `observe_on` влияет на диспетчеризацию только следующих после неё операций. Теперь посмотрим, как изменится поведение программы, если заменить метод `observe_on` на метод `subscribe_on`.

```
#include "rxcpp/rx.hpp"
int main(){
    // идентификатор главного потока
    printf("Главный поток: %d\n",
        std::this_thread::get_id());

    // операция map и подписчик
    // работают в новом потоке
    rxcpp::observable<>::range(0,15)
        .map([] (int i) {
            printf(
                "Map %d: %d\n",
                std::this_thread::get_id(),
                i);
            return i;
        })
        .take(5)
        .subscribe_on(rxcpp::synchronize_new_thread())
        .subscribe([&](int i){
            printf(
                "Sub %d : %d\n",
                std::this_thread::get_id(),
                i);
        });

    // выждать две секунды
    rxcpp::observable<>
        ::timer(std::chrono::milliseconds(2000))
        .subscribe([] (long) { });

    return 0;
}
```



Данная программа печатает на консоль следующий текст:

```
Главный поток: 1
Map 2: 0
```

```

Map 2: 1
Sub 2: 0
Map 2: 2
Sub 2: 1
Map 2: 3
Sub 2: 2
Map 2: 4
Sub 2: 3
Sub 2: 4

```



Легко видеть, что обе функции, и преобразователь данных, и подписчик, отработали в отдельном потоке. Этот пример отчётливо демонстрирует, что метод `subscribe_on` устанавливает поток выполнения для всех операций, встречающихся на текущем маршруте данных.

Планировщик с циклом выполнения `run_loop`

Библиотека RxCpp не содержит встроенного планировщика, который ставит все задачи на выполнение в главном потоке. Однако хорошего приближения к такому поведению можно добиться, воспользовавшись классом планировщика `run_loop`. В следующем примере источник данных работает в фоновом потоке, а функция-подписчик выполняется в главном потоке программы. Чтобы добиться такого поведения, используются методы `observe_on` и `subscribe_on`.

```

#include "rxcpp/rx.hpp"
int main() {
    // идентификатор главного потока
    printf("Главный поток %d\n", std::this_thread::get_id());

    // объект-планировщик, выполняющий
    // свой цикл обработки в главном потоке
    rxcpp::schedulers::run_loop rlp;

    // координация
    auto main_thread = rxcpp::observe_on_run_loop(rlp);
    // рабочий поток
    auto worker_thread = rxcpp::synchronize_new_thread();
    // объект-подписка
    rxcpp::composite_subscription scr;

    rxcpp::observable<>::range(0, 15)
        .map([] (int i) {
            // выполняется в рабочем потоке
            printf("Map %d: %d\n", std::this_thread::get_id(), i);
            return i;
        })
        .take(5)
        .subscribe_on(worker_thread)
        .observe_on(main_thread)
        .subscribe(scr, [&] (int i) {
            // выполняется в главном потоке

```



```

        printf("Sub %d: %d\n", std::this_thread::get_id(), i);
    });

    //----- Execute the Run Loop
    while (scr.is_subscribed() || !rlp.empty()) {
        while (!rlp.empty() && rlp.peek().when < rlp.now()) {
            rlp.dispatch();
        }
    }

    return 0;
}

```

Эта программа напечатает на консоль такой текст:

Главный поток: 1

Map 2: 0

Map 2: 1

Sub 1: 0

Sub 1: 1

Map 2: 2

Map 2: 3

Sub 1 : 2

Map 2: 4

Sub 1: 3

Sub 1: 4



Можно убедиться, что функция-преобразователь, которую операция `map` применяет к каждому элементу потока данных, выполняется в рабочем потоке, а функции-подписчики работают в главном потоке программы. Это достигнуто с помощью трюка с применением методов `subscribe_on` и `observe_on` одного за другим – их свойства были рассмотрены в предыдущем разделе.

ОПЕРАЦИИ НАД ПОТОКАМИ ДАННЫХ

Операция над потоком данных – это сущность, которая применяет определённую функцию к каждому элементу потока данных и образует тем самым новый поток. При этом объекты данных, поступающие на вход, остаются неизменными – операция представляет собой чистую функцию (в том смысле, который этот термин имеет в парадигме функционального программирования). В примерах программ, рассмотренных в предыдущих разделах, встречалось множество разнообразных операций над потоками. В главе 9 будет показано, как создавать собственные операции. Именно благодаря тому, что операция не изменяет исходные данные, и становится возможным декларативное управление планировщиком. Операции над потоками данных в реактивном программировании можно разделить на следующие категории:

- операции создания потоков;
- операции преобразования данных;
- операции фильтрации;

- операции комбинирования данных;
- операции обработки ошибок;
- вспомогательные операции;
- логические операции;
- математические операции.

Кроме того, есть несколько операций, не подпадающих ни под одну из этих категорий. Сделаем краткий обзор важнейших операций каждой категории.

Операции создания потоков



К этой разновидности относятся операции, которые помогают создавать разнообразные наблюдаемые источники с теми или иными данными. Выше приводились примеры использования функций `create`, `from`, `interval` и `range`. Рекомендуем читателю освежить в памяти эти примеры, а также обратиться к документации по библиотеке RxCpp за более подробной информацией. Краткая сводка этой группы операций дана в следующей таблице.

Имя	Описание
<code>create</code>	Создаёт наблюдаемый источник, который для каждого нового подписчика выполняет определённую функцию (как правило, вызывает методы подписчика в определённой последовательности)
<code>defer</code>	Создаёт промежуточный объект-источник, который при подключении каждого нового наблюдателя выполняет определённую фабричную функцию для создания настоящего источника
<code>empty</code>	Создаёт источник, который не выдаёт никаких данных и сразу сообщает о завершении потока данных
<code>from</code>	Создаёт источник, который в качестве своих данных выдаёт значения, переданные через аргументы
<code>interval</code>	Создаёт источник, который в качестве своих данных выдаёт значения из определённого диапазона
<code>just</code>	Создаёт источник, который выдаёт единственное значение
<code>range</code>	Создаёт источник, данные для которого берутся из диапазона итераторов (в частности, из контейнера)
<code>never</code>	Создаёт источник, который никогда ничего не выдаёт (ни данных, ни признака завершения потока)
<code>repeat</code>	Создаёт источник, который бесконечно повторяет определённую последовательность значений
<code>timer</code>	Создаёт источник, который выдаёт значение после заданного промежутка времени
<code>throw</code>	Создаёт источник, который выдаёт только сигнал ошибки

Операции преобразования данных

Эта группа операций позволяет создать новый наблюдаемый источник на основе данных из другого источника. Исходный источник при этом изменений не претерпевает. К каждому элементу исходного потока применяется некоторая функция-преобразователь.

Имя	Описание
buffer	Данные нового источника – это смежные, непересекающиеся, ограниченные по длине «пачки» данных исходного источника
flat_map	К каждому элементу x исходного потока применяется функция, возвращающая поток новых значений y ; затем к каждой паре (x, y) применяется ещё одна функция и из возвращаемых ею значений формируется поток результатов
group_by	Элементы нового источника – это группы значений из исходного источника, выделенных по определённому ключу
map	Элементы нового источника – результаты применения определённой функции к элементам исходного источника
scan	Элементы нового источника – результаты последовательных применений функции-аккумулятора
window	Элементы нового источника – в свою очередь, источники данных, содержащие смежные, непересекающиеся, ограниченные по длине «пачки» данных исходного источника

Операции фильтрации

Возможность фильтровать потоки данных – это одна из наиболее часто встречающихся задач. Естественно, что библиотека RxCpp содержит множество операций для фильтрации данных. Критерии фильтрации обычно задаются функциями-предикатами или лямбда-выражениями. Некоторые наиболее интересные фильтры показаны в таблице.

Имя	Описание
debounce	Элемент попадает в выходной поток, если в течение определённого промежутка времени исходный источник данных не генерировал никаких данных
distinct	В выходной поток попадают только различающиеся между собой значения из входного потока
element_at	В выходной поток попадает значение, стоящее во входном потоке под заданным номером
filter	В выходной поток попадают только те значения из входного потока, которые удовлетворяют определённому предикату
first	В выходной поток попадает только значение, поступившее из входного потока первым
ignore_elements	В выходной поток попадает только сигнал окончания исходного потока
last	В выходной поток попадает только значение, поступившее из входного потока последним
sample	В выходной поток попадают значения, поступившие из входного потока последними в каждом периоде времени одинаковой длительности
skip	В выходной поток попадают все значения из входного потока, за исключением определённого числа значений в начале потока
skip_last	В выходной поток попадают все значения из входного потока, за исключением определённого числа значений в конце потока
take	В выходной поток попадает только определённое число значений в начале потока
take_last	В выходной поток попадает только определённое число значений в конце потока

Операции комбинирования данных

Одна из основных целей реактивной модели программирования состоит в том, чтобы ослабить, насколько возможно, связь источника данных с наблюдателем. Поэтому очевидна необходимость в промежуточных операциях, позволяющих тем или иным способом соединять между собой несколько потоков данных. В библиотеке RxCpp имеется ряд таких операций.

Имя	Описание
combine_latest	Когда очередной элемент поступает от какого-либо из двух источников, последние значения от обоих источников комбинируются заданной функцией, и её результат помещается в выходной поток
merge	Слияние нескольких источников данных в один: значение, полученное от любого источника-аргумента, попадает в выходной поток
start_with	Помещает в выходной поток заданную последовательность значений, затем помещает все значения из источника-аргумента
switch_on_next	Превращает наблюдаемый источник данных, значения которого суть источники данных, в свою очередь, в единый источник данных: в выходной поток помещаются элементы, получаемые от элемента, полученного последним из источника источников
zip	Комбинирует значения, получаемые от нескольких наблюдаемых источников, с помощью заданной функции, и помещает в выходной поток возвращённые ею значения

Операции обработки ошибок

Поток, выдаваемый наблюдаемым источником, может содержать не только данные и признак нормального завершения, но и сигнал ошибки. Следующие операции позволяют организовать их обработку.

Имя	Описание
catch	Не поддерживается библиотекой RxCpp
retry	Передавать на выход все данные, полученные из входного потока, пока он выдаёт данные; если входной поток выдаёт ошибку, подключаться к нему заново (количество повторных попыток ограничено)

Вспомогательные операции

В этой группе собраны разнообразные полезные инструменты для работы с источниками данных.

Имя	Описание
finally	Передать на выход без изменений все данные из входного потока, а после его завершения выполнить определённое действие
observe_on	Задать планировщик, посредством которого должна быть организована доставка данных от наблюдаемого источника к наблюдателю
subscribe	Подписать наблюдателя на оповещения от наблюдаемого источника
subscribe_on	Задать планировщик, которым должен пользоваться наблюдаемый источник, когда на него подписывается наблюдатель
scope	Создать ресурс, поддерживающий операцию освобождения, время жизни которого совпадает со временем работы наблюдаемого источника

Логические операции

Операции из этой группы имеют дело с логическими значениями и проверками условий, а также управляют потоками данных в зависимости от этих условий.

Имя	Описание
all	Выдать в выходной поток логическое значение «истина», если все элементы входного потока удовлетворяют заданному предикату; в противном случае выдать значение «ложь»
amb	Передать на выход содержимое того из нескольких входных потоков, который первым выдаст данные или сигнал завершения
contains	Выдать в выходной поток логическое значение «истина», если во входном потоке встретилось заданное значение
default_if_empty	Если входной поток пуст, выдать в выходной поток заранее заданное значение
sequence_equal	Выдать в выходной поток значение «истина», если оба входных потока завершаются нормальным образом и выдают одни и те же значения; в противном случае выдать в выходной поток значение «ложь»
skip_until	Пропустить начальные элементы входного потока до первого элемента, удовлетворяющего некоторому предикату
skip_while	Пропустить начальные элементы входного потока, пока они удовлетворяют некоторому предикату
take_until	Передавать на выход элементы входного потока до первого элемента, удовлетворяющего некоторому предикату
take_while	Передавать на выход элементы входного потока, пока они удовлетворяют некоторому предикату

Математические операции и агрегирование потоков

Эти операции работают с потоком значений как с одним целым и превращают весь поток в единственное итоговое значение.

Имя	Описание
average	Вычислить среднее арифметическое элементов входного потока и выдать это значение в выходной поток
concat	Передать на выход содержимое обоих входных потоков, не допуская их перемешивания
count	Подсчитать число элементов во входном потоке и выдать это число в выходной поток
max	Определить наибольшее значение во входном потоке и выдать его в выходной поток
min	Определить наименьшее значение во входном потоке и выдать его в выходной поток
reduce	Выполнить свёртку всех значений из входного потока по бинарной операции и выдать окончательный результат в выходной поток
sum	Подсчитать сумму значений из входного потока и выдать её в выходной поток

Операции для управления подключениями

К этой группе относятся операции, которые позволяют тонко настраивать поведение наблюдаемых источников данных при подписке.



Имя	Описание
connect	Запустить генерацию данных наблюдаемым источником
publish	Преобразовать обычный наблюдаемый источник в источник с управляемыми подключениями
ref_count	Преобразовать наблюдаемый источник с управляемыми подключениями в обычный источник
replay	Обеспечить доставку всем подписчикам (даже подключившимся позже) одних и тех же данных

Итоги

Эта глава призвана привести читателя к пониманию того, как различные части библиотеки RxCpp и отдельные элементы реактивной модели программирования вместе образуют целостную картину. Главу открывал разговор о наблюдаемых источниках данных, затем рассматривались особенности горячих и холодных источников. Далее речь шла о механизме подписки и способах его использования. В следующем разделе рассказывалось об объектах-темах, которые сочетают в себе свойства наблюдаемого источника и наблюдателя, в том числе об их разновидностях, поддерживающих дополнительную функциональность. Затем следовал разбор важной темы – планировщиков и способов их тонкой настройки. Завершал главу обзор основных категорий операций, поддерживаемых в библиотеке RxCpp. В следующей главе читатель узнает, как применить все эти сведения для разработки реальных программ с графическим интерфейсом пользователя на основе библиотеки Qt.



Глава 9

.....

Реактивное программирование графических интерфейсов на основе каркаса Qt

Каркас Qt (произносится «кьют» как английское слово «cute», означающее как «находчивый, остроумный», так и «миловидный») предоставляет программисту целую экосистему для создания на языке C++ кроссплатформенных и многоплатформенных приложений с графическим интерфейсом. Если при создании программ ограничить себя исключительно переносимым ядром библиотеки, можно извлечь немалую выгоду, написав приложение один раз и затем компилируя его для каких угодно платформ, – в этом состоит поддерживаемая библиотекой парадигма. Впрочем, библиотека Qt позволяет использовать и специфические возможности отдельных платформ – например, при написании приложений для ОС Windows поддерживается технология ActiveX.

Нередко библиотека Qt оказывается предпочтительнее библиотеки MFC, даже если приложение создаётся только для ОС Windows. Это вполне объясняется сочетанием крайней простоты программирования (для работы нужно довольно небольшое подмножество языка C++) с необычайным богатством возможностей. Целью разработчиков каркаса Qt изначально была, конечно же, кроссплатформенная разработка приложений. Переносимость приложений между платформами, не требующая модификации их кода, разнообразие возможностей, открытый исходный код самой библиотеки, а также наличие подробной, постоянно обновляемой документации делают этот каркас чрезвычайно удобным для программистов. Всё это обеспечило каркасу Qt процветание на протяжении почти четверти столетия – с момента выпуска его первой версии в 1995 г.

Каркас Qt предоставляет программистам единую и всеохватывающую среду для создания разнообразных приложений: так, поддерживаются графические интерфейсы пользователя, программный интерфейс к движку WebKit для отображения веб-страниц, обработка мультимедийных данных, интерфейс к файловой системе, технология OpenGL и многое другое. Рассказ обо всех возможностях этой замечательной системы потребовал бы отдельной книги. Задача настоящей главы – представить краткое введение в разработку реактивных приложений с графическим пользовательским интерфейсом на основе каркаса Qt и библиотеки RxCpp. Основы реактивной модели программирования были изучены ранее в главах 7 и 8. Пришло время применить изученное на практике! Библиотека Qt обладает собственной хорошо продуманной системой обработки событий, и читателю нужно сначала хорошо изучить эти её особенности, чтобы затем добавить к ним возможности библиотеки RxCpp. В этой главе будут рассмотрены следующие вопросы:

- вводный курс разработки графических интерфейсов пользователя с использованием каркаса Qt;
- программа «Здравствуй, мир» на основе библиотеки Qt;
- модель обработки событий в каркасе Qt: сигналы, слоты и метаобъектный компилятор;
- интеграция библиотеки RxCpp с моделью событий библиотеки Qt;
- создание собственных операций над потоками данных средствами библиотеки RxCpp.

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ ИНТЕРФЕЙСОВ ПОЛЬЗОВАТЕЛЯ НА ОСНОВЕ КАРКАСА QT

Каркас Qt предназначен для разработки кроссплатформенных приложений, то есть позволяет создавать программы, одинаково хорошо компилирующиеся в исполняемый код для различных платформ, не требующие для этого модификации исходного кода и работающие с такой же производительностью, как и приложения, специально написанные под конкретную платформу. Помимо приложений с графическим интерфейсом пользователя, эта система бывает полезна и для написания консольных приложений с интерфейсом командной строки, но всё же основное её предназначение – это создание графических интерфейсов.

Приложения, основанные на библиотеке Qt, обычно пишутся на языке C++, однако существуют интерфейсы привязки к другим языкам. Данная библиотека позволяет справиться со многими трудностями, характерными для разработки на языке C++, благодаря многообразному интерфейсу прикладного программирования и мощным инструментам разработки. Каркас Qt поддерживает множество компиляторов (и связанных с ними инструментальных наборов), включая компиляторы GCC, clang и Visual C++. Система Qt также предоставляет разработчикам инструмент Qt Quick для разработки логики взаимодействия

пользователя с приложением, включающий в себя язык QML – декларативный язык сценариев, предназначенный для моделирования визуальных интерфейсов и основанный на языке ECMAScript. Всё это значительно упрощает быструю разработку приложений для мобильных платформ, причём та же самая логика может быть написана вручную, если требуется максимально возможная производительность. Совместное использование языков ECMAScript и C++ позволяет сочетать простоту декларативного стиля с высокой производительностью.

Каркас Qt в настоящее время разрабатывается и поддерживается компанией Qt Company. Данное ПО доступно как под открытой, так и под коммерческой лицензией. В первых версиях библиотеки использовались собственная графическая подсистема и набор визуальных элементов, что позволяло создавать графические интерфейсы, выглядящие совершенно одинаково на различных платформах, а также подражать внешнему виду любой платформы (например, в системе GNU Linux получить графический интерфейс в стиле ОС Windows). Это позволяло разработчикам легко переносить приложения между платформами и свести к минимуму зависимость приложений от конкретных платформ. Однако из-за несовершенства данного механизма в последующих версиях библиотеки Qt перешли к использованию визуальных элементов целевой платформы и соответствующих системных вызовов. Это решило проблемы, возникавшие ранее с собственной графической подсистемой, но платой за это стала потеря единого, независимого от целевой платформы внешнего вида приложений. Библиотека Qt обладает превосходным программным интерфейсом привязки для языка программирования Python, получившим название PyQt.

Есть несколько важных принципов, которые программисту нужно осмыслить, прежде чем использовать библиотеку Qt в своей работе. В следующих разделах будут кратко разобраны основы объектной модели, модель сигналов и слотов, система обработки событий и система мета-объектов.

Объектная модель библиотеки Qt

Ключевыми критериями качества каркасов для создания графических интерфейсов являются как эффективность выполнения, так и гибкость вместе с высоким уровнем абстракции. Объектная модель, встроенная в язык C++, обеспечивает крайне высокую эффективность, однако её статическая сущность может оказаться обременительной в некоторых классах задач. Каркас Qt соединяет высокое быстродействие, присущее языку C++, с очень гибкой объектной моделью. Важнейшие составные части библиотеки Qt – это:

- механизм **сигналов и слотов** для общения объектов друг с другом;
- механизм **свойств объекта**, поддерживающих чтение и запись значений;
- мощный аппарат событий, включая средства фильтрации событий;
- таймеры с богатыми функциональными возможностями, обеспечивающие плавную асинхронную работу графических интерфейсов приложения;

- механизм **интернационализации** приложений с контекстно-зависимым переводом;
- собственная реализация **умных указателей**, которые автоматически обнуляются при уничтожении объекта;
- возможность **динамического преобразования типов** даже между разными библиотеками.

Большая часть этой функциональности реализована штатными средствами языка C++ в виде классов, порождённых от базового класса `QObject`. Для реализации оставшихся элементов, таких как сигналы и слоты или механизм свойств, требуется система метаобъектов, поддерживаемая особым инструментом – **компилятором метаобъектов** (meta-object compiler, МОС), включённым в состав каркаса Qt. Система метаобъектов представляет собой расширение языка C++, которое делает его более приспособленным для программирования графических интерфейсов. Этот инструмент работает как предварительный компилятор, который генерирует вспомогательный код, основываясь на специальных синтаксических конструкциях в исходном коде. Рассмотрим некоторые классы, играющие важную роль в объектной модели библиотеки Qt.

Имя	Описание
<code>QObject</code>	Базовый класс для всех остальных классов библиотеки Qt
<code>QPointer</code>	Шаблон умного указателя на объект класса, производного от класса <code>QObject</code>
<code>QSignalMapper</code>	Класс, отвечающий за пересылку сигналов от известных отправителей
<code>QVariant</code>	Тип-объединение наиболее важных типов данных
<code>QMetaClassInfo</code>	Метаданные класса (т. е. объект-хранилище различной информации о каком-либо классе)
<code>QMetaEnum</code>	Метаданные о типе перечисления
<code>QMetaMethod</code>	Метаданные о функции-члене класса
<code>QMetaObject</code>	Метаданные об объекте
<code>QMetaProperty</code>	Метаданные о свойстве
<code>QMetaType</code>	Класс для управления именованными типами в системе метаобъектов
<code>QObjectCleanupHandler</code>	Класс для управления временем жизни объектов классов, порождённых от класса <code>QObject</code>

Объекты в библиотеке Qt обычно трактуются не как значения, а как идентичности. Идентичности клонируются, а не копируются или присваиваются. Клонирование – более сложная операция, чем копирование или присваивание значений. По этой причине конструкторы копирования и операции присваивания удалены из класса `QObject` и его подклассов.

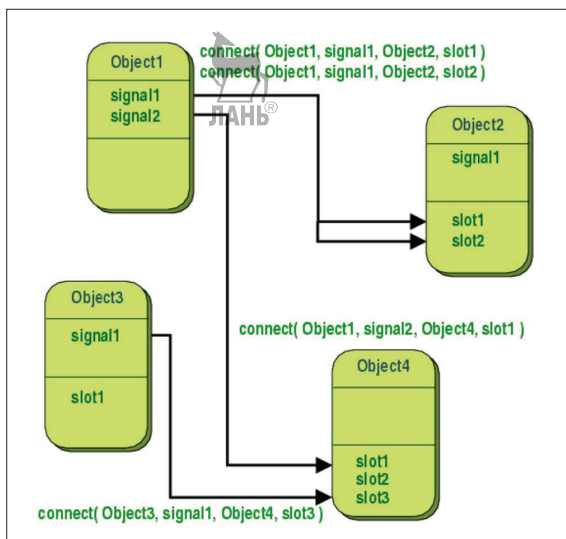
Сигналы и слоты

Сигналы и слоты – это механизм, используемый в библиотеке Qt, чтобы организовать общение между объектами. Механизм сигналов и слотов занимает центральное место в архитектуре библиотеки Qt и особенно важен для программирования графических интерфейсов. С помощью этого механизма элементы

визуального интерфейса получают оповещения об изменениях, происходящих в других элементах интерфейса. Вообще говоря, этот механизм обеспечивает общение любых объектов, наследующих класс `QObject`. Например, когда пользователь нажимает мышью на кнопку **Заккрыть**, об этом приходит оповещение, а его обработка автоматически вызывает метод `close` объекта-окна.

Механизм сигналов и слотов представляет собой альтернативу традиционной для программирования на языках С и С++ технике функций обратного вызова. Объект испускает сигнал всякий раз, когда происходит определённое событие. Все классы визуальных элементов управления в каркасе Qt обладают предопределёнными сигналами. Кроме того, разработчик может создать свой класс визуального элемента, порождённый от библиотечного класса, и объявить в нём новые сигналы. Слотом называется специальная функция-обработчик, которая автоматически вызывается в ответ на сигнал. Как и в случае сигналов, классы визуальных элементов обладают множеством предопределённых слотов, но разработчик может добавлять в порождённых классах и свои слоты и подписывать их на те или иные сигналы.

На следующей диаграмме, взятой из официальной документации по библиотеке Qt (<http://doc.qt.io/archives/qt-4.8/signalsandslots.html>), показано, как происходит общение объектов через сигналы и слоты.



Сигналы и слоты представляют собой механизм коммуникации с чрезвычайно слабой связью между участниками: класс, испускающий сигнал, ничего не знает о слотах (возможно, нескольких), которые его примут. Этот механизм представляет собой прекрасный пример систем, работающих по принципу «выстрели и забудь». Внутренние механизмы библиотеки гарантируют, что

если сигнал соединён со слотом, этот слот автоматически будет вызван с нужными аргументами и в нужный момент времени. Как сигналы, так и слоты могут принимать сколько угодно параметров каких угодно типов. Механизм в целом превосходно обеспечивает корректную работу с типами. Для связывания сигнала со слотом их сигнатуры должны полностью совпадать – поэтому ошибки несоответствия типов может обнаружить компилятор.

Все классы, порождённые от класса `QObject` прямо или косвенно (т. е. от его подклассов, как, например, класс `QWidget`), могут обладать сигналами и слотами. Сигналы могут испускаться объектом, когда в нём происходит изменение, которое может представлять интерес для других объектов. При этом сам объект не обязан знать, есть ли у этого сигнала получатели. К одному сигналу может подключиться сколь угодно много слотов. Подобным же образом к одному слоту можно подключить сколько угодно сигналов (возможно, от разных объектов-источников). Можно даже подключить сигнал к другому сигналу, строя тем самым цепочки сигналов.

Таким образом, система сигналов и слотов образует чрезвычайно гибкий и расширяемый механизм программирования.

Подсистема событий

События, как они понимаются в каркасе Qt, происходят в самом приложении или вследствие действий пользователя, о которых приложению следует знать. События представлены объектами класса `QEvent` и порождённых от него подклассов. Получать и обрабатывать события может объект любого класса, порождённого от класса `QObject`, но особенно это характерно для классов визуальных элементов интерфейса.

Всякий раз, когда происходит событие, создаётся экземпляр соответствующего подкласса класса `QEvent`, затем он передаётся адресату – объекту класса `QObject` (или его подкласса) – путём вызова его метода `event`. Сам по себе этот метод не обрабатывает событие. Вместо этого он, исходя из фактического типа объекта-события, вызывает функцию-обработчик, наиболее подходящую для этого типа, и возвращает логическое значение «истина». Если же подходящего обработчика не нашлось, событие игнорируется, а метод `event` возвращает значение «ложь».

Некоторые события (например, события классов `QCloseEvent` и `QMoveEvent`) происходят из самого приложения; некоторые другие, как события `QMouseEvent` и `QKeyEvent`, приходят от оконной системы; прочие же – скажем, событие `QTimerEvent`, возникают из иных источников. Большинство событий оформляется в виде подклассов класса `QEvent`, в которых имеются поля данных и методы, выражающие специфические для этого типа событий параметры и поведение. Например, в классе `QMouseEvent` имеются методы `x` и `y`, с помощью которых объект-адресат может узнать положение указателя мыши.

Каждый объект-событие содержит целочисленный идентификатор, характеризующий тип события, который можно получить с помощью метода `type` – им

удобнее всего пользоваться во время выполнения программы, чтобы быстро выяснить, каков фактический класс объекта-события.

Обработчики событий



В общем случае обработка событий осуществляется путём вызова подходящих виртуальных функций-обработчиков. Именно эти виртуальные функции ответственны за правильные действия в ответ на то или иное событие. Если виртуальная функция, реализованная в специфическом подклассе, отвечает лишь за часть необходимой обработки, может понадобиться обращение к реализации из базового класса.

В следующем примере показан метод специфического, созданного программистом класса визуального элемента, который обрабатывает исключительно щелчки левой кнопкой мыши. Нажатия всех остальных кнопок отправляются для обработки базовому классу `QLabel`.

```
void my_QLabel::mouseMoveEvent(QMouseEvent *e) {
    if (e->button() == Qt::LeftButton) {
        // нажатия левой кнопкой обрабатывать здесь
        qDebug() << "X: " << e->x() << "Y: " << e->y() << "\n";
    }
    else {
        // нажатия остальных кнопок оставить базовому классу
        QLabel::mouseMoveEvent(e);
    }
}
```

Если разработчик порождённого класса хочет полностью подменить всю функциональность базового, он должен реализовать в виртуальной функции все возможные случаи события. Если же требуется лишь расширить поведение базового класса, можно реализовать своими руками лишь интересующие аспекты поведения, а обработку всех прочих случаев делегировать базовому классу.

Отправка событий

Во многих приложениях, основанных на каркасе Qt, бывает нужно генерировать собственные типы событий и обрабатывать их наравне с событиями, встроенными в каркас. Это несложно сделать: достаточно создать экземпляр своего класса события и послать его на обработку посредством методов `sendEvent` или `postEvent` класса `QCoreApplication`.

Метод `sendEvent` выполняется синхронно: он немедленно выполняет обработку события. У всех событий есть метод `isAccepted`, который позволяет узнать, было ли это событие принято или отвергнуто последним из вызванных для него обработчиков.

Метод `postEvent` работает асинхронно. Он помещает объект-событие в очередь для последующей обработки. На следующей итерации главного цикла об-

работки событий накопленные в очереди события направляются на обработку с некоторыми оптимизациями. Например, если в очередь добавлено несколько событий типа «размер окна изменён», то на обработку отправляется лишь одно, соответствующее окончательному размеру окна, что позволяет избежать многократной перерисовки содержимого окна.



Система метаобъектов

Система метаобъектов составляет основу для реализации механизма сигналов и слотов, механизма динамических свойств объекта и рефлексии – работы с информацией о типах и времени выполнения.

Система метаобъектов опирается на три ключевых элемента:

- класс `QObject` – общий предок всех классов, для которых предполагается использовать механизмы метаобъектов;
- макрос `Q_OBJECT`, который нужно вписать в объявление класса, в нём работали метаобъектные механизмы;
- метаобъектный компилятор МОС, который для каждого класса, порождённого от класса `QObject`, генерирует код реализации его метаобъектной функциональности.

Метаобъектный компилятор МОС отработывает до того, как исходный код поступает собственно компилятору с языка C++. Встретив в исходном коде какого-либо класса макрос `Q_OBJECT`, компилятор МОС создаёт ещё один исходный файл с реализацией метаобъектной функциональности для этого класса. Затем обычный компилятор языка C++ обрабатывает модули, созданные разработчиком, и эти сгенерированные метаобъектным компилятором файлы.

ПРОГРАММА «ЗДРАВСТВУЙ, МИР» НА ОСНОВЕ БИБЛИОТЕКИ Qt

Пора заняться разработкой графического приложения на языке C++ с использованием каркаса Qt. Для того чтобы продолжить изучение следующих разделов, читателю рекомендуется загрузить библиотеку, инструментарий и интегрированную среду разработки Qt Creator с официального сайта (<https://www.qt.io/download>). Примеры, приведённые далее в этой главе, полностью подпадают под условия открытой лицензии LGPL и состоят исключительно из кода на языке C++, который читатель может ввести вручную. Каркас Qt в целом разработан интуитивно понятным и приятным в использовании, чтобы целое приложение можно было создать, набирая код вручную, без использования интегрированной среды разработки.



Система Qt Creator представляет собой кроссплатформенную интегрированную среду разработки для языков C++, JavaScript и QML и входит, как часть инструментария разработки графических приложений, в состав каркаса Qt. Эта среда разработки содержит отладчик и встроенный визуальный редактор графических интерфейсов. Возможности редактора исходных текстов включают подсветку синтаксиса и автоматическое дополнение. В системах GNU Linux и FreeBSD среда Qt Creator использует компилятор из кол-

лекции GCC. В системе Windows среда при стандартной установке может использовать компиляторы MinGW или MSVC, также при компиляции среды из исходного кода её можно настроить на использование отладчика Microsoft Console Debugger. Ещё можно настроить среду на использование компилятора clang. За более подробной информацией мы отсылаем к Википедии (https://en.wikipedia.org/wiki/Qt_Creator).

Начнём с простой программы «Здравствуй, мир», которая создаёт и отображает на экране окно со статическим текстом.

```
#include <QApplication>
#include <QLabel>

int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QLabel label("Здравствуй, мир Qt!");
    label.show();
    return app.exec();
}
```



В этой программе используются два заголовочных файла: `QApplication` и `QLabel`. Имена заголовочных файлов в библиотеке Qt всегда совпадают с именами классов, что чрезвычайно удобно. Класс `QApplication` отвечает за управление ресурсами приложения. В каждом приложении, основанном на каркасе Qt, должен быть ровно один объект этого класса. Его конструктор принимает переданные программе параметры командной строки, а метод `exec` запускает главный цикл обработки событий.

❗ Цикл обработки событий выполняется постоянно, пока работает приложение, и отвечает за постановку событий в очередь, выборку их из очереди в соответствии с приоритетами и отправку объектам для обработки. В приложениях, построенных по принципу обработки событий, часть функциональности обычно бывает реализована в виде пассивного интерфейса и вызывается исключительно в ответ на определённые события. Обычно цикл обработки событий продолжает свою работу до тех пор, пока не происходит прерывающее событие – например, пока пользователь не нажмёт на кнопку закрытия главного окна приложения.

Класс `QLabel` – простейший из визуальных элементов в библиотеке Qt. Это текстовая метка, т. е. окно (возможно, дочернее для другого окна) с текстом. В этом примере текстовая метка при инициализации получает текст «Здравствуй, мир Qt!». Когда данное приложение вызывает метод `show` этого объекта, текстовая метка отображается на экране. В данном случае она является главным окном приложения, поэтому обладает собственной рамкой, заголовком и кнопками минимизации, максимизации и закрытия.

Для того чтобы собрать это приложение, помимо исходного текста на языке C++, нужен ещё файл проекта – оформленное на особом языке описание того, из каких исходных файлов проект состоит и как их компилировать. Чтобы создать файл проекта и собрать приложение, нужно выполнить следующие действия:

- создать директорию для проекта и сохранить файл с исходным кодом на языке C++ в эту директорию;
- открыть консоль и проверить версию установленной в системе утилиты `qmake` (часть инструментария Qt) с помощью команды `qmake -v`. Если система не может найти эту утилиту, нужно добавить её расположение в файловой системе к переменной среды `PATH`;
- перейти в директорию проекта, т. е. сделать её текущей директорией, и выполнить команду `qmake -project`. Это приведёт к созданию файла проекта, который имеет расширение `.pro`;
- открыть файл проекта и после имеющегося в нём слова `INCLUDEPATH` добавить следующий текст:

```
INCLUDEPATH += .
QT += widgets
```

- запустить утилиту `qmake` без параметров командной строки. На основе файла проекта будет создан `make`-файл с подробными правилами сборки приложения;
- запустить утилиту `make` (или, в зависимости от текущей платформы, заменяющие её утилиты `nmake` или `gmake`) – это приведёт к сборке приложения;
- теперь можно запустить приложение – на экране появится небольшое окно с приветствием;
- если позднее вносятся изменения в исходный код уже имеющихся исходных файлов, для построения приложения нужно повторить лишь шаг 6. Если в проект добавлены новые исходные файлы, нужно вписать их в файл проекта и повторить с шага 5.

i Эта пошаговая инструкция годится для подготовки и сборки практически любых приложений, основанных на каркасе Qt, могут лишь различаться изменения, вносимые в файл проекта. Всюду далее этой главе при разборе примеров фраза «сконфигурировать, построить и запустить» означает именно эту последовательность шагов.

Прежде чем переходить к более сложным и интересным примерам, давайте немного позабавимся. Изменим в исходном коде строку, в которой инициализируется объект класса `QLabel`, следующим образом:

```
QLabel label("<h2><i>Здравствуй, мир</i> <font color=green>Qt!</font><h2>");
```

Затем заново соберём и запустим приложение. Как показывает этот пример, библиотека Qt позволяет легко менять внешний вид пользовательского интерфейса с помощью языка разметки HTML.

В следующем разделе будет показано, как обрабатывать события и как использовать механизм сигналов и слотов для обмена информацией между объектами.

СОБЫТИЯ, СИГНАЛЫ И СЛОТЫ НА ПРИМЕРЕ

В этот разделе нам предстоит создать приложение, которое обрабатывает события от мыши, адресованные текстовой метке. Для этого понадобится расширить стандартный обработчик событий от мыши, реализованный в классе `QLabel`, породив от него свой подкласс. Окончательной обработкой этих событий будет заниматься диалоговое окно, в котором размещена такая модифицированная текстовая область. Логика данного приложения в общих чертах такова.

- Создать собственный класс `my_QLabel`, порождённый от встроенного в библиотеку класса `QLabel`, и расширить в нём метод, ответственный за обработку событий от мыши (таких как перемещение мыши, нажатие кнопки и выход указателя мыши за границы этого графического объекта).
- Определить в классе `my_QLabel` собственные сигналы и испускать их, когда происходят соответствующие события.
- Создать собственный класс диалогового окна на основе библиотечного класса `QDialog`, вручную назначить координаты и размеры размещённых в нём графических элементов, включая и модифицированную текстовую область, которая умеет обрабатывать события от мыши.
- В этом классе определить слоты для обработки сигналов от визуального объекта `my_QLabel` – пусть они отображают информацию в диалоговом окне.
- В главной функции приложения, после создания экземпляра класса `QApplication`, инициализировать диалоговое окно и запустить главный цикл приложения.
- Создать файл проекта, собрать приложение и запустить его.

Создание собственного визуального объекта

Начнём с объявления класса в заголовочном файле:

```
#ifndef MY_QLABEL_H
#define MY_QLABEL_H

#include <QLabel>
#include <QMouseEvent>

class my_QLabel : public QLabel
{
    Q_OBJECT
public:
    explicit my_QLabel(QWidget *parent, int x, int y);

    void mouseMoveEvent(QMouseEvent *evt);
    void mousePressEvent(QMouseEvent* evt);
    void leaveEvent(QEvent* evt);

    int x, y;
```

```
signals:
    void Mouse_Pressed();
    void Mouse_Position();
    void Mouse_Left();
};
```



```
#endif // MY_QLABEL_H
```

Классы `QLabel` и `QMouseEvent`, которые используются в этом коде, объявлены в одноимённых заголовочных файлах, директивы для их включения расположены в начале кода. Класс `my_QLabel` сделан порождённым от класса `QLabel`, что позволяет унаследовать поведение по умолчанию и переопределить только то, что должно отличаться. Поскольку библиотечный класс `QLabel` порождён от класса `QObject`, наш класс получает в наследство и механизмы обработки сигналов.

Макрос `Q_OBJECT` в объявлении класса извещает метаобъектный компилятор о том, что для нашего класса `my_QLabel` нужно сгенерировать специальный код. Этот генерируемый код необходим для поддержки данным классом сигналов и слотов, рефлексии и механизма динамических свойств.

Помимо конструктора, объявление этого класса содержит методы для обработки трёх событий от мыши: перемещение, нажатие кнопки и выход за границы визуального объекта. Кроме того, две общедоступные¹ переменные-члена целого типа содержат текущие координаты указателя мыши. Наконец, в секции `signals` объявлены три специфических сигнала, которыми данный объект оповещает своих наблюдателей о каждом из трёх событий.

Рассмотрим теперь, как эти методы реализованы в соответствующем модуле.

```
#include "my_qlabel.h"
```

```
my_QLabel::my_QLabel(QWidget *parent): QLabel(parent), x(0), y(0)
{}
```

```
void my_QLabel::mouseMoveEvent(QMouseEvent *evt)
{
    this->x = evt->x();
    this->y = evt->y();

    emit Mouse_Position();
}
```



Конструктор нашего класса принимает один аргумент, указатель на визуальный элемент интерфейса, родительский для данного объекта, – таково общее правило, которому следуют все визуальные классы библиотеки Qt. Этот указа-

¹ Вряд ли нужно обосновывать крайнюю неудачность такого проектного решения. Следовало бы снабдить сигнал `Mouse_Position` (как и соответствующий ему слот) двумя аргументами. – *Прим. перев.*

тель просто передаётся в конструктор базового класса. Кроме того, в конструкторе присваиваются начальные значения координатам мыши. В обработчике события перемещения (`mouseMoveEvent`) значения этих переменных обновляются, затем объект испускает сигнал `Mouse_Position`. Позднее будет показано, как объект диалогового окна соединяет этот сигнал со своим слотом и перерисовывает своё содержимое.

```
void my_QLabel::mousePressEvent(QMouseEvent *evt)
{
    emit Mouse_Pressed();
}
```

```
void my_QLabel::leaveEvent(QEvent *evt)
{
    emit Mouse_Left();
}
```



Обработчик события `mousePressEvent` (нажатие кнопки мыши) генерирует сигнал `Mouse_Pressed`, а обработчик события `leaveEvent` (выход указателя мыши за границу визуального элемента) испускает сигнал `Mouse_Left`. Эти сигналы, как будет показано далее, также подключены к соответствующим слотам диалогового окна, владеющего данным объектом, и эти слоты, в свою очередь, вызывают перерисовку содержимого окна. Таким образом, создание собственного класса визуального элемента на основе библиотечного класса закончено.

Создание главного диалогового окна приложения

Теперь, когда класс текстовой области, способной реагировать на события мыши, реализован, пора заняться классом диалогового окна, который управляет расположением своих графических элементов и обрабатывает сигналы, генерируемые объектом класса `my_QLabel`. Начнём с заголовочного файла `dialog.h`.

```
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>

class my_QLabel;
class QLabel;

class Dialog : public QDialog
{
    Q_OBJECT

public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();

private slots:
    void Mouse_CurrentPosition();
```



```

void Mouse_Pressed();
void Mouse_Left();

private:
    void initializeWidgets();
    my_QLabel *label_MouseArea ;
    QLabel *label_Mouse_CurPos;
    QLabel *label_MouseEvents;
};

#endif // DIALOG_H

```

В этом фрагменте кода объявлен класс `Dialog`, порождённый от библиотечного класса `QDialog`. Для классов `QLabel` и `my_QLabel` достаточно одних лишь упреждающих объявлений, их полные объявления будут нужны только в модуле с реализацией диалогового окна. Как уже отмечалось выше, объявление класса, участвующего в обмене сигналами (а также пользующегося рефлексией и обладающего динамическими свойствами), должно содержать макрос `Q_OBJECT`.

В классе диалогового окна, помимо конструктора и деструктора, объявлены закрытые слоты, которые можно подключить к сигналам класса `my_QLabel`. Слоты представляют собой обыкновенные методы, и их можно вызывать обычным способом, однако у слотов есть одна особенность: их можно подключать к сигналам. Так, в данном примере слот `Mouse_CurrentPosition` можно подключить к сигналу `Mouse_Position` класса `my_QLabel`, который испускается из обработчика `mouseMoveEvent`. Подобным же образом слоты `Mouse_Pressed` и `Mouse_Left` можно подключить к одноимённым событиям, которые генерируют, соответственно, обработчики `mousePressEvent` и `leaveEvent`.

Наконец, в закрытой секции этого класса объявлены поля-указатели на визуальные элементы, которыми владеет данное окно, и функция `initializeWidgets`, отвечающая за инициализацию и размещение этих элементов.

Реализация диалогового окна размещена в представленном ниже файле `dialog.cpp`.

```

#include "dialog.h"
#include "my_qlabel.h"
#include <QVBoxLayout>
#include <QGroupBox>

Dialog::Dialog(QWidget *parent): QDialog(parent)
{
    this->setWindowTitle("Обработка событий от мыши");
    initializeWidgets();

    connect(
        label_MouseArea,
        SIGNAL(Mouse_Position()),
        this,

```

```

        SLOT(Mouse_CurrentPosition()));
connect(
    label_MouseArea,
    SIGNAL(Mouse_Pressed()),
    this, SLOT(Mouse_Pressed()));
connect(
    label_MouseArea,
    SIGNAL(Mouse_Left()),
    this,
    SLOT(Mouse_Left()));
}

```

В конструкторе устанавливается текст, отображающийся в заголовке диалогового окна. Затем вызывается функция `initializeWidgets`, о которой речь пойдёт вскоре. После этого три вызова функции `connect` соединяют сигналы, выдаваемые объектом `label_MouseArea` класса `my_QLabel`, с соответствующими слотами класса `Dialog`.

```

void Dialog::Mouse_CurrentPosition()
{
    label_Mouse_CurPos->setText(
        QString("X = %1, Y = %2")
            .arg(label_MouseArea->x)
            .arg(label_MouseArea->y));
    label_MouseEvents->setText("Мышь движется");
}

```

Функция `Mouse_CurrentPosition` – это слот, на который поступают сигналы о каждом перемещении мыши по объекту `label_MouseArea` класса `my_QLabel`. Эта функция меняет текст, отображаемый в визуальном объекте `label_Mouse_CurPos`, на новые значения координат мыши, а элементу `label_MouseEvents` устанавливает текст «Мышь движется».

```

void Dialog::Mouse_Pressed()
{
    label_MouseEvents->setText("Нажата кнопка мыши");
}

```

Функция `Mouse_Pressed` – это слот, привязанный к сигналу, который выдаётся всякий раз, когда пользователь нажимает кнопку мыши внутри объекта `label_MouseArea` (нашего визуального элемента, отлавливающего события мыши). Эта функция показывает в элементе `label_MouseEvents` диалогового окна сообщение «Нажата кнопка мыши».

```

void Dialog::Mouse_Left()
{
    label_MouseEvents->setText("Мышь ушла");
}

```

Наконец, всякий раз, когда мышиный указатель выходит за границу визуального объекта `label_MouseArea`, этот объект получает событие `leaveEvent`, об-

работчик которого испускает сигнал `Mouse_Left`, а он, в свою очередь, соединён с одноимённым слотом диалогового окна. Функция-слот `Mouse_Left` меняет текст в элементе `label_MouseEvents` диалогового окна на сообщение «Мышь ушла».

Остаётся разобрать функцию `initializeWidgets`, которая создаёт дочерние визуальные элементы и размещает их в диалоговом окне.

```
void Dialog::initializeWidgets()
{
    label_MouseArea = new my_QLabel(this);
    label_MouseArea->setText("Площадка для мыши");
    label_MouseArea->setMouseTracking(true);
    label_MouseArea->setAlignment(
        Qt::AlignCenter|Qt::AlignHCenter);
    label_MouseArea->setFrameStyle(2);
```

В этом фрагменте создаётся объект `label_MouseArea` созданного нами класса `my_QLabel`. Затем настраиваются его свойства: отображаемый текст, режим отслеживания событий от мыши в его области, выравнивание по центру окна и толщина границы.

```
    label_Mouse_CurPos = new QLabel(this);
    label_Mouse_CurPos->setText("X = 0, Y = 0");
    label_Mouse_CurPos->setAlignment(
        Qt::AlignCenter|Qt::AlignHCenter);
    label_Mouse_CurPos->setFrameStyle(2);

    label_MouseEvents = new QLabel(this);
    label_MouseEvents->setText("Последнее событие");
    label_MouseEvents->setAlignment(
        Qt::AlignCenter|Qt::AlignHCenter);
    label_MouseEvents->setFrameStyle(2);
```

Похожим образом инициализируются и настраиваются два других визуальных элемента, это обычные текстовые поля библиотечного типа `QLabel`. В одном из них будут отображаться координаты мыши, в другом – последнее полученное от неё событие.

```
    QGroupBox *groupBox = new QGroupBox("Мышиные новости", this);
    QVBoxLayout *vbox = new QVBoxLayout;
    vbox->addWidget(label_Mouse_CurPos);
    vbox->addWidget(label_MouseEvents);
    vbox->addStretch(0);
    groupBox->setLayout(vbox);

    label_MouseArea->move(40, 40);
    label_MouseArea->resize(280,260);
    groupBox->move(330,40);
    groupBox->resize(200,150);
```

```
}
```

Под конец создаётся объект класса `QVBoxLayout`, который представляет собой контейнер визуальных элементов, автоматически управляющий расположением своих элементов. Данный тип контейнера выстраивает все свои элементы по вертикали. Объекты `label_Mouse_CurPos` и `label_MouseEvents` добавляются в этот контейнер. Создаётся визуальный объект – группа, ему устанавливается текст заголовка, а в качестве содержимого группы устанавливается вертикальный контейнер. В последнюю очередь устанавливаются координаты и размеры двух прямоугольных областей, из которых состоит диалоговое окно. Создание и конфигурирование внешнего вида окна окончено.

Запуск приложения

Теперь пора разобрать модуль `main.cpp`, который отвечает за создание и отображение главного окна.

```
#include "dialog.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Dialog dialog;
    dialog.resize(545, 337);
    dialog.show();
    return app.exec();
}
```

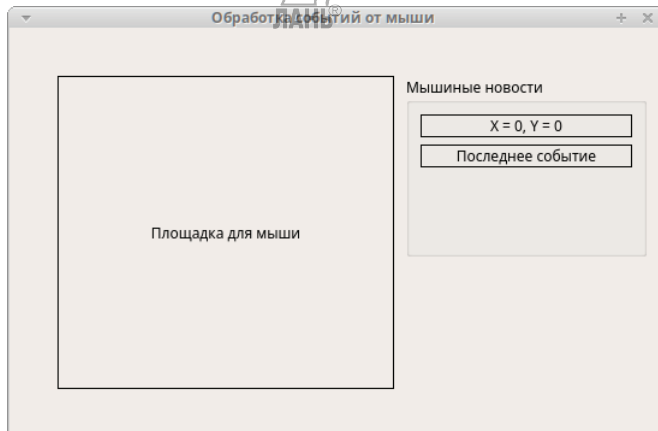
Этот код практически не отличается от кода приветственного приложения, о котором шла речь ранее. Отличие состоит в том, что здесь создаётся экземпляр класса `Dialog`, подробно описанного в предыдущем разделе, а также устанавливается размер этого окна с помощью метода `resize`. Тем самым исходный код приложения разобран полностью. Для его сборки и запуска понадобится ещё файл проекта, который приведём здесь полностью.

```
QT += widgets

SOURCES += \
    main.cpp \
    dialog.cpp \
    my_qlabel.cpp

HEADERS += \
    dialog.h \
    my_qlabel.h
```

Теперь всё готово, чтобы собрать и запустить приложение. На экране появится следующее диалоговое окно:



Если курсор мыши попадёт в панель, расположенную в левой части окна, координаты мыши начнут отображаться в верхнем из двух текстовых полей, расположенных справа, а в нижнем поле будет написано, что мышь движется. Если нажать любую кнопку мыши, пока её указатель по-прежнему находится в левой панели, сообщение об этом появится во втором текстовом поле. Наконец, когда указатель мыши покидает панель, об этом также отображается сообщение.

Из этого раздела читатель узнал, как средствами каркаса Qt создавать диалоговые окна, наполнять их визуальными элементами, управлять взаимным расположением элементов и другими подобными вещами. Также было рассказано о тонкой настройке визуального элемента (в данном примере – поля со статическим текстом) и о том, как обрабатывать системные события. Помимо этого, было показано, как соединять объекты посредством сигналов и слотов, объявленных самим разработчиком. Все эти сведения вместе взятые пригодились при разработке приложения, которое перехватывает события от мыши, произошедшие в его окне, и отображает сведения об этих событиях.

Займёмся теперь разработкой другого приложения, которое тоже перехватывает события от мыши, произошедшие в визуальном элементе, и отображает координаты указателя в другом элементе. Однако на этот раз обработка события будет осуществляться по-другому: первоначально перехватываемые средствами каркаса Qt, события будут проходить через механизмы подписки, предоставляемые библиотекой RxCpp, а затем снова попадать в графические объекты библиотеки Qt.

ИНТЕГРАЦИЯ БИБЛИОТЕК RxCPP И QT

В предыдущих разделах представлен беглый обзор каркаса Qt «с высоты птичьего полёта». В частности, были разобраны средства обработки системных событий и высокоуровневый механизм сигналов и слотов. В двух предыдущих

главах изучалась библиотека RxCpp и присущая ей модель программирования. Среди прочего было рассмотрено множество полезных и интересных реактивных операций.

Этот раздел посвящён разработке приложения, обрабатывающего события от мыши, происходящие в визуальном элементе, как и приложение из предыдущего примера. Однако на этот раз вместо испускания сигналов в обработчике события будет использоваться подписка на низкоуровневые события каркаса Qt средствами библиотеки RxCpp и последующая фильтрация потока событий. События, оставшиеся после фильтрации, будут отправляться подписчикам, снова в каркас Qt.

Реактивная фильтрация событий из каркаса Qt

Как уже отмечалось выше, каркас Qt обладает надёжным механизмом обработки системных событий. Нужно каким-то образом перекинуть мост между концептуальными схемами библиотек Qt и RxCpp. Разбор нашего приложения стоит начать с класса, который служит прослойкой между двумя событийными механизмами (заголовочный файл `rx_eventfilter.h`).

```
#ifndef RX_EVENTFILTER_H
#define RX_EVENTFILTER_H

#include <rxcpp/rx.hpp>
#include <QEvent>

namespace rxevt
{
    // фильтр-транслятор событий
    class EventEater: public QObject
    {
    public:
        EventEater(
            QObject* parent,
            QEvent::Type type,
            rxcpp::subscriber<QEvent*> s) :
            QObject(parent),
            eventType(type),
            eventSubscriber(s)
        {}

        ~EventEater()
        {
            eventSubscriber.on_completed();
        }
    }
}
```



Из библиотеки RxCpp подключается заголовочный файл `rxcpp/rx.hpp`, в котором содержатся объявления классов `subscriber` и `observable`, а из библиотеки Qt нужен класс `QEvent`, объявленный в одноимённом файле. Всё, что объявлено в файле `rx_eventfilter.h`, помещается в пространстве имён `rxevt`. Класс `EventEater`

ег фильтрует события, передавая на дальнейшую обработку только те, которые имеют определённый тип, заданный при инициализации объекта, и игнорируя все остальные. Для этого служат два поля этого класса. Во-первых, это поле `eventSubscriber` – реактивный подписчик, обрабатывающий данные типа `QEvent`. Во-вторых, поле `eventType`, в котором хранится тип интересующих подписчика событий.

В конструкторе класса `EventEater` происходит обращение к конструктору базового класса – ему передаётся указатель на объект-владелец, чьи события данный фильтр будет обрабатывать. Полям `eventSubscriber` и `eventType` присваиваются начальные значения: реактивный подписчик, ответственный за дальнейшую обработку событий, и тип события, который нужно отбирать.

```
bool eventFilter(QObject* obj, QEvent* event)
{
    if(event->type() == eventType)
    {
        eventSubscriber.on_next(event);
    }

    return QObject::eventFilter(obj, event);
}
```

Эта функция, определённая в базовом классе `QObject`, переопределена таким образом, что отправляет событие реактивному подписчику в том случае, если его тип совпадает с указанным при инициализации. Объект класса `EventEater` получает на вход через функцию `eventFilter` события любых типов. Эта функция может запретить дальнейшую обработку события или отправить событие объекту-адресату. Она возвращает значение «истина», если событие должно быть отброшено, значение «ложь» означает разрешение на дальнейшую обработку.

```
private:
    QEvent::Type eventType;
    rxcpp::subscriber<QEvent*> eventSubscriber;
};
```

Теперь напомним вспомогательную функцию, которая позволяет любой объект из каркаса Qt преобразовать в наблюдаемый источник из библиотеки `RxCpp`.

```
// функция-фабрика, создающая наблюдаемый источник событий
rxcpp::observable<QEvent*> from(
    QObject* qobject,
    QEvent::Type type)
{
    if(!qobject) return rxcpp::sources::never<QEvent*>();

    return rxcpp::observable<>::create<QEvent*>(
        [qobject, type](rxcpp::subscriber<QEvent*> s) {
            qobject->installEventFilter(
```

```

        new EventEater(qobject, type, s));
    }
};
}

} // rxevt

#endif // RX_EVENTFILTER_H

```

Эта функция работает следующим образом. Механизм обработки сигналов в библиотеке Qt позволяет назначить (с помощью метода `installEventFilter`) некоторый объект (производный от класса `QObject`) предварительным фильтром событий, адресованных другому объекту (также производному от класса `QObject`). Это весьма мощный инструмент. В нашем случае в качестве объекта-фильтра используется объект нашего класса `EventEater`. Таким образом, функция `from`, получая на вход указатель `qobject` и интересующий тип событий, создаёт наблюдаемый источник, который всякий раз при подключении наблюдателя создаёт объект-прослойку и устанавливает её фильтром событий для объекта `qobject`.

Создание окна и размещение его элементов

Напишем теперь код, который создаёт окно с двумя расположенными на нём визуальными элементами. Один из них будет следить за перемещением мыши, как в предыдущем примере, а другой – отображать сведения о событиях, приходящих от мыши, и её текущих координатах.

Модуль `main.cpp` удобно разделить на две части и рассматривать их по отдельности. Начнём с той из них, которая отвечает за создание и размещение визуальных объектов.

```

#include <QApplication>
#include <QLabel>
#include <QWidget>
#include <QVBoxLayout>
#include <QMouseEvent>
#include "rx_eventfilter.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    // создать главное окно
    auto widget = std::unique_ptr<QWidget>(new QWidget());
    widget->resize(280,200);

    // создать и настроить область для отслеживания мыши
    auto label_mouseArea = new QLabel("Площадка для мыши");
    label_mouseArea->setMouseTracking(true);
    label_mouseArea->setAlignment(
        Qt::AlignCenter|Qt::AlignHCenter);
    label_mouseArea->setFrameStyle(2);

    // создать и настроить область для отображения сообщений

```

```
auto label_coordinates = new QLabel("X = 0, Y = 0");
label_coordinates->setAlignment(
    Qt::AlignCenter|Qt::AlignHCenter);
label_coordinates->setFrameStyle(2);
```

Заголовочный файл `gx_eventfilter.h` был рассмотрен ранее – в нём находится определение класса-прослойки, который перехватывает события, адресованные какому-либо объекту, производному от класса `QObject`, и выступает наблюдаемым источником, который можно сопрягать с другими средствами библиотеки `RxCpp`. В отличие от предыдущего примера, в этой программе не создаётся собственный класс диалогового окна – вместо этого просто создаётся объект библиотечного класса `QWidget`, затем на нём размещаются два визуальных элемента типа `QLabel` (статический текст), чьим размещением управляет вертикальный контейнер. Здесь же устанавливается начальный размер главного окна. Кроме того, как и в предыдущем примере, для визуального элемента, играющего роль площадки для мыши, устанавливается режим перехвата событий.

```
// настроить растяжимость элементов в контейнере
label_mouseArea->setSizePolicy(
    QSizePolicy::Expanding, QSizePolicy::Expanding);
label_coordinates->setSizePolicy(
    QSizePolicy::Expanding, QSizePolicy::Expanding);

auto layout = new QVBoxLayout;
layout->addWidget(label_mouseArea);
layout->addWidget(label_coordinates);
layout->setStretch(0, 4);
layout->setStretch(1, 1);
widget->setLayout(layout);
```

Здесь устанавливается растягивающая политика управления размером для обоих элементов окна по обоим координатам. Однако площадка для мыши и область сообщений обладают различной жёсткостью при растяжении.

Наблюдатели для различных типов событий

Вторая часть функции `main` осуществляет подписку реактивных наблюдателей на три типа событий от мыши: события перемещения, нажатия и двойного щелчка.

```
// событие перемещения
rxevt::from(label_mouseArea, QEvent::MouseMove)
    .subscribe([&label_coordinates](const QEvent* e) {
        auto me = static_cast<const QMouseEvent*>(e);
        label_coordinates->setText(
            QString("Перемещение: X = %1, Y = %2")
                .arg(me->x())
                .arg(me->y()));
    });
```

Функция `rxevt::from` была создана нами и подробно рассмотрена выше. На основе объекта из системы Qt она создаёт реактивный наблюдаемый источник событий одного определённого типа, заданного вторым аргументом. Конкретно в этом фрагменте в реактивный источник преобразуется визуальный объект `label_mouseArea`, при этом отбираются исключительно события типа `QEvent::MouseMove`. На получившийся наблюдаемый источник подписывается функция-наблюдатель, которая меняет текст в области сообщений: теперь там будет отображено, что последнее событие – это перемещение, и текущие координаты мыши.

```
// событие щелчка
rxevt::from(label_mouseArea, QEvent::MouseButtonPress)
.subscribe([&label_coordinates](const QEvent* e) {
    auto me = static_cast<const QMouseEvent*>(e);
    label_coordinates->setText(
        QString("Нажатие X = %1, Y = %2")
            .arg(me->x())
            .arg(me->y()));
});
```

Этот фрагмент во всём подобен предыдущему, за исключением того, что для второго реактивного источника отбираются события типа `QEvent::MouseButtonPress`. Обработчик выводит текст о том, что произошедшее событие есть нажатие, и координаты, в которых оно произошло.

```
// событие двойного щелчка
rxevt::from(label_mouseArea, QEvent::MouseButtonDblClick)
.subscribe([&label_coordinates](const QEvent* e) {
    auto me = static_cast<const QMouseEvent*>(e);
    label_coordinates->setText(
        QString("Двойной щелчок: X = %1, Y = %2")
            .arg(me->x())
            .arg(me->y()));
});
```

```
widget->show();
return app.exec();
```

```
}
```

События типа `QEvent::MouseButtonDblClick` (двойной щелчок) обрабатываются по той же схеме, что и два предыдущих, в обработчике отличается лишь текст сообщения. Завершают функцию `main` отображение главного окна посредством функции `show` и запуск главного цикла обработки событий.

Файл проекта `Mouse_EventFilter.pro`, необходимый для сборки приложения, показан ниже.

```
QT += core widgets
CONFIG += c++14
TARGET = Mouse_EventFilter
INCLUDEPATH += include
```



```
SOURCES += \  
    main.cpp  
HEADERS += \  
    rx_eventfilter.h
```

Библиотека RxCpp состоит исключительно из заголовочных файлов. В директории проекта создана поддиректория `include`, в которую скопированы файлы этой библиотеки. Эта поддиректория добавлена к списку путей `INCLUDE-PATH`, который задаёт для компилятора места для заголовочных файлов. Теперь у читателя есть всё, чтобы собрать и запустить приложение.

Знакомство с библиотекой RxQt

Библиотека RxQt – это библиотека с открытым исходным кодом, написанная поверх библиотеки RxCpp. Её задача состоит в том, чтобы упростить обработку событий и сигналов из каркаса Qt средствами реактивного программирования. Чтобы понять, как устроена эта библиотека, рассмотрим в качестве примера ещё одно приложение, которое тоже следит за событиями от мыши и фильтрует их, но на этот раз для фильтрации будут использоваться средства реактивной библиотеки. Исходный код библиотеки RxQt можно загрузить из репозитория <https://github.com/tetsurom/rxqt>.

```
#include <QApplication>  
#include <QLabel>  
#include <QMouseEvent>  
#include "gravity_qlabel.h"  
#include "rxqt.hpp"  
  
int main(int argc, char *argv[]) {  
    QApplication app(argc, argv);  
  
    auto widget = new QWidget();  
    widget->resize(350, 300);  
    widget->setCursor(Qt::OpenHandCursor);  
  
    auto xDock = new QLabel(widget);  
    xDock->setStyleSheet("QLabel {background-color: red}");  
    xDock->resize(9,9);  
    xDock->setGeometry(0, 0, 9, 9);  
  
    auto yDock = new QLabel(widget);  
    yDock->setStyleSheet("QLabel {background-color: blue}");  
    yDock->resize(9,9);  
    yDock->setGeometry(0, 0, 9, 9);
```



Этот блок кода создаёт главное окно приложения – объект класса `QWidget`. В этом окне размещаются два графических объекта типа `QLabel`, которые играют в приложении роль отметок на координатных осях. Одна метка, окрашенная в красный цвет, должна перемещаться вдоль верхней границы окна и отмечать координату *X*, а другая, синяя, – вдоль левой границы (координата *Y*).

```

rxqt::from_event(widget, QEvent::MouseButtonPress)
.filter([](const QEvent* e) {
    auto me = static_cast<const QMouseEvent*>(e);
    return (Qt::LeftButton == me->buttons());
})
.subscribe([&](const QEvent* e) {
    auto me = static_cast<const QMouseEvent*>(e);
    widget->setCursor(Qt::ClosedHandCursor);
    xDock->move(me->x(), 0);
    yDock->move(0, me->y());
});

```

В этом фрагменте кода функция `rxqt::from_event` создаёт реактивный наблюдаемый источник, т. е. объект типа `rxcpp::observable<QEvent*>`. Из всех событий, относящихся к объекту `widget` (т. е. главному окну приложения), сам источник отбирает лишь нажатия кнопки мыши и игнорирует остальные события. Затем к нему применяется операция фильтрации, которая оставляет в потоке событий нажатия только левой кнопки. Наконец, на события из этого источника подписывается в качестве наблюдателя лямбда-функция, которая меняет вид указателя мыши на хватающую ладонь (за это отвечает метод `setCursor` с аргументом `Qt::ClosedHandCursor`), а также устанавливает координаты двух меток. Новое значение координаты *X* метки `xDock` устанавливается равным текущей координате *X* указателя мыши, а координата *Y* этой метки остаётся нулевой. Тем самым данная метка отмечает проекцию указателя мыши на верхнюю границу окна. Подобным же образом для объекта `yDock` устанавливается значение координаты *Y*, как у указателя мыши: этот объект отмечает проекцию указателя на левую границу окна.

```

rxqt::from_event(widget, QEvent::MouseMove)
.filter([](const QEvent* e) {
    auto me = static_cast<const QMouseEvent*>(e);
    return (Qt::LeftButton == me->buttons());
})
.subscribe([&](const QEvent* e) {
    auto me = static_cast<const QMouseEvent*>(e);
    xDock->move(me->x(), 0);
    yDock->move(0, me->y());
});

```

В этом блоке кода таким же способом, как и в предыдущем фрагменте, создаётся наблюдаемый источник данных средствами библиотеки `RxQt`. Источник представляет собой поток событий типа перемещения мыши, при этом фильтр отбирает только перемещения при нажатой левой кнопке. Лямбда-функция, подписанная на события из этого потока, делает то же, что и подписчик из предыдущего фрагмента, – поддерживает в актуальном состоянии проекции мышиного указателя на координатные оси.

```

rxqt::from_event(widget, QEvent::MouseButtonRelease)
    .subscribe([&widget](const QEvent* e) {
        widget->setCursor(Qt::OpenHandCursor);
    });

```

```
});

widget->show();
return app.exec();
}
```

Ещё один наблюдаемый источник отбирает события отпускания кнопки мыши. Его подписчик меняет вид указателя мыши на раскрытую ладонь. Далее следуют две привычные строки: отображение главного окна и запуск главного цикла обработки событий.

Чтобы сделать наше приложение более забавным, добавим на окно ещё одну плавающую метку, похожую на объекты `xDock` и `yDock`. При нажатой кнопке мыши эта метка будет следовать за указателем, словно привязанная к нему силой притяжения. Сначала объявим для такой притягивающейся метки отдельный класс¹.

```
#ifndef GRAVITY_QLABEL_H
#define GRAVITY_QLABEL_H

#include <QLabel>

class Gravity_QLabel: public QLabel {
public:
    explicit Gravity_QLabel(QWidget *parent = nullptr):
        QLabel(parent), prev_x(0), prev_y(0)
    {}

    int prev_x, prev_y;
};

#endif // GRAVITY_QLABEL_H
```

Теперь нужно вернуться к функции `main` и создать в главном окне визуальный объект – экземпляр только что объявленного класса `Gravity_QLabel`.

```
auto gravityDock = new Gravity_QLabel(widget);
gravityDock->setStyleSheet(
    "QLabel {background-color: green}");
gravityDock->resize(9, 9);
gravityDock->setGeometry(0, 0, 9, 9);
```

Всякий раз, когда происходит нажатие на левую кнопку мыши, объекту `gravityDock` нужно устанавливать координаты, совпадающие с текущими координатами мыши. Для этого в первую из трёх лямбда-функций, отвечающую за обработку события `QEvent::MouseButtonPress`, нужно добавить новую строку:

```
gravityDock->move(me->x(), me->y());
```

¹ Логика этого класса весьма неудачна: открытое для всех клиентов состояние и полное отсутствие собственного поведения противоречат самой идее класса. – *Прим. перев.*

При перемещении мыши с нажатой левой кнопкой нужно менять координаты объекта `gravityDock` так, чтобы он следовал за указателем мыши, стремясь его догнать. Для этого в лямбда-функцию, обрабатывающую события `QEvent::MouseMove`, добавим строки:

```
gravityDock->prev_x = gravityDock->prev_x * .96
+ me->x() * .04;
gravityDock->prev_y = gravityDock->prev_y * .96
+ me->y() * .04;
gravityDock->move(
    gravityDock->prev_x,
    gravityDock->prev_y);
```

Согласно этому алгоритму, новое положение метки `gravityDock` состоит на 96 % из её предыдущего положения и на 4 % из координат мыши.

Теперь приложение можно собрать, запустить и убедиться, что метки следуют, каждая своим способом, за координатами мышиного указателя. Тем самым закончена разработка приложения, демонстрирующего взаимодействие механизма событий из каркаса `Qt` с реактивными средствами библиотеки `RxCpp` через посредство библиотеки `RxQt`.

Итоги

В этой главе рассматривались вопросы реактивного программирования графических пользовательских интерфейсов с использованием каркаса `Qt`. Открывал главу краткий обзор возможностей, предоставляемых данным каркасом для разработки графических приложений. Далее следовало изложение основных понятий каркаса `Qt`: важнейших классов, системы метаобъектов, механизма сигналов и слотов. На основе этих сведений было создано простейшее приложение «Здравствуй, мир» с текстовым полем в качестве главного окна. Затем был рассмотрен пример приложения, которое следило за положением указателя мыши. Этот пример позволил лучше понять средства обработки событий и использование механизма сигналов и слотов для общения объектов между собой. После этого был разобран пример ещё одного приложения, обрабатывающего события от мыши, но на этот раз использующего средства библиотеки `RxCpp` для подписки на события и их фильтрации. Тем самым было показано, как использовать библиотеку `RxCpp` в среде графического интерфейса, чтобы приложение в целом отвечало реактивной модели программирования. В заключительном разделе рассматривалась библиотека `RxQt` с открытым исходным кодом, предназначенная для интеграции библиотек `Qt` и `RxCpp`.

.....

Шаблоны и идиомы реактивного программирования на языке C++

В предыдущих главах было немало рассказано о том, как применять реактивную модель программирования при разработке на языке C++. Была изучена библиотека RxCpp и присущая ей модель программирования, основные элементы этой библиотеки и реактивное программирование пользовательских интерфейсов.

В этой главе будут рассмотрены следующие вопросы:

- общие сведения о шаблонах проектирования и их внедрении в практику разработки;
- связь шаблонов проектирования с идиомой реактивного программирования;
- некоторые шаблоны и идиомы реактивного программирования.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ И ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Идеи объектно-ориентированного программирования (ООП) достигли критической массы в начале 1990-х годов благодаря широкому распространению хороших компиляторов языка C++. Многие программисты начала 90-х силились понять, что такое ООП и как на практике применить его в больших проектах. До появления всеохватывающей среды обмена знаниями, которой позднее стал интернет, это было и впрямь нелегко. Первые энтузиасты ООП писали технические отчёты, публиковали статьи в академических и популярных журналах и проводили семинары, на которых распространяли идеи ООП. Ведущие

журналы для программистов, такие как Dr. Dobb's Journal и C++ Report, завели у себя регулярные колонки, посвящённые ООП.

Возникла острая необходимость в передаче знания от знатоков к всё расширяющемуся сообществу программистов, но именно этого и не происходило. Легендарный немецкий математик Карл Фридрих Гаусс советовал всегда учиться у мастеров. Хотя Гаусс имел в виду прежде всего математику, этот принцип вполне справедлив и для любой достаточно сложной области человеческой деятельности. Однако в случае программирования мастеров было всё ещё слишком мало, и модель обучения через наставничество дала сбой.

i Джеймс Коплиен (James Coplien) опубликовал в 1991 г. книгу под названием «Advanced C++ Programming Styles and Idioms»¹, в которой речь шла о шаблонах нижнего уровня (идиомах), связанных с языком программирования C++. Хотя сегодня на эту работу не очень часто ссылаются, авторы считают её выдающейся книгой и превосходным каталогом лучших практических приёмов объектно-ориентированного программирования.

Эрих Гамма начал работу над каталогом шаблонов проектирования в рамках своей диссертации, вдохновившись идеями архитектора-градостроителя Кристофера Александера. В знаменитой работе Александера (в соавторстве с Сарой Исикава и Мюрием Сильверштейном) было замечено, что при проектировании зданий и городских районов часто повторяются одни и те же шаблоны. Если выделить эти шаблоны, описать их в виде каталога и сделать всеобщим достоянием, можно значительно облегчить работу градостроителей – ведь теперь они могут вести проектирование, оперируя целыми шаблонами. Именно эта идея впечатлила Эриха Гамму. Вскоре ещё трое специалистов, пришедших к подобным идеям, Ральф Джонсон, Джон Влissидес и Ричард Хелм, объединили свои усилия с Гаммой и создали каталог из двадцати трёх шаблонов проектирования, ныне ласково именуемый каталогом «Банды четырёх». Их книга о шаблонах объектно-ориентированного программирования² вышла в издательстве Addison Wesley в 1994 г., сразу стала исключительно популярна в среде программистов и дала мощный толчок шаблонно-ориентированной разработке программ. Каталог «Банды четырёх» охватывал шаблоны, возникающие на этапе проектирования программ.

В 1996 г. группа инженеров из компании Siemens опубликовала труд, озаглавленный «Шаблонно-ориентированная архитектура программных систем» (Pattern-Oriented Software Architecture, POSA), в центре внимания которого находились архитектурные аспекты построения программных систем. Каталог шаблонов POSA был изложен в серии из пяти книг, вышедших в издательстве John Wiley and Sons. Впоследствии к коллективу авторов присоединился Дуглас Шмидт, создатель библиотеки ACE (Adaptive Communication Environment – адаптивная среда обмена данными) для поддержки сетевого взаимодействия

¹ Коплиен Дж. Программирование на C++. Классика CS. СПб.: Питер, 2005. 479 с.

² Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влissидес. СПб.: Питер, 2014. 366 с.

и системы ТАО (сокращение от «The ACE ORB», т. е. «ORB для библиотеки ACE», где ORB – в свою очередь, сокращение от Object Request Broker – объектный брокер запросов). Он же позднее стал главой консорциума OMG (Object Management Group), занимающегося разработкой и продвижением объектно-ориентированных технологий и стандартов, таких как технология CORBA (Common Object Request Broker Architecture – общая архитектура объектных брокеров запросов) и язык UML (Unified Modelling Language – унифицированный язык моделирования).

За этими двумя прорывами последовала лавина других работ. Наиболее значимыми стали следующие каталоги шаблонов:

- «Шаблоны корпоративных приложений» Мартина Фаулера с соавторами;
- «Шаблоны интеграции корпоративных приложений» Грегора Хоупа и Бобби Вульфа;
- «Основные паттерны J2EE» Дипака Алера;
- «Проблемно-ориентированное проектирование» Эрика Эванса;
- «Корпоративные шаблоны и архитектуры, управляемые моделями» Джима Арлоу и Иллы Нойштадта.

Эти книги, весьма примечательные сами по себе, имеют заметный уклон в область разработки корпоративных программных систем, переживавшую в ту пору свой расцвет. Для разработчиков на языке C++ наиболее важными были и остаются каталоги «Банды четырёх» и POSA.

ОСНОВНЫЕ КАТАЛОГИ ШАБЛОНОВ

Шаблон называют типовую, пригодную для многократного применения архитектурную конструкцию, составляющую решение некоторой проблемы, часто возникающую при разработке программных систем. Для удобства шаблонам дают имена. Чаще всего шаблоны систематизируют в своего рода репозиториях или каталогах. Некоторые из них публикуются в виде книг. Самый знаменитый и широко используемый из них – каталог «Банды четырёх».

Шаблоны «Банды четырёх»

«Банда четырёх», как называют в шутку коллектив авторов, дала первый толчок движению за внедрение шаблонов в практику разработки программ. В центре внимания этой четвёрки находились архитектура и проектирование объектно-ориентированных программных систем. Идеи Кристофера Алксандера хорошо подошли и для индустрии программного обеспечения и нашли применение в областях архитектуры приложений, параллельного программирования, информационной безопасности и во многих других. «Банда четырёх» разделила свой каталог на три раздела: структурные, порождающие и поведенческие шаблоны. В их работе для пояснения идеи шаблонов исполь-



зованы языки C++ и Smalltalk. Однако описанные там шаблоны перенесены на большинство существующих ныне языков программирования. Рассмотрим следующую таблицу.

№	Область	Шаблоны
1	Порождение	Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка
2	Структура	Адаптер, Мост, Композит, Декоратор, Фасад, Приспособленец, Заместитель
3	Поведение	Цепочка обязанностей, Команда, Интерпретатор, Итератор, Посредник, Хранитель, Наблюдатель, Состояние, Стратегия, Шаблонный метод, Посетитель

Авторы уверены, что основательное знакомство с шаблонами «Банды четырёх» необходимо любому программисту. Эти шаблоны встречаются повсюду, независимо от предметной области. Они позволяют рассуждать и общаться об устройстве программной системы, абстрагируясь от конкретного языка. Шаблоны «Банды четырёх» широко применяются в обособленных мирах, сложившихся вокруг платформы .NET и языка Java. Разработчики каркаса Qt тоже широко использовали эти шаблоны для создания интуитивно удобной модели программирования.

Каталог POSA

Фундаментальный пятитомный труд «Шаблонно-ориентированная архитектура программных систем» (Pattern-Oriented Software Architecture, POSA), в котором рассматриваются специализированные шаблоны для разработки критически важных подсистем, оказал значительное влияние на разработку ПО. Этот каталог особенно полезен для разработчиков, занимающихся такими особо важными подсистемами крупных программных систем, как механизмы хранения баз данных, механизмы коммуникации для распределённых систем, связующее ПО (middleware) и т. д. Особенность данного каталога шаблонов состоит в том, что он особенно хорошо подходит для программирования на языке C++. Перечень шаблонов, вошедших в пять вышедших томов, представлен в следующей таблице.



№	Область	Шаблоны
1	Архитектура	Слои, Конвейеры и фильтры, Доска объявлений, Брокер, Модель – представление – контроллер (model-view-controller, MVC), Представление – абстракция – контроллер, Микроядро, Рефлексия
2	Проект	Целое – часть, Главный – дублёр, Заместитель, Обработчик команд, Обработчик представлений, Передатчик – приёмник, Клиент – диспетчер – сервер, Издатель – подписчик
3	Доступ и конфигурация	Фасад обёртки, Конфигуратор компонентов, Перехватчик, Интерфейс расширения
4	Обработка событий	Реактор, Проактор, Признак асинхронного завершения, Приёмщик – соединитель
5	Синхронизация	Блокировка в области видимости, Блокировка со стратегией, Потокобезопасный интерфейс, Блокировка с двойной проверкой

№	Область	Шаблоны
6	Параллельное программирование	Активный объект, Следящий объект, Синхронно-асинхронная работа, Ведущий–ведомый, Память потока
7	Доступ к ресурсам	Искатель, Ленивый доступ, Жадный доступ, Частичный доступ
8	Жизненный цикл ресурсов	Кеширование, Фонд ресурсов, Координатор, Менеджер жизненного цикла ресурсов
9	Освобождение ресурсов	Лизинг, Изгнание
10	Распределённые вычисления	Подборка шаблонов из различных каталогов в контексте распределённого программирования
11	Методология	В отдельный том вынесены общие вопросы метауровня, касающиеся понятия шаблона в общем виде, языков шаблонов и их использования

Каталог POSA стоит внимательно изучать ради глубокого понимания основополагающих принципов, на которых зиждется архитектура крупных программных систем, простирающихся по всему земному шару. По мнению авторов, этот каталог получил гораздо меньше внимания, чем заслужил.

ЕЩЁ РАЗ О ШАБЛОНАХ ПРОЕКТИРОВАНИЯ

Связь между шаблонами «Банды четырёх» и реактивным программированием глубже, чем может показаться на первый взгляд. Шаблоны проектирования из каталога «Банды четырёх» относятся главным образом к объектно-ориентированной парадигме. Реактивное же программирование в большей степени связано с функциональной парадигмой, потоками данных и параллельной обработкой. Как уже говорилось ранее, реактивное программирование позволяет восполнить некоторые недостатки классических шаблонов «Банды четырёх» – в частности, об этом шла речь в начале главы 5.

Объектно-ориентированный подход к разработке программ направлен в основном на моделирование иерархий. Например, шаблон «Композит» представляет собой удобное средство для моделирования иерархических отношений вида «целое–часть». Всюду, где присутствует композит, можно ожидать и набор объектов-посетителей, работающих с ним сообща – посетители позволяют единообразно обрабатывать иерархические отношения вида «общее–частное». Иными словами, пара шаблонов «Композит–посетитель» может служить каноническим инструментом для создания объектно-ориентированных систем.

Реализация посетителя должна обладать некоторым знанием о структуре композита¹. Обработка сложных структур данных на основе шаблона «Посе-

¹ Это утверждение, настойчиво повторяемое авторами (см. главу 5), по меньшей мере, спорно. Знание о внутренней структуре композита может быть инкапсулировано в самом композите – в частности, метод `AcceptVisitor` класса `Composite` может отвечать за поочерёдный обход всех своих подобъектов объектом-посетителем. Иной возможный подход состоит в том, чтобы интерфейс объекта-композиции предоставлял способ обхода своих подобъектов: например, итератор по подобъектам или метод `for_each` для применения функции-обработчика к каждому подобъекту. В обоих случаях посетитель взаимодействует с композитом через интерфейс, абстрагирующий от внутреннего устройства композита. – *Прим. перев.*

титель» чрезмерно усложняется по мере того, как растёт число различных посетителей. Появление в системе преобразователей и фильтров ещё более запутывает картину.

На помощь приходит шаблон «Итератор», который хорошо подходит для поэлементного прохода по разного рода последовательностям, потокам, спискам элементов. Сочетая средства объектно-ориентированного и функционального программирования, можно легко фильтровать и преобразовывать последовательности. Разработанная корпорацией Microsoft технология встроенного в язык программирования языка запросов (Language-integrated query, LINQ) и технология обработки потоков, появившаяся в 8-й версии языка Java, служат отличными примерами мощи итераторов.

Но как же быть с преобразованием иерархических структур данных в линейные последовательности элементов? Большую часть иерархий можно «разгладить» в поток элементов для дальнейшей обработки. Всё это вместе взятое приводит к следующей логике обработки данных:

- моделировать иерархию данных с помощью шаблона «Композит»;
- разгладить иерархию в линейную последовательность, используя шаблон «Посетитель»;
- для обхода получившейся последовательности использовать шаблон «Итератор»;
- применить к элементам последовательности комбинацию фильтров и преобразователей¹, полученные результаты направить для дальнейшей обработки.

Описанный выше подход называют втягивающим. Потребители, или клиенты, сами осуществляют запрос к источнику данных (или событий), тем самым втягивая в себя из источника элемент за элементом. Эта схема обладает следующими недостатками:

- втягивание клиентом данных, которые впоследствии окажутся ему не нужны;
- преобразователи и фильтры применяются к данным на стороне приёмника, а не источника;
- приёмник данных (клиент) может заблокировать источник данных (сервер);
- данный подход плохо подходит для асинхронной обработки².

Все эти трудности можно преодолеть, просто перевернув архитектуру вверх ногами: пусть теперь сервер асинхронно вталкивает свои данные в поток, а приёмник данных лишь реагирует на новые данные, поступающие из пото-

¹ Для этой цели хорошо подходит шаблон «Декоратор». Так, для фильтрации последовательности данных нужно на итератор навесить декоратор, который пропускает элементы, не удовлетворяющие предикату. – *Прим. перев.*

² Отнюдь не бесспорное утверждение. Скажем, в стандарте C++ 17 имеются десятки алгоритмов параллельной обработки последовательностей на основе итераторов. – *Прим. перев.*

ка. При таком устройстве системы становится легко разместить фильтрацию и преобразование данных на стороне сервера, т. е. источника данных. Отсюда же автоматически получаем возможность обрабатывать на стороне клиента только те данные, которые ему действительно нужны: клиент может просто игнорировать ненужные элементы потока. В целом схема вталкивания данных выглядит следующим образом:

- данные трактуются как потоки элементов и как наблюдаемые источники;
- к потокам данных можно применять операции поэлементного преобразования и фильтрации, в том числе операции высшего порядка (т. е. принимающие функции в качестве аргументов);
- операция всегда принимает на вход наблюдаемый источник и выдаёт в качестве результата новый наблюдаемый источник;
- клиент может подписаться на оповещения от какого-либо наблюдаемого источника;
- наблюдатели обладают стандартизированными механизмами обработки сообщений.

Таким образом, налицо тесная связь между шаблонами объектно-ориентированного проектирования и реактивным программированием. Если при сочетании этих двух парадигм руководствоваться здравым смыслом, можно получить высококачественный, хорошо расширяемый и поддерживаемый программный код. В предыдущих главах читатель изучил, как видоизменить объектно-ориентированные шаблоны «Композит» и «Посетитель», чтобы разгладить иерархическую структуру данных и тем самым воспользоваться шаблоном «Итератор». Было рассмотрено, как усовершенствовать привычную схему итеративной обработки последовательностей одним небольшим изменением: сделав так, чтобы, испустив событие, источник мог сразу о нём забыть. В следующем разделе эти принципы будут проиллюстрированы написанием конкретных примеров кода.



ОТ ШАБЛОНОВ ПРОЕКТИРОВАНИЯ

К РЕАКТИВНОМУ ПРОГРАММИРОВАНИЮ

Несмотря на то что движение за внедрение шаблонов в практику программирования в основном связано с объектно-ориентированной парадигмой, а реактивное программирование основано на функциональном подходе, между ними довольно много общего. В предыдущей главе говорилось, что:

- объектно-ориентированная парадигма хорошо подходит для моделирования структурных аспектов систем;
- парадигма функционального программирования хороша для моделирования поведенческих аспектов систем.

Чтобы лучше продемонстрировать связь между объектно-ориентированным и реактивным программированием, напомним программу, которая проходит по всем файлам и поддиректориям заданной директории.

Для обработки дерева файловой системы понадобится композитная структура с узлами двух типов (унаследованных от общего базового класса `EntryNode`):

- класс `FileNode`, представляющий информацию о файле;
- класс `DirectoryNode`, представляющий информацию о директории.

Для обработки иерархий, содержащих элементы этих двух типов, понадобятся различные посетители. В данном примере их будет два:

- посетитель для печати имён файлов и директорий;
- посетитель для преобразования композитной иерархии в линейный список имён.

Погрузимся же без лишних слов в решение задачи. Рассмотрим следующий код:

```
//----- DirReact.cpp
#include <rxcpp/rx.hpp>
#include <memory>
#include <map>
#include <algorithm>
#include <string>
#include <vector>
// удалить в системах POSIX
#include <windows.h>
#include <functional>
#include <thread>
#include <future>
using namespace std;
// Упреждающие объявления:
// модель дерева директорий и файлов
class FileNode;
class DirectoryNode;
// интерфейс посетителя
class IFileFolderVisitor;
```



Упреждающие объявления подсказывают компилятору, что такими-то именами можно пользоваться как именами классов, однако определение этим классам будет дано в другом месте программы. Это позволяет сократить объём кода. Объект класса `FileNode` будет хранить в себе имя файла и его размер. Объект класса `DirectoryNode` содержит имя директории и список объектов, представляющих файлы и поддиректории этой директории. Для обработки иерархии объектов `FileNode` и `DirectoryNode` предназначен интерфейс `IFileFolderVisitor`. Ниже показаны дальнейшие объявления:

```
// информация о файле
struct FileInformation {
    string name;
    long size;
    FileInformation(string const& pname, long psize)
        : name(pname), size(psize)
    {}
};
```

```
// базовый для классов узлов иерархии
class EntryNode {
protected:
    string name;
    int isdir;
    long size;
public:
    virtual bool Isdir() = 0;
    virtual long getSize() = 0;
    virtual void Accept(IFileFolderVisitor& ivis) = 0;
    virtual ~EntryNode() {}
};
```

Чтобы композит можно было наполнять разнородными объектами, нужен общий базовый класс для всех конкретных узлов иерархии. В данном примере это класс `EntryNode`¹. В нём хранятся имя, размер файла и признак директории. Помимо трёх виртуальных функций, которые должны быть реализованы в подклассах, в нём есть и виртуальный деструктор. Виртуальность гарантирует вызов правильного деструктора при уничтожении полиморфного объекта.

```
// интерфейс посетителя
class IFileFolderVisitor{
public:
    virtual void Visit(FileNode& fn )=0;
    virtual void Visit(DirectoryNode& dn )=0;
    virtual IFileFolderVisitor() {}
};
```

Когда структура данных имеет вид иерархического композита, есть смысл организовать обработку такой иерархии посредством посетителя. Для каждого типа узла, который может входить в иерархию, в интерфейсе посетителя должен быть свой метод `visit`. Кроме того, в каждом классе узла иерархии должен быть свой метод `ассепт`, который принимает объект-посетитель в качестве аргумента и делегирует работу тому его методу `visit`, который соответствует фактическому типу данного узла. Этот приём называется **двойной диспетчеризацией** вызовов.

```
// Узел, соответствующий файлу
class FileNode : public EntryNode {
public:
    FileNode(string pname, long psize) {
        isdir = 0;
```

¹ Как архитектура, так и качество кода, предложенного авторами, далеко от идеального. Параметр «размер» имеет смысл лишь для файлов, но не для директорий, поэтому нет смысла выносить его на уровень базового класса `EntryNode`. Далее, при наличии в базовом классе поля `isdir` (которое следовало бы сделать константным) нет смысла делать метод `Isdir` (который также следовало бы объявить со спецификатором `const`) виртуальным. То же справедливо для поля `size` и метода `getSize`. – *Прим. перев.*

```

        name = pname;
        size = psize;
    }
    ~FileNode() {
        cout << "Деструктор объекта FileNode " << name << endl;
    }
    virtual bool Isdir() { return isdir == 1; }
    string getname() { return name; }
    virtual long getSize() { return size; }
    virtual void Accept(IFileFolderVisitor& ivis) {
        ivis.Visit(*this);
    }
};

```

Конструктор класса FileNode сохраняет в поля объекта имя файла и его размер. В этом классе реализованы все чистые виртуальные методы, объявленные в базовом классе EntryNode. Метод ассерт делегирует вызов правильному методу посетителя.

```

// Узел, представляющий директорию
class DirectoryNode : public EntryNode {
    list<unique_ptr<EntryNode>> files;
public:
    DirectoryNode(string pname) {
        files.clear();
        isdir = 1;
        name = pname;
    }

    ~DirectoryNode() {
        files.clear();
    }

    list<unique_ptr<EntryNode>>& GetAllFiles() { return files; }

    bool AddFile(string pname , long size) {
        files.push_back(
            unique_ptr<EntryNode>(
                new FileNode(pname,size)));
        return true;
    }

    bool AddDirectory(DirectoryNode*dn) {
        files.push_back(unique_ptr<EntryNode>(dn));
        return true;
    }

    bool Isdir() { return isdir == 1; }
    string getname() { return name; }
    void setname(string pname) { name = pname; }
    long getSize() {return size; }
};

```

```

void Accept(IFileFolderVisitor& ivis) {
    ivis.Visit(*this);
}
};

```

Класс `DirectoryNode` моделирует директорию¹. В нём содержится список файлов и поддиректорий. Для хранения в одном списке объектов двух разных типов используется умный указатель на базовый класс. Конечно же, в этом классе реализованы все чистые виртуальные функции, объявленные в базовом классе `EntryNode`. Методы `AddFile` и `AddDirectory` служат для наполнения списка дочерних узлов. При обходе директории с использованием специфических функций конкретной ОС данный объект будет наполняться дочерними узлами через эти два метода. За это должен отвечать следующий класс.

¹ Следует обратить внимание читателя на совершенно неудовлетворительное качество кода, который предложен здесь в качестве примера для подражания. Выбор списка в качестве структуры данных для хранения дочерних узлов неудачен: согласно специфике предметной области, дочерние элементы не должны иметь одинаковых имён, тогда как список не гарантирует уникальности элементов. Уникальность автоматически обеспечивают ассоциативные контейнеры `std::map` и `std::unordered_map`. Как конструктор, так и деструктор содержат очистку списка дочерних узлов: `files.clear()`. Это излишне: на момент входа в тело конструктора список `files` уже создан посредством своего конструктора по умолчанию и поэтому гарантированно пуст. Симметричным образом после выполнения тела деструктора какого-либо объекта выполняется деструкция всех его подобъектов. Деструкция списка `files`, в свою очередь, предполагает деструкцию всех его элементов. Тем самым в конструкторе и деструкторе делается излишняя работа. Вместе с тем для другого поля не делается необходимая работа: неинициализированным остаётся поле `size`, унаследованное от базового класса, в этом поле находятся неинициализированные данные (т. н. «мусор»). Далее, легко убедиться, что реализация методов `Isdir`, `getname` и `getSize` в обоих порождённых классах (`FileNode` и `DirectoryNode`) одинакова, поэтому нет никакого смысла делать их виртуальными. Метод `GetAllFiles` нуждается в «исправлении имени»: он возвращает список не одних лишь файлов, а файлов и директорий, поэтому имя `GetChildren` подошло бы лучше. Ещё хуже, что этот метод возвращает неконстантную ссылку на поле `files`, тем самым позволяя клиенту портить состояние объекта. Строки в качестве аргументов (например, в конструктор и в метод `setname`) следовало бы передавать по константной ссылке или по ссылке `rvalue`, а никак не по значению. Возврат значения логического типа из методов `AddFile` и `AddDirectory` лишён всякого смысла: во-первых, в данной реализации эти методы всегда возвращают значение «истина», а во-вторых, даже теоретически невозможно представить себе ситуацию, когда добавление элемента в список может закончиться неудачей. В методе `AddFile` объект типа `FileNode` создаётся посредством операции `new`, а затем полученный «сырой» указатель передаётся в конструктор умного указателя `std::unique_ptr` – между тем азбучная истина состоит в том, что для этого следует применять функцию `std::make_unique`. Наконец, передача «сырого» указателя в метод `AddDirectory` с его дальнейшим оборачиванием в умный указатель недопустима абсолютно: ведь вызывающий контекст сохраняет в своём владении необёрнутый указатель и может либо уничтожить объект, либо обернуть его в ещё один умный указатель, что неминуемо приведёт к краху программы. – *Прим. перев.*

```
// вспомогательный класс,
// должен быть реализован отдельно для каждой ОС
class DirHelper {
public:
    static DirectoryNode *SearchDirectory(
        const std::string& refcstrRootDirectory)
    {
        // выполнить обход директории средствами ОС
        // и построить объект класса DirectoryNode
        return DirNode;
    }
};
```

Устройство класса DirHelper будет различно для системы Windows и систем стандарта POSIX (в частности, систем GNU Linux и macOS), поэтому его код в этой книге не приводится. Полностью исходный код можно найти на сайте данной книги. Этот класс должен всего лишь рекурсивно обойти директорию и наполнить объект класса DirectoryNode дочерними узлами.

```
// посетитель, который печатает содержимое директории
class PrintFolderVisitor : public IFileFolderVisitor {
public:
    void Visit(FileNode& fn) { cout << fn.getname() << endl; }
    void Visit(DirectoryNode& dn ) {
        cout << "In a directory " << dn.getname() << endl;
        list<unique_ptr<EntryNode>>& ls = dn.GetAllFiles();
        for (auto& itr : ls) { itr.get()->Accept(*this); }
    }
};
```

Класс PrintFolderVisitor представляет собой посетителя, который выводит на консоль информацию о файлах и директориях. Этот класс служит примером реализации посетителя для композитного объекта. В нашем случае композит содержит подобъекты двух типов, и реализация посетителя выходит довольно простой. Однако в случаях, когда типов узлов иерархии достаточно много, создание посетителей оказывается непростой задачей. Особенно сложным может оказаться создание фильтрующих и поэлементно преобразовывающих посетителей, в которых зачастую приходится применять уловки, специфичные для конкретных задач.

Рассмотрим функцию, которая приводит в действие всю систему. Её текст приведён ниже.

```
void TestVisitor(string directory) {
    // просканировать директорию, включая поддиректории
    DirectoryNode *dirs = DirHelper::SearchDirectory(directory);
    if (dirs == 0) { return; }
    PrintFolderVisitor *fs = new PrintFolderVisitor();
    dirs->Accept(*fs);
    delete fs;
    delete dirs;
}
```

Эта функция рекурсивно обходит дерево директорий, начиная с заданной, и строит композитный объект типа `DirectoryNode`. Затем для печати содержимого директории на консоль создаётся и запускается посетитель `PrintFolderVisitor`. Завершает пример главная функция:

```
int main(int argc, char *argv[]) {
    TestVisitor("D:\Java");
    return 0;
}
```

РАЗГЛАЖИВАНИЕ ИЕРАРХИИ И ЛИНЕЙНЫЙ ПРОХОД

Реализации посетителя нужно обладать некоторым знанием о внутренней структуре композита. Для некоторых композитов бывает нужно реализовать десятки посетителей. В особенности сложно бывает реализовать поэлементное преобразование и фильтрацию узлов посредством интерфейса посетителя. Каталог шаблонов «Банды четырёх» содержит шаблон «Итератор», который удобно использовать для прохода по линейной последовательности элементов. Вопрос теперь состоит в том, как превратить иерархическую структуру объектов в линейную, чтобы её можно было обработать с помощью итератора. Большую часть иерархий можно разгладить в список, последовательность или поток, применив для этого специальный объект-посетитель. Создадим такой посетитель для нашего примера. Его код представлен ниже.

```
// разгладить дерево директорий в линейный список
class FlattenVisitor : public IFileFolderVisitor {
    list<FileInformation> files;
    string CurrDir;
public:
    FlattenVisitor() { CurrDir = ""; }
    ~FlattenVisitor() { files.clear(); }
    list<FileInformation> GetAllFiles() { return files; }
    void Visit(FileNode& fn) {
        files.push_back(FileInformation {
            CurrDir + "\\" + fn.getname(),
            fn.getSize()
        });
    }

    void Visit(DirectoryNode& dn) {
        CurrDir = dn.getname();
        files.push_back(FileInformation(CurrDir, 0));
        list<unique_ptr<EntryNode>>& ls = dn.GetAllFiles();
        for (auto& itr : ls) { itr.get()->Accept(*this); }
    }
};
```



Класс `FlattenVisitor` строит список объектов, содержащих информацию о файлах и директориях¹. Для каждой директории перебираются все расположенные в ней файлы и поддиректории, и для каждой из них вызывается метод `accept` в соответствии с уже знакомой читателю схемой двойной диспетчеризации. Создадим теперь функцию, которая приводит в действие описанный здесь объект-посетитель и возвращает список объектов типа `FileInformation`, который можно обрабатывать с помощью итератора.

```
list<FileInformation> GetAllFiles(string dirname) {
    list<FileInformation> ret_val;
    // просканировать содержимое директории, включая поддиректории
    DirectoryNode *dirs = DirHelper::SearchDirectory(dirname);
    if (dirs == 0) { return ret_val; }
    FlattenVisitor *fs = new FlattenVisitor();
    dirs->Accept(*fs);
    ret_val = fs->GetAllFiles();
    delete fs;
    delete dirs;
    return ret_val;
}

int main(int argc, char *argv[]) {
    list<FileInformation> rs = GetAllFiles("D:\\Java");
    for( auto& as : rs )
        cout << as.name << endl;
    return 0;
}
```

Объект класса `FlattenVisitor` рекурсивно обходит иерархическую структуру, заключённую в объекте класса `DirectoryNode`, и наполняет список полными именами файлов. Разгладив иерархию в список, можно далее работать с ним посредством итератора.

Таким образом, мы разобрали, как моделировать иерархию объектов с помощью шаблона «Композит» и как в случае необходимости привести иерархическую структуру данных к линейному виду, удобному для применения шаблона «Итератор». В следующем разделе будет показано, как итератор, в свою очередь, может быть преобразован в наблюдаемый источник. Для реализации наблюдаемого источника будет использоваться библиотека `RxCpp`, при этом проталкивание объектов данных от источника к приёмнику будет осуществляться по принципу «выстрелил и забыл».

¹ Этот код совершенно некорректен. Как этот посетитель обработает полное имя файла `A\\B\\C\\file1.txt`? Ещё более интресная ошибка возникнет, если в некоторой директории `D` находятся на одном уровне поддиректория `E` и файл `file2.txt`, и посетитель сначала зайдёт в поддиректорию `E`, выйдет из неё и займётся файлом `file2.txt`. Какое полное имя он занесёт в список? Помимо того, код написан весьма неряшливо. Например, нет необходимости инициализировать поле `CurDir` пустой строкой в конструкторе, ведь это и без того делает конструктор по умолчанию. – *Прим. перев.*

ОТ ИТЕРАТОРОВ К НАБЛЮДАЕМЫМ ИСТОЧНИКАМ

Шаблон «Итератор» – это стандартный механизм для выборки данных из контейнеров библиотеки STL, а также разного рода потоков и объектов-генераторов. Итераторы хорошо подходят для данных, определённым образом организованных в пространстве памяти. В сущности, это означает, что потребителю заранее известно, сколько элементов данных ему нужно или что данные уже имеются в наличии. Однако нередко бывает так, что элементы данных приходят асинхронно, в непредсказуемые моменты времени, а потребитель не может знать заранее, сколько таких элементов ещё придёт в будущем. В таких случаях программисту приходится либо смириться с тем, что итератор может ждать неопределённо долго, либо идти на уловки с тайм-аутом. Тогда семантика вталкивания представляется лучшим решением. Благодаря поддержке тем (subjects) в библиотеке RxCpp можно воспользоваться преимуществами подхода «выстрелил и забыл». Создадим класс, который испускает информацию о содержимом директории в виде последовательности объектов.

```
// Упрощённая реализация шаблона "Активный объект"
template <class T>
struct ActiveObject {
    rxcpp::subjects::subject<T> subj;

    // выстрелил и забыл
    void FireNForget(T & item) {
        subj.get_subscriber().on_next(item);
    }

    rxcpp::observable<T> GetObservable() {
        return subj.get_observable();
    }

    ActiveObject(){}
    ~ActiveObject() {}
};

// Этот класс использует механизм "выстрелил и забыл"
// для проталкивания данных (событий) к приёмнику
class DirectoryEmitter {
    string rootdir;
    // самодостаточный активный объект
    ActiveObject<FileInformation> act;
public:
    DirectoryEmitter(string s) {
        rootdir = s;
        // подписка
        act.GetObservable().subscribe(
            [] (FileInformation item) {
                cout << item.name << ":" << item.size << endl;
            });
    };
};
```



```

    }

    bool Trigger() {
        std::packaged_task<int>> task(
            [&] () { EmitDirEntry(); return 1; });
        std::future<int> result = task.get_future();
        task();
        // если раскомментировать следующую строку, метод
        // будет работать синхронно, дожидаясь результата,
        // иначе метод завершается немедленно, а вталкивание
        // данных выполняется асинхронно
        // double dresult = result.get();
        return true;
    }

    // пройти по списку файлов, к каждому
    // применить активный объект
    bool EmitDirEntry() {
        list<FileInformation> rs = GetAllFiles(rootdir);
        for (auto& a : rs) { act.FireNForget(a); }
        return false;
    }
};

int main(int argc, char *argv[]) {
    DirectoryEmitter emitter("D:\\Java");
    emitter.Trigger();
    return 0;
}

```

В классе `DirectoryEmitter` использована конструкция `packaged_task` из арсенала современного языка C++, которая позволяет выполнять асинхронные вызовы по принципу «выстрелил и забыл». В данном примере присутствует закомментированная строка, которая, если её раскомментировать, заставляет метод работать синхронно.

ШАБЛОН «ЯЧЕЙКА»

Читатель уже хорошо знает, что реактивное программирование в целом построено вокруг обработки последовательности значений, сменяющих друг друга со временем. Возможны две трактовки таких временных последовательностей.

- **Ячейка.** Это единичная сущность (переменная, область в памяти), значение в которой время от времени изменяется. Такие хранилища изменяемых значений называют ещё свойствами, или поведением.
- **Поток.** Это последовательность элементов данных, каждый из которых остаётся неизменным. Со временем последовательность пополняется новыми элементами. В контексте наблюдаемых источников обычно пользуются именно этой трактовкой данных.

Ниже будет показан упрощённый пример программирования в терминах обновляемой ячейки. При этом будет воплощена лишь базовая функциональность. Представленный ниже код нуждается в существенной доработке для использования в реальных проектах. В частности, эту реализацию можно оптимизировать, если добавить к ней класс `CellController` (контроллер ячеек), который бы централизованно принимал оповещения об обновлениях от всех ячеек. Затем этот контроллер мог бы инициировать каскад обновлений других ячеек в соответствии с графом зависимостей. Представленная здесь упрощённая реализация лишь демонстрирует, что метафора изменяемых ячеек представляет собой достаточно хороший инструмент для организации вычислений по цепочке зависимостей.

```
#include <rxcpp/rx.hpp>
#include <memory>
#include <map>
#include <algorithm>
using namespace std;

class Cell {
private:
    std::string name;
    std::map<std::string, Cell*> parents;
    rxcpp::subjects::behavior<double> *behsubject;

public:
    string get_name() { return name; }
    void SetValue(double v ) {
        behsubject->get_subscriber().on_next(v);
    }

    double GetValue() {
        return behsubject->get_value();
    }

    rxcpp::observable<double> GetObservable() {
        return behsubject->get_observable();
    }

    Cell(std::string pname) {
        name = pname;
        behsubject = new rxcpp::subjects::behavior<double>(0);
    }

    ~Cell() {
        delete behsubject;
        parents.clear();
    }

    bool GetCellNames(string& a, string& b) {
        if (parents.size() != 2) { return false; }
```



```

    int i = 0;
    for(auto p : parents ) {
        (i == 0) ? a = p.first : b = p.first;
        i++;
    }
    return true;
}

// в этом примере у ячейки должно быть ровно два родителя
bool Recalculate() {
    string as , bs;
    if (!GetCellNames(as,bs)) { return false; }
    auto a = parents[as];
    auto b = parents[bs];
    SetValue(a->GetValue() + b->GetValue());
    return true;
}

bool Attach(Cell& s) {
    if (parents.size() >= 2) { return false; }
    parents.insert(pair<std::string,Cell*>(s.get_name(),&s));
    s.GetObservable().subscribe(
        [=] (double a ) { Recalculate(); });
    return true;
}

bool Detach(Cell& s) { /* не реализовано */ }
};

```

В данном предельно упрощённом примере предполагается, что у каждой ячейки есть ровно две родительские ячейки, т. е. две ячейки, от которых она непосредственно зависит. Если значение хотя бы одной из родительских ячеек изменилось, значение текущей ячейки должно быть вычислено заново. В этом примере, ради краткости кода, реализована единственная зависимость от родительских ячеек: операции сложения. За вычисление нового значения зависимой ячейки отвечает метод `recalculate`.

```

int main(int argc, char *argv[]) {
    Cell a("a");
    Cell b("b");
    Cell c("c");
    Cell d("d");
    Cell e("e");
    // присоединить ячейки a и b к ячейке c,
    // теперь c == a + b
    c.Attach(a);
    c.Attach(b);
    // присоединить ячейки c и d к ячейке e,
    // теперь e == c + d == a + b + d
    e.Attach(c);
    e.Attach(d);
}

```

```

a.SetValue(100); // должно напечатать 100
cout << "Значение " << c.GetValue() << endl;
b.SetValue(200); // должно напечатать 300
cout << "Значение " << c.GetValue() << endl;
b.SetValue(300); // должно напечатать 400
cout << "Значение " << c.GetValue() << endl;
d.SetValue(-400); // должно напечатать 0
cout << "Значение " << e.GetValue() << endl;
}

```

Эта главная функция демонстрирует распространение изменений по зависимым ячейкам. Произведённое вручную изменение значения вызывает каскад изменений в зависимых ячейках.



ШАБЛОН «АКТИВНЫЙ ОБЪЕКТ»

Активный объект – это объект, который позволяет разорвать жёсткую связь между вызовом метода и его выполнением. Активные объекты представляют собой инструмент для реализации асинхронных вызовов по принципу «выстрелил и забыл». К активному объекту может быть подключен планировщик, ответственный за обработку запросов на выполнение. Данный шаблон складывается из следующих шести частей:

- объект-посредник, предоставляющий клиентам интерфейс с общедоступными методами;
- интерфейс, посредством которого активному объекту можно послать запрос на выполнение метода;
- список ожидающих выполнения запросов от клиентов;
- планировщик, который решает, в каком порядке обслуживать запросы;
- реализация методов активного объекта;
- функция обратного вызова, общая переменная или иной механизм, позволяющий клиенту получить запрошенный результат.

Рассмотрим в подробностях реализацию активного объекта. Следующий пример кода создан для демонстрационных целей. Код, предназначенный для использования в реальном проекте, потребовал бы некоторых дополнительных усовершенствований, однако соблюдение всех промышленных критериев качества сделало бы код слишком объёмным для нашего рассмотрения.

```

#include <gxcpp/gx.hpp>
#include <memory>
#include <map>
#include <algorithm>
#include <string>
#include <vector>
#include <windows.h>
#include <functional>
#include <thread>
#include <future>
using namespace std;

```



```
// упрощённая реализация активного объекта
template <class T>
class ActiveObject {
    // диспетчер
    rxcpp::subjects::subject<T> subj;

protected:
    ActiveObject() {
        subj.get_observable().subscribe( [= ] (T s) {
            Execute(s);
        });
    }

    virtual void Execute(T s) {}

public:
    // выстрелил и забыл
    void FireNForget(T item) {
        subj.get_subscriber().on_next(item);
    }

    rxcpp::observable<T> GetObservable() {
        return subj.get_observable();
    }

    virtual ~ActiveObject() {}
};
```



В приведённом выше коде экземпляр класса `subject<T>` используется в качестве механизма оповещения исполнителя о новых запросах на асинхронный вызов методов. Метод `FireNForget` отправляет значение аргумента в объекту `subj` через посредство её метода `get_subscriber`. Этот метод немедленно возвращает выполнение вызвавшему контексту, а тем временем скрытые от клиента асинхронные механизмы обеспечат (возможно, в какой-то момент времени в будущем) вызов метода `Execute` с этим значением аргумента. Предполагается, что данный класс будет использоваться в качестве базового для пользовательских классов, содержащих конкретную реализацию метода `Execute`. Рассмотрим пример такого класса.

```
class ConcreteObject : public ActiveObject<double> {
public:
    ConcreteObject() {}
    virtual void Execute(double a) {
        cout << "Hello World....." << a << endl;
    }
};

int main(int argc, char *argv[]) {
    ConcreteObject temp;
    for (int i=0; i<=10; ++i)
        temp.FireNForget(i*i);
    return 0;
}
```

В этом фрагменте кода объявляется пользовательский класс, обладающий функциональностью активного объекта и реализующий конкретный алгоритм обработки `Execute` с аргументом вещественного типа. Главная функция многократно посылает активному объекту запросы на асинхронное вычисление, в результате чего вызывается переопределённый метод `Execute`.

ШАБЛОН «РЕСУРС ВЗАЙМЫ»

Как и явствует из названия, этот шаблон проектирования предполагает выдачу ресурса во временное пользование вызываемой функции. При этом выполняется следующая последовательность действий:

- создаётся необходимый для работы функции ресурс;
- ресурс передаётся функции во временное пользование;
- функция выполняет свою работу, используя выданный ресурс, и возвращает управление вызывавшему контексту;
- ресурс уничтожается.

Выдача ресурса займы с гарантированным последующим освобождением помогает избежать утечки ресурсов. В следующем фрагменте кода показан пример реализации этого шаблона.

```
#include <rxcpp/rx.hpp>
using namespace std;
// Пример реализации ресурса, выдаваемого займы.
// Объект открывает файл и не допускает просачивания
// дескриптора файла в клиентский код. Файл остаётся
// в исключительном владении данного объекта
class ResourceLoan {
    FILE *file;
    string filename;
public:
    ResourceLoan(string pfile) {
        filename = pfile;
        file = fopen(filename.c_str(), "rb");
    }

    // прочитать до 1024 байт в буфер. Буфер и фактический
    // размер прочитанных данных отдать пользовательской
    // функции на обработку
    int ReadBuffer(function<int(char pbuffer[], int val)> func) {
        if (file == nullptr) { return -1; }
        char buffer[1024];
        int result = fread (buffer, 1, 1024, file);
        return func(buffer, result);
    }

    // деструктор закрывает файл
    ~ResourceLoan() { fclose(file); }
};
```

```
// Демонстрация работы объявленного выше класса
int main(int argc, char *argv[]) {
    ResourceLoan res("a.bin");
    int nread;

    // напечатать и вернуть размер прочитанных данных
    auto rlambda = [] (char buffer[] , int val) {
        cout << "Size " << val << endl;
        return val;
    };

    // дескриптор файла скрыт от пользовательской функции
    while ((nread = res.ReadBuffer(rlambda)) > 0) {}

    // при выходе за область видимости объекта ResourceLoan
    // файл автоматически закрывается
    return 0;
}
```

Шаблон «Ресурс займа» действительно позволяет избежать утечки ресурсов. Объект-обёртка остаётся единственным владельцем ресурса и никогда не отдаёт его клиенту. Однако клиент может делать с этим ресурсом всё, что угодно, через предоставляемый обёрткой безопасный интерфейс. Освобождение ресурса происходит автоматически при уничтожении обёртки. Приведённая выше главная функция демонстрирует использование такого объекта.

ШАБЛОН «ШИНА СОБЫТИЙ»

Шина событий работает как посредник между источниками и приёмниками событий. Источник, или производитель, испускает события и направляет их на общую шину, а объекты, подписанные на события, автоматически получают оповещения. Этот шаблон может считаться частным случаем шаблона «Посредник»¹. Реализация шины событий состоит из следующих основных элементов:

- производители – объекты, испускающие события;
- потребители – объекты, в конечном счёте получающие события для обработки;
- контроллеры – объекты, выступающие одновременно производителями и потребителями событий.

Ниже представлен пример реализации шины событий. Контроллеры в данном примере не реализованы для краткости.

```
#include <rxcpp/rx.hpp>
#include <memory>
#include <map>
#include <algorithm>
```

¹ Кроме того, очевидна его связь с шаблоном «Наблюдатель». – Прим. перев.

```
using namespace std;
```

```
// Объект-событие
```

```
struct EVENT_INFO{
```

```
    int id;
```

```
    int err_code;
```

```
    string description;
```

```
    EVENT_INFO() {
```

```
        id = err_code = 0;
```

```
        description = "default";
```

```
    }
```

```
    EVENT_INFO(int pid, int perr_code, string pdescription) {
```

```
        id = pid;
```

```
        err_code = perr_code;
```

```
        description = pdescription;
```

```
    }
```

```
void Print() {
```

```
    cout
```

```
        << "id & Error Code"
```

```
        << id
```

```
        << ":"
```

```
        << err_code
```

```
        << ":"
```

```
        << description
```

```
        << endl;
```

```
    }
```

```
};
```



Структура `EVENT_INFO` моделирует событие и содержит следующие поля:

- идентификатор события;
- код ошибки;
- описание события.

Следующий отрывок кода вполне очевиден:

```
// эту функцию будут вызывать потребители
```

```
template <class T>
```

```
void DoSomethingWithEvent(T ev)
```

```
{ ev.Print(); }
```

```
// упреждающее объявление
```

```
template <class T>
```

```
class EventBus;
```

```
// производитель отправляет события на шину
```

```
template <class T>
```

```
class Producer {
```

```
    string name;
```

```
public:
```

```
    Producer(string pname ) { name = pname; }
```




```

    bool Fire(T ev, EventBus<T> *bev) {
        bev->FireEvent(ev);
        return false;
    }
};

```

Реализация производителя событий весьма проста. Метод `Fire` принимает шину – объект типа `EventBus<T>`, параметризованный подходящим типом `T`, – и вызывает у шины метод `FireEvent`. Реализация потребителя выглядит несколько сложнее. Ниже представлен её код.

```

// потребитель подписывается на события
template <class T>
class Consumer {
    string name;
    // объект subscription позволяет отписаться
    rxcpp::composite_subscription subscription;

public:
    Consumer(string pname) { name = pname;}

    // подключить потребителя к шине
    bool Connect( EventBus<T> *bus ) {
        // предотвратить двойную подписку
        if (subscription.is_subscribed() )
            subscription.unsubscribe();
        // создать новую подписку
        subscription = rxcpp::composite_subscription();
        auto subscriber = rxcpp::make_subscriber<T>(
            subscription,
            [=] (T value) { DoSomethingWithEvent<T>(value); },
            [] () { printf("OnCompleted\n");});
        // собственно, подписаться
        bus->GetObservable().subscribe(subscriber);
        return true;
    }

    // в деструкторе - отписаться
    ~Consumer() { Disconnect(); }

    bool Disconnect() {
        if (subscription.is_subscribed())
            subscription.unsubscribe();
    }
};

```

Принцип действия потребителя событий довольно прост. Метод `Connect` отвечает за подписку потребителя на события, исходящие от предоставляемого шиной объекта-темы – от той его стороны, которая выступает наблюдаемым источником. Если при этом потребитель уже был подписан на события, старая подписка отменяется.

```

// реализация шины событий
template <class T>
class EventBus {
private:
    std::string name;
    // объект-тема, на который подписываются потребители
    rxcpp::subjects::behavior<T> *replaysubject;

public:
    EventBus<T>() {
        replaysubject = new rxcpp::subjects::behavior<T>(T());
    }

    ~EventBus() {delete replaysubject;}

    // подключить потребителя к шине
    bool AddConsumer( Consumer<T>& b ) {b.Connect(this);}

    // испустить событие
    bool FireEvent (T& event) {
        replaysubject->get_subscriber().on_next(event);
        return true;
    }

    string get_name() { return name;}

    rxcpp::observable<T> GetObservable() {
        return replaysubject->get_observable();
    }
};

```

Шина данных работает как трубопровод для транспортировки событий от производителей к потребителям. Внутренние механизмы шины опираются на объект-тему, который выступает наблюдаемым источником данных, извещаая потребителей. В приведённой здесь упрощённой реализации оставлены без внимания аспекты функционирования шины в многопоточной среде. Остаётся показать только главную функцию, которая приводит в действие всю систему.

```

int main(int argc, char *argv[]) {
    // создать шину событий
    EventBus<EVENT_INFO> program_bus;
    // создать производителя и двух потребителей
    // подключить потребителей к шине
    Producer<EVENT_INFO> producer_one("первый производитель");
    Consumer<EVENT_INFO> consumer_one("потребитель А");
    Consumer<EVENT_INFO> consumer_two("потребитель Б");
    program_bus.AddConsumer(consumer_one);
    program_bus.AddConsumer(consumer_two);

    // испустить событие
    EVENT_INFO ev;
    ev.id = 100;
}

```

```

ev.err_code = 0;
ev.description = "Здравствуй, мир";
producer_one.Fire(ev,&program_bus);

// испустить ещё одно событие, создав нового производителя
ev.id = 100;
ev.err_code = 10;
ev.description = "Произошла ошибка";
Producer<EVENT_INFO> producer_two("второй производитель");
producer_two.Fire(ev,&program_bus);
return 0;
}

```



Главная функция выполняет следующие действия.

1. Создание шины событий.
2. Создание производителя событий.
3. Создание потребителей событий.
4. Генерация событий, которые автоматически отправляются на шину и приходят потребителям.

В этой главе мы разобрали лишь небольшую часть шаблонов проектирования, удобных для создания реактивных программ. В центре нашего внимания находились взаимосвязи между шаблонами «Банды четырёх» и принципами реактивного программирования. Авторы убеждены, что модель реактивного программирования представляет собой не что иное, как усовершенствованную версию классических шаблонов «Банды четырёх». Реализация этих усовершенствований средствами языка программирования стала возможной благодаря появлению в них средств, характерных для функциональной парадигмы. Сочетание объектно-ориентированного и функционального стилей программирования представляется наилучшим подходом к созданию современного кода на языке C++. Эта идея лежит в основе настоящей главы.

Итоги



В этой главе читатель погрузился в чудесный мир шаблонов и идиом программирования. Глава открывалась разбором шаблонов «Банды четырёх», затем были изложены более специализированные шаблоны реактивного программирования. На примерах показаны шаблоны «Ячейка», «Активный объект», «Ресурс взаймы» и «Шина событий». Понимание связей между классическими шаблонами «Банды четырёх» и принципами реактивного программирования поможет читателю шире взглянуть на данный предмет.

Следующая глава посвящена разработке микросервисов на языке C++.

Глава 11

Реактивные микросервисы на языке C++

В предыдущих главах рассмотрен ряд важных аспектов реактивного программирования на языке C++, в том числе:

- 1) реактивная модель программирования и её концептуальные основания;
- 2) библиотека RxCpp и присущая ей модель программирования;
- 3) реактивное программирование пользовательских интерфейсов с использованием библиотек Qt и RxCpp;
- 4) шаблоны проектирования и их связь с реактивной моделью программирования.

Если ещё раз окинуть взором разобранные ранее примеры, можно заметить, что они замкнуты в себе, т. е. обрабатывают лишь те события, которые генерируются этим же вычислительным процессом. При этом уделялось большое внимание средствам параллельного и многопоточного программирования и синхронизации доступа к общей памяти (на эти задачи в значительной степени ориентированы библиотеки Rx.Net и RxJava). Система Akka и ряд подобных ей позволяют перенести модель реактивного программирования в мир распределённых систем. С помощью системы Akka можно программировать реактивную логику взаимодействия между различными процессами. Реактивная модель программирования хороша также для программирования серверов и клиентов, общающихся между собой по интерфейсу REST. Библиотека RxJs широко используется для создания кода клиентской стороны, выполняющегося в браузере и работающего с сервером по интерфейсу REST. Библиотеку RxCpp тоже можно использовать для программирования веб-клиентов, агрегирующих информацию от нескольких серверов. Средства библиотеки RxCpp можно с пользой применять и в консольных приложениях, и в приложениях с графическим пользовательским интерфейсом. Ещё одна возможная область применения этой библиотеки состоит в агрегировании данных от множества мелких сервисов для поставки их веб-клиентам.

В этой главе будет рассматриваться задача создания веб-приложения на языке C++ с использованием специальной библиотеки для поддержки интер-

фейса REST как на серверной, так и на клиентской стороне. В процессе решения этой задачи будет рассмотрено, что такое микросервисы и как их использовать в своих разработках. Также речь будет идти о том, как применить библиотеку RxCpp вместе с обёрткой над библиотекой libcurl для коммуникации с удалёнными приложениями по протоколу REST и для обработки веб-страниц. В качестве примера, демонстрирующего этот подход в действии, будет использована библиотека RxCurl, созданная Кирком Шупом (Kirk Shoop) первоначально для его анализатора данных из системы Twitter.

Язык C++ и ВЕБ-ПРОГРАММИРОВАНИЕ

В наши дни большая часть приложений, предназначенных для работы в среде веб, разрабатывается на языках Python, Java, C#, PHP и других языках особо высокого уровня. Однако даже для обеспечения работы таких приложений обычно бывают нужны обратные прокси-серверы и веб-серверы, такие как Nginx, Apache, IIS, на которые возлагается обработка потока сообщений, создаваемого приложениями. Такие серверы, предназначенные для высокоэффективной обработки интенсивных потоков данных, обычно пишутся на языке C++. Также по причинам, связанным с эффективностью, на языке C++ написаны популярные веб-браузеры и клиентские библиотеки для протокола HTTP, такие как libwww, libcurl или WinInet.

Одна из причин популярности статически типизированных языков Java и C#, как и языков с динамической типизацией Python, Ruby и PHP, состоит в том, что первые поддерживают механизмы рефлексии, а вторые – т. н. «утиную» типизацию. Оба этих механизма позволяют веб-приложениям динамически подгружать обработчики для разных типов запросов и разных типов содержимого сообщений. Читателю рекомендуется самостоятельно отыскать и изучить сведения о рефлексии и «утиной» типизации.

Модель программирования REST

REST (от англ. REpresentational State Transfer, передача состояния представления) – это архитектурный стиль взаимодействия компонентов распределённого приложения, описанный одним из создателей протокола HTTP Роем Филдингом в своей диссертации. На сегодняшний день это одна из наиболее популярных технологий взаимодействия между клиентами и серверами в составе распределённой системы. Архитектура REST сфокусирована на понятии ресурса и хорошо согласуется с шаблоном CRUD, широко применяемым при создании корпоративных информационных систем. Вместе с архитектурой REST для представления передаваемых данных часто используется формат JSON (JavaScript Object Notation) – в отличие от протокола SOAP, для которого характерно использование формата XML. Модель программирования, связанная с архитектурой REST, опирается на «глаголы» – методы запросов, определённые в протоколе HTTP. Метод определяет, какого рода действие должно

быть выполнено над ресурсом, идентифицируемым адресной строкой. Чаще всего используются следующие методы запросов:

- POST – создание нового ресурса;
- GET – чтение данных из ресурса;
- PUT – внесение изменений в существующий ресурс (также может работать и подобно методу POST, создавая новый ресурс);
- DELETE – удаление ресурса.

Библиотека REST SDK для языка C++

Пакет C++ REST SDK корпорации Microsoft – это набор средств программирования на языке C++, предназначенный для организации клиент-серверных взаимодействий, обеспечивающий высокую производительность, присущую компиляции в машинный код, и использующий мощь современных средств асинхронного программирования, поддерживаемых языком. Цель этой библиотеки состоит в том, чтобы помочь разработчикам на языке C++ в подключении к веб-сервисам и взаимодействии с ними. Для этого пакет поддерживает следующую функциональность:

- клиент и сервер протокола HTTP;
- формат представления данных JSON;
- асинхронные потоки данных;
- клиент протокола WebSocket;
- протокол авторизации OAuth.

Библиотека C++ REST SDK опирается на богатые средства управления асинхронными задачами, предоставляемые библиотекой параллельных шаблонов PPL. Асинхронные задачи из библиотеки PPL предоставляют программисту мощную модель асинхронного программирования на основе новых средств языка C++. Библиотека C++ REST SDK поддерживает платформы Windows desktop, Windows Store (UWP), Linux, macOS, Unix, iOS и Android.

Программирование HTTP-клиента с использованием библиотеки C++ REST SDK

Модель программирования, лежащая в основе библиотеки C++ REST SDK, асинхронна по своей сути, но также предоставляет возможность для синхронных вызовов. Следующая программа демонстрирует асинхронные вызовы функций из программного интерфейса HTTP-клиента. Программа получает данные веб-страницы с сервера и сохраняет их в файл. В ней используются так называемые **продолжения** задач (continuations) – средство, позволяющее собирать асинхронные действия в цепочки, указывая, что по окончании одной задачи должна быть выполнена некоторая другая. При этом с точки зрения пользовательского кода вся такая цепочка выглядит как одна асинхронная задача.

```
#include <cpprest/http_client.h>
#include <cpprest/filestream.h>
```

```
#include <string>
#include <vector>
#include <algorithm>
#include <sstream>
#include <iostream>
#include <fstream>
#include <random>
#include "cpprest/json.h"
#include "cpprest/http_listener.h"
#include "cpprest/uri.h"
#include "cpprest/asyncrt_utils.h"
```



```
// Пример использования REST SDK:
// клиентское приложение для загрузки веб-страницы
```

```
using namespace utility; // разные полезные функции
using namespace web; // URI и общая функциональность
using namespace web::http; // протокол HTTP, общие средства
using namespace web::http::client; // клиент протокола HTTP
using namespace concurrency::streams; // асинхронные потоки
```

```
int main(int argc, char* argv[]) {
    auto fileStream = std::make_shared<ostream>();
    // открыть файл для записи
    pplx::task<void> requestTask =
        fstream::open_ostream(U("google_home.html"))
        .then([=] (ostream outFile) {
            *fileStream = outFile;
            // создать HTTP-клиент
            http_client client(U("http://www.google.com"));
            // построить URI
            uri_builder builder(U("/"));
            // послать запрос
            return client.request(
                methods::GET, builder.to_string());
        })
        .then([=] (http_response response) {
            printf(
                "Received response status code:%un",
                response.status_code());
            return response.body()
                .read_to_end(fileStream->streambuf());
        })
        .then([=] (size_t) { return fileStream->close(); });

    // Никакие действия пока не выполняются:
    // лишь построена цепочка асинхронных задач.
    // Теперь выполнить её и обработать ошибки.
    try {
        requestTask.wait();
    }
    catch (const std::exception &e) {
```

```

    printf("Error exception:%sn", e.what());
}

getchar(); // приостановить выполнение
return 0;
}

```

Эта программа иллюстрирует стиль программирования, основанный на компоновке асинхронных задач с помощью механизма продолжений. Большая часть этого кода отвечает за составление цепочки действий, тогда как собственно выполнение программой своей основной задачи сводится к одному вызову метода `wait` для этой цепочки. Впрочем, библиотека позволяет программировать и в более привычном синхронном стиле. Читателю рекомендуется обратиться к документации за более подробной информацией.

Программирование HTTP-сервера

В предыдущем разделе было показано, как создать HTTP-клиента на языке C++ с помощью библиотеки REST SDK. Для этого был использован программный интерфейс, основанный на асинхронных задачах и продолжениях. Программа отправляла на сервер запрос, получала веб-страницу и сохраняла её в файл. Теперь пора заняться разработкой HTTP-сервера на основе той же библиотеки REST SDK. Средства библиотеки позволяют настраивать прослушивание определённого порта в ожидании веб-запроса и назначать обработчики для каждого типа веб-запроса: например, для запросов типа GET, PUT и POST.

```

// (директивы включения заголовочных файлов опущены)
// Простой веб-сервер на основе библиотеки REST SDK
using namespace std;
using namespace web;
using namespace utility;
using namespace http;
using namespace web::http::experimental::listener;

// класс-обёртка над классом http_listener
// из библиотеки REST SDK
class SimpleServer {
public:
    SimpleServer/utility::string_t url);
    ~SimpleServer() {}
    pplx::task<void> Open() { return m_listener.open(); }
    pplx::task<void> Close() { return m_listener.close(); }

private:
    // обработчики для разных типов HTTP-запросов
    void HandleGet(http_request message);
    void HandlePut(http_request message);
    void HandlePost(http_request message);
    void HandleDelete(http_request message);
}

```



```
// точка входа запросов
http_listener m_listener;
};
```

Класс SimpleServer – по сути своей обёртка над библиотечным классом http_listener. Этот класс умеет слушать входящие запросы по протоколу HTTP и позволяет устанавливать обработчики для каждого типа запроса (GET, PUT, POST и т. д.). Всякий раз, когда по сети приходит новый запрос, объект класса http_listener вызывает соответствующий обработчик и передаёт запрос ему.

```
SimpleServer::SimpleServer(utility::string_t url)
    : m_listener(url)
{
    using namespace std::placeholders;
    // подписка методов-обработчиков на запросы нужных типов
    m_listener.support(
        methods::GET,
        std::bind(&SimpleServer::HandleGet, this, _1));
    m_listener.support(
        methods::PUT,
        std::bind(&SimpleServer::HandlePut, this, _1));
    m_listener.support(
        methods::POST,
        std::bind(&SimpleServer::HandlePost, this, _1));
    m_listener.support(
        methods::DEL,
        std::bind(&SimpleServer::HandleDelete, this, _1));
}
```

Показанный выше фрагмент кода настраивает объект класса http_listener, устанавливая функции-обработчики для различных типов запросов. В этом примере поддерживаются запросы четырёх типов: GET, PUT, POST и DELETE – это наиболее распространённые команды, поддерживаемые большинством систем, основанных на архитектуре REST.

```
// В этом упрощённом примере вся обработка запроса
// заключается в том, чтобы напечатать информацию о
// запросе на консоль и вернуть код "200 OK" вместе
// с сообщением об успешном завершении
void SimpleServer::HandleGet(http_request message) {
    ucout << message.to_string() << endl;
    message.reply(status_codes::OK, L"GET: успешно выполнено");
};

void SimpleServer::HandlePost(http_request message) {
    ucout << message.to_string() << endl;
    message.reply(status_codes::OK, L"POST: успешно выполнено");
};

void SimpleServer::HandleDelete(http_request message) {
    ucout << message.to_string() << endl;
    message.reply(status_codes::OK, L"DELETE: успешно выполнено");
}
```

```
void SimpleServer::HandlePut(http_request message) {
    ucout << message.to_string() << endl;
    message.reply(status_codes::OK, L"PUT: успешно выполнено");
};
```



Представленный выше код построен по единому образцу, понятному для любого разработчика. Всё, что делает каждый из четырёх обработчиков, – это вывод на консоль параметров запроса и отсылка клиенту ответа об успешном выполнении операции. В следующем разделе будет показано, как обратиться к этим четырём функциям нашего сервера с помощью утилит postman и curl.

```
// умный указатель на единственный экземпляр объекта-сервера
std::unique_ptr<SimpleServer> g_http;

// запустить сервер, назначив ему адрес
void StartServer(const string_t& address) {
    // построить URI для прослушивания
    uri_builder uri(address);
    uri.append_path(U("dbdemo/"));
    auto addr = uri.to_uri().to_string();

    // создать экземпляр сервера и запустить обработку запросов
    g_http = std::make_unique<SimpleServer>(addr);
    g_http->Open().wait();

    // сообщить пользователю о старте сервера
    ucout
        << utility::string_t(U("Готов к приёму запросов: "))
        << addr
        << std::endl;

    return;
}

// закрыть соединение, дождаться завершения асинхронной операции
void ShutDown() {
    g_http->Close().wait();
    return;
}

// сконфигурировать и запустить систему в целом
int wmain(int argc, wchar_t *argv[]) {
    utility::string_t port = U("34567");
    if (argc == 2) {
        port = argv[1];
    }

    // построить базовый адрес сервера
    utility::string_t address = U("http://localhost:");
    address.append(port);

    StartServer(address);
}
```



```

std::cout << "Нажмите ввод для завершения." << std::endl;
std::string line;
std::getline(std::cin, line);
ShutDown();
return 0;
}

```

Главная функция программы создаёт экземпляр класса-обёртки, используя для этого вспомогательную функцию `StartServer`, запуская тем самым функционирование сервера. Затем главная функция ожидает нажатия клавиши «Ввод» и останавливает сервер. Запустив это приложение, можно воспользоваться утилитами `postman` и `curl`, чтобы проверить его в действии.

ТЕСТИРОВАНИЕ HTTP-СЕРВЕРА С ПОМОЩЬЮ УТИЛИТ CURL И POSTMAN

Инструмент `curl` – это кроссплатформенная утилита с интерфейсом командной строки, доступная в системах Windows, GNU Linux, macOS и в других системах стандарта POSIX. Эта утилита позволяет передавать данные по сети посредством различных прикладных протоколов, основанных на стеке TCP/IP. Поддерживаются такие широко распространённые протоколы, как HTTP, HTTPS, FTP, FTPS, SCP, TFTP, DICT, TELNET и LDAP.

Воспользуемся утилитой `curl`, чтобы продемонстрировать работу HTTP-сервера, реализованного в предыдущем разделе. При вызове утилиты ей необходимо через параметры командной строки передать тип запроса, адрес ресурса (URI) и, возможно, дополнительные данные. В частности, команды для отсылки на сервер запросов типа GET и PUT выглядят следующим образом:

```

curl -X GET -H "Content-Type: application/json" http://localhost:34567/dbdemo/
curl -X PUT http://localhost:34567/dbdemo/ -H "Content-Type: application/json" -d
'{"SimpleContent": "Value"}'

```

Эти две команды можно поместить в файл сценария для оболочки командной строки или подать непосредственно с консоли. Результатом их выполнения должен стать текст:

```

GET: успешно выполнено
PUT: успешно выполнено

```

Изучив документацию к утилите `curl`, читатель сможет самостоятельно проверить работу остальных методов протокола HTTP.

`Postman` – это мощное инструментальное средство для тестирования сервисов, работающих по протоколу HTTP. Сначала это был побочный проект индийского программиста по имени Абхинав Астана. Данный инструмент был разработан как подключаемый модуль для браузера Chrome, вскоре его популярность превзошла самые смелые ожидания автора. Ныне инструмент превратился в самостоятельный программный продукт, вокруг которого сфор-

мировалась компания, руководимая Астаной. Читатель может бесплатно попробовать продукт postman и протестировать с его помощью свой сервер.

Создание HTTP-клиента с помощью библиотеки libcurl

Выше был приведён пример использования утилиты curl. Следует сказать, что она представляет собой обёртку над библиотекой libcurl. В этом разделе будет показано, как создать собственное приложение-клиент для протокола HTTP на основе этой библиотеки. Это приложение позволит обращаться к REST-интерфейсу сервера.

```
// пример использования библиотеки libcurl
#include <stdio.h>
#include <curl/curl.h>

int main() {
    CURL *curl;
    CURLcode res;

    // инициализация библиотеки
    curl = curl_easy_init();
    if(curl) {
        // установить адрес
        curl_easy_setopt(
            curl,
            CURLOPT_URL,
            "http://example.com");

        // включить поддержку перенаправлений
        curl_easy_setopt(
            curl,
            CURLOPT_FOLLOWLOCATION,
            1L);

        // всё настроено, можно принимать данные
        res = curl_easy_perform(curl);
        if (res != CURLE_OK) {
            // ошибка
            cout
                << "Произошла ошибка: "
                << curl_easy_strerror(res)
                << endl;
        }

        curl_easy_cleanup(curl);
    }

    return 0;
}
```



Эта программа запрашивает веб-страницу по адресу <http://example.com>, получает её содержимое и выводит его на консоль. Модель программирования,

лежащая в основе библиотеки libcurl, очень проста, библиотека хорошо документирована. На сегодняшний день это одна из наиболее популярных библиотек, предоставляющих приложениям доступ к протоколам TCP/IP.

Реактивная библиотека-обёртка RxCurl

Главную роль в разработке библиотеки RxCpp сыграл Кирк Шуп, ныне работающий в корпорации Microsoft. Однажды он решил создать приложение, анализирующее содержимое сети Twitter (<https://github.com/kirkshoop/twitter>), чтобы на этом примере продемонстрировать различные аспекты реактивного программирования. Среди прочих подзадач в рамках этого проекта было и создание обёртки над библиотекой libcurl, которая бы позволяла проводить обработку веб-запросов типа GET и POST в реактивном стиле. Авторы данной книги расширили первоначальную реализацию, добавив поддержку методов PUT и DELETE. Рассмотрим код библиотеки RxCurl, доступный вместе с исходными кодами примеров к книге:

```
// Простой клиент, получающий данные по протоколу
// HTTP с использованием библиотеки RxCurl
```

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <map>
#include <chrono>
using namespace std;
using namespace std::chrono;
```

```
// библиотеки curl и RxCpp
#include <curl/curl.h>
#include <rxcpp/rx.hpp>
using namespace rxcpp;
using namespace rxcpp::rxo;
using namespace rxcpp::rxs;
```

```
// модифицированная библиотека RxCurl
#include "rxcurl.h"
using namespace rxcurl;
```

```
int main() {
    // создать фабрику HTTP-запросов
    string url = "http://example.com";
    auto factory = create_rxcurl();
    auto request = factory.create(
        http_request(url, "GET", {}, {}) |
        rxo::map([] (http_response r) {
            return r.body.complete;
        })
    );
};
```



Сначала результатом инициализации библиотеки становится объект factory – фабрика HTTP-запросов, затем с помощью этой фабрики создаётся запрос

(объект `request`), а к нему с помощью метода `map` прикрепляется функция-обработчик ответа. Тем самым запрос превращается в наблюдаемый источник. Тип `http_request` играет ключевую роль в подобных веб-приложениях, её определение представлено ниже:

```
struct http_request {
    string url;
    string method;
    std::map<string, string> headers;
    string body;
};
```

Вернёмся к примеру приложения и рассмотрим следующий фрагмент кода:

```
// выполнить блокирующий запрос
observable<string> response_message;
request.as_blocking().subscribe(
    [&] (observable<string> s) {
        response_message = s.sum();
    },
    [] () {});
```

На запрос, преобразованный в наблюдаемый источник данных (объект `request`), можно подписать функцию-обработчик. Обработчик события `on_next` вызывается всякий раз при получении успешного ответа от веб-сервера. В данном случае обработчиком выступает лямбда-функция, которая агрегирует (посредством метода `sum`) данные из всех ответов и строит таким образом строку с полным текстом запрошенного ресурса. Как только формирование текста из отдельных ответов завершится, этот текст становится доступен в наблюдаемом источнике `response_message`.

```
// получить ответ от веб-сервера
string html;
response_message.as_blocking().subscribe(
    [&html] (string temp) { html = temp; },
    [] () {} );

// напечатать результат на консоль
cout << html << endl;
return 0;
}
```



На данные из наблюдаемого источника `response_message` подписывается лямбда-функция, которая полученную из источника строку помещает в переменную `html`. Наконец, остаётся вывести на консоль текст из этой переменной. Читателю рекомендуется самостоятельно изучить, как устроен заголовочный файл `gxcurl.h`.

ИСПОЛЬЗОВАНИЕ ФОРМАТА JSON с протоколом HTTP

Долгое время формат XML держал фактическую монополию на способ представления данных, составляющих содержимое запроса к веб-сервисам. Так,

он почти исключительно используется в сервисах, основанных на протоколе SOAP. Однако с распространением архитектуры REST всё шире для представления данных стал использоваться формат JSON (JavaScript Object Notation). В следующей таблице показаны представления одной и той же структуры данных в этих двух форматах.

XML	JSON
<pre> <person> <firstName>John</firstName> <lastName>Smith</lastName> <age>25</age> <address> <streetAddress>21 2nd Street</streetAddress> <city>New York</city> <state>NY</state> <postalCode>10021</postalCode> </address> <phoneNumber> <type>home</type> <number>212 555-1234</number> </phoneNumber> <phoneNumber> <type>fax</type> <number>646 555-4567</number> </phoneNumber> <gender> <type>male</type> </gender> </person> </pre>	<pre> { "firstName": "John", "lastName": "Smith", "age": 25, "address": { "streetAddress": "21 2nd Street", "city": "New York", "state": "NY", "postalCode": "10021" }, "phoneNumber": [{ "type": "home", "number": "212 555-1234" }, { "type": "fax", "number": "646 555-4567" }], "gender": { "type": "male" } } </pre>

Формат JSON поддерживает следующие типы данных:

- текстовая строка;
- число;
- объект (в свою очередь, заданный в формате JSON);
- массив;
- логическое значение.

Создадим для примера объект, содержащий в себе значения всех этих типов:

- name: строковый тип со значением "John";
- age: числовой тип со значением 35;
- spouse: объект в нотации JSON;
- siblings: массив (элементы которого суть строки);
- employed: логическое значение «истина».

Представление данного объекта в целом выглядит так:

```

{
  "name": "John",
  "age": 35,
  "spouse": {

```

```

        "name": "Joanna",
        "age": 30,
        "city": "New York"
    },
    {
        "siblings": ["Bob", "Bill", "Peter" ]
    },
    { "employed": true }
}

```

Познакомившись с форматом JSON и его основными элементами, напомним простую программу, демонстрирующую работу с этим форматом средствами библиотеки REST SDK.

```

// Консольное приложение для демонстрации средств
// для работы с форматом JSON в библиотеке REST SDK
using namespace std;
using namespace web;
using namespace utility;
using namespace http;
using namespace web::http::experimental::listener;

// структура данных для примера
struct EMPLOYEE_INFO {
    utility::string_t name;
    int age;
    double salary;

    // преобразование текста в формате JSON в объект данных
    static EMPLOYEE_INFO JSonToObject(
        const web::json::object & object)
    {
        EMPLOYEE_INFO result;
        result.name = object.at(U("name")).as_string();
        result.age = object.at(U("age")).as_integer();
        result.salary = object.at(U("salary")).as_double();
        return result;
    }
}

```

Статический метод `JSonToObject` превращает код в формате JSON в объект типа `EMPLOYEE_INFO`. Функция `web::json::object::at` возвращает ссылку на подобъект, находящийся в объекте по заданному ключу. Для извлечения значений простых типов (числовой, строковый, логический) используются методы `as_string`, `as_integer`, `as_double`, `as_bool`.

```

// преобразование объекта данных в текст в формате JSON
web::json::value ObjectToJson() const
{
    web::json::value result = web::json::value::object();
    result[U("name")] = web::json::value::string(name);
    result[U("age")] = web::json::value::number(age);
}

```



```

        result[U("salary")] = web::json::value::number(salary);
        return result;
    }
};

```

Метод `ObjectToJson` преобразовывает экземпляр типа `EMPLOYEE_INFO` в представление в формате JSON. Для преобразования значений простых типов используются статические методы класса `web::json::value`, в данном примере `string`, и `number`.

Следующий фрагмент кода создаёт и наполняет данными JSON-объект.

```

// создание объекта с подобъектами и массивом
void MakeAndShowJSONObject()
{
    // создать JSON-объект
    json::value group;
    group[L"Title"] = json::value::string(U("Native Developers"));
    group[L"Subtitle"] = json::value::string(
        U("C++ devekloers on Windws/GNU LINUX"));
    group[L"Description"] = json::value::string(
        U("A Short Description here "));

    // создать объект-элемент массива
    json::value item;
    item[L"Name"] = json::value::string(U("Praseed Pai"));
    item[L"Skill"] = json::value::string(U("C++ / java "));
    // создать ещё один объект-элемент массива
    json::value item2;
    item2[L"Name"] = json::value::string(U("Peter Abraham"));
    item2[L"Skill"] = json::value::string(U("C++ / C# "));
    // создать JSON-массив
    json::value items;
    items[0] = item;
    items[1] = item2;

    // сделать массив подобъектом имеющегося объекта
    group[L"Resources"] = items;

    // преобразовать JSON-объект в текстовое представление
    utility::stringstream_t stream;
    group.serialize(stream);

    // отобразить строку
    std::wcout << stream.str();
}

```

Теперь рассмотрим главную функцию, демонстрирующую работу функций, описанных выше.

```

int wmain(int argc, wchar_t *argv[])
{
    EMPLOYEE_INFO dm;

```

```
dm.name = L"Sabhir Bhatia";
dm.age = 50;
dm.salary = 10000;
wcout << dm.ObjectToJson().serialize() << endl;
```

Здесь создаётся объект структурного типа `EMPLOYEE_INFO`, и его полям присваиваются определённые значения. Затем он преобразовывается в JSON-объект с помощью метода `ObjectToJson` и далее, посредством метода `serialize`, в текстовое представление.

```
utility::string_t port =
    U("{ \"Name\": \"Alex\", \"Age\": 55, \"salary\": 20000 }");
web::json::value json_par;
json::value obj = json::value::parse(port);
wcout << obj.serialize() << endl;
```

Этот фрагмент демонстрирует, как преобразовать текстовое представление в JSON-объект и обратно. Наконец, посмотрим в действии функцию, которая «на лету» строит объект сложной структуры:

```
MakeAndShowJSONObject();
getchar();
return 0;
}
```

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ C++ REST SDK ДЛЯ СОЗДАНИЯ СЕРВЕРА

В этом разделе используется код из блестящей статьи Мариуса Бансилы, посвящённой библиотеке C++ REST SDK и доступной по адресу <https://mariusbancila.ro/blog/2017/11/19/revisited-full-fledged-client-server-example-with-c-rest-sdk-2-10/>. В частности, оттуда позаимствована реализация ассоциативного хранилища данных. Авторы книги выражают коллеге свою благодарность.

Создадим микросервис, основанный на архитектуре REST, применив при этом всё изученное ранее о библиотеке C++ REST SDK. В этой разработке найдёт своё применение также и `RxCurl` Кирка Шупа, к которой авторы добавили поддержку методов `PUT` и `DELETE`. Наш REST-сервис будет поддерживать следующие методы:

- GET: возвращает список всех хранящихся на сервере пар «ключ-значение». Ответ должен иметь формат {ключ1: значение1, ключ2: значение2, ...};
- POST: возвращает список значений по заданным в запросе ключам. Запрос должен быть в формате [ключ1, ключ2, ...], а ответ – в формате {ключ1: значение1, ключ2: значение2, ...};
- PUT: сохраняет на сервере набор пар «ключ-значение». Метод ожидает запроса в формате {ключ1: значение1, ключ2: значение2, ...};
- DELETE: удаляет заданные в запросе ключи и связанные с ними значения. Запрос должен иметь вид [ключ1, ключ2, ...].

Начнём разбор кода с реализации.

```
#include <cpprest/http_client.h>
#include <cpprest/filestream.h>
#include <string>
#include <vector>
#include <algorithm>
#include <sstream>
#include <iostream>
#include <fstream>
#include <random>
#include <set>

#include "cpprest/json.h"
#include "cpprest/http_listener.h"
#include "cpprest/uri.h"
#include "cpprest/asyncrt_utils.h"

#ifdef _WIN32
#ifndef NOMINMAX
#define NOMINMAX
#endif
#include <Windows.h>
#else
#include <sys/time.h>
#endif

using namespace std;
using namespace web;
using namespace utility;
using namespace http;
using namespace web::http::experimental::listener;

// выводит на консоль текстовое представление JSON-объекта
void DisplayJSON(json::value const & jvalue){
    wcout << jvalue.serialize() << endl;
}

// Функция, проделывающая основную работу.
// Принимает в качестве аргументов запрос
// и функцию-обработчик, ответственную за
// обработку конкретного типа запроса
void RequestWorker(
    http_request& request,
    function<void(json::value const&, json::value&> handler)
){
    auto result = json::value::object();
    request
        .extract_json()
        .then([&result, &handler] (plx::task<json::value> task) {
            try {
                auto const & jvalue = task.get();
```



```

        if (!jvalue.is_null())
            handler(jvalue, result); // вызвать обработчик
    }
    catch (http_exception const & e) {
        wcout << L"Ошибка! " << e.what() << endl;
    }
}
.wait();
request.reply(status_codes::OK, result);
}

```

Функция `RequestWorker`, расположенная в глобальном пространстве имён, принимает два аргумента: запрос и функцию-обработчик, которая, в свою очередь, должна иметь два аргумента:

- тело запроса в виде JSON-объекта;
- ссылку на JSON-объект, в который нужно поместить ответ на запрос.

Функция `RequestWorker` вычленяет текст запроса, интерпретирует его как JSON-объект и передаёт на обработку в задачу-продолжение. Приёмником для результата обработки запроса выступает ссылка на локальную переменную этой функции.

Для выполнения нашим сервером своей основной задачи понадобится создать ассоциативное хранилище, т. е. хранилище пар «ключ-значение», имитирующее функциональность настоящих баз данных.

```

// Игрушечная база данных из одной таблицы с ключом типа строки
class HttpKeyValueDBEngine {
    // данные хранятся здесь
    map<utility::string_t, utility::string_t> storage;
public:
    HttpKeyValueDBEngine() {
        storage[L"Praseed"] = L"45";
        storage[L"Peter"] = L"28";
        storage[L"Andrei"] = L"50";
    }
}

```

Для простоты реализации соответствие между ключами и значениями будем хранить в стандартном ассоциативном контейнере из библиотеки STL. В конструкторе база данных наполняется некоторыми начальными значениями. Далее следуют обработчики для четырёх типов запросов, которые выполняют соответствующие действия над этим контейнером.

```

// Обработчик для метода GET:
// Пройти по всему контейнеру и поместить ключи
// и значения в поток ответа
void GET_HANDLER(http_request& request) {
    auto resp_obj = json::value::object();
    for (auto const & p : storage)
        resp_obj[p.first] = json::value::string(p.second);
    request.reply(status_codes::OK, resp_obj);
}

```

Метод¹ GET_HANDLER должен вызываться объектом-слушателем, когда он принимает HTTP-запрос и обнаруживает, что это запрос типа GET. В этом обработчике создаётся пустой JSON-объект и по очереди наполняется значениями из внутреннего хранилища. Затем этот объект помещается в HTTP-запрос в качестве ответа, который и будет возвращён клиенту.

```
// Обработчик для метода POST
// Получив из запроса список ключей, вернуть их значения
void POST_HANDLER(http_request& request) {
    RequestWorker(
        request,
        [&](json::value const & jvalue, json::value & result)
        {
            // вывести на консоль для диагностики
            DisplayJSON(jvalue);

            for (auto const & e : jvalue.as_array()) {
                if (e.is_string()) {
                    auto key = e.as_string();
                    auto pos = storage.find(key);

                    if (pos == storage.end()) {
                        // этот ключ не найден
                        result[key] = json::value::string(
                            L"not found");
                    }
                    else {
                        // добавить пару ключ-значение в ответ
                        result[pos->first] =
                            json::value::string(pos->second);
                    }
                }
            }
        });
}
```

Метод POST_HANDLER ожидает, что тело запроса будет содержать массив строк. Метод проходит в цикле по этому массиву и, рассматривая каждый элемент как ключ, получает из хранилища соответствующее ему значение. Полученные при этом пары «ключ-значение» накапливаются в JSON-объекте, который будет возвращён клиенту в качестве ответа. Если некоторые из указанных в запросе ключей отсутствуют в хранилище, вместо значения клиенту возвращается сообщение о том, что такой ключ не найден.

¹ Качество предложенного авторами архитектурного решения лежит ниже порога допустимого. Один и тот же класс `HttpKeyValueDBEngine` отвечает как за хранение данных, представляя собой обёртку над библиотечным классом `std::map`, так и за подробности протокола HTTP, тем самым грубо нарушая принцип единственной ответственности (single responsibility) – один из основополагающих принципов красивой архитектуры объектно-ориентированных систем. – *Прим. перев.*

```

// Обработчик для метода PUT:
// Добавить новые ключи или обновить значения имеющихся
void PUT_HANDLER(http_request& request) {
    RequestWorker(
        request,
        [&](json::value const & jvalue, json::value & result)
        {
            DisplayJSON(jvalue);
            for (auto const& e: jvalue.as_object()) {
                if (e.second.is_string()) {
                    auto key = e.first;
                    auto value = e.second.as_string();

                    if (storage.find(key) == storage.end()) {
                        // известить клиента о новом ключе
                        result[key] =
                            json::value::string(L"<put>");
                    }
                    else {
                        // известить, что ключ обновлён
                        result[key] =
                            json::value::string(L"<updated>");
                    }

                    storage[key] = value;
                }
            }
        });
}

```

Обработчик для метода PUT получает из запроса JSON-объект, представляющий собой список пар «ключ-значение». Проходя по этому списку, алгоритм для каждого ключа определяет, есть ли уже такой ключ в хранилище. В ответе клиент для каждого ключа получает сообщение о том, был ли этот ключ добавлен впервые или обновлён.

```

// Обработчик для метода DELETE:
// Удалить из хранилища заданный список ключей
void DEL_HANDLER(http_request& request) {
    RequestWorker(
        request,
        [&](json::value const & jvalue, json::value & result)
        {
            // просканировать список ключей
            // и отбросить несуществующие
            set<utility::string_t> keys;
            for (auto const & e : jvalue.as_array()) {
                if (e.is_string()) {
                    auto key = e.as_string();
                    auto pos = storage.find(key);

```

```

        if (pos == storage.end()) {
            result[key] =
                json::value::string(L"<failed>");
        }
        else {
            result[key] =
                json::value::string(L"<deleted>");
            // добавить в список на удаление
            keys.insert(key);
        }
    }
    // удалить отображенные ключи
    for (auto const & key : keys)
        storage.erase(key);
});
}
};

```

Метод `DEL_HANDLER` ожидает, что содержимое запроса окажется массивом строк-ключей. Сначала метод проходит по элементам этого массива и для каждой строки определяет, существует ли в хранилище такой ключ. Если существует, то ключ добавляется ко множеству тех ключей, которые фактически должны быть удалены, а в ответ добавляется сообщение о том, что этот ключ удалён. В противном случае в ответ добавляется сообщение, что попытка удаления этого ключа вызвала ошибку.

```

// глобальный объект-хранилище данных
HttpKeyValueDBEngine g_dbengine;

```

Таким образом, реализован механизм хранения данных и четыре вида запросов к нему. Теперь нужно сделать эту базу данных доступной для внешнего мира, превратив её в сервер, принимающий запросы по сети в соответствии с архитектурой REST. Обработчики HTTP-запросов будут делегировать свою работу методам разобранного выше класса `HttpKeyValueDBEngine`. Этот код весьма похож на тот, который ранее был написан для класса `SimpleServer`.

```

class RestDbServiceServer
{
public:
    RestDbServiceServer(utility::string_t url);
    pplx::task<void> Open() { return m_listener.open(); }
    pplx::task<void> Close() { return m_listener.close(); }

private:
    void HandleGet(http_request message);
    void HandlePut(http_request message);
    void HandlePost(http_request message);
    void HandleDelete(http_request message);
}

```

```

    http_listener m_listener;
};

RestDbServiceServer::RestDbServiceServer(utility::string_t url)
    : m_listener(url)
{
    using namespace std::placeholders;
    m_listener.support(
        methods::GET,
        std::bind(&RestDbServiceServer::HandleGet, this, _1));
    m_listener.support(
        methods::PUT,
        std::bind(&RestDbServiceServer::HandlePut, this, _1));
    m_listener.support(
        methods::POST,
        std::bind(&RestDbServiceServer::HandlePost, this, _1));
    m_listener.support(
        methods::DEL,
        std::bind(&RestDbServiceServer::HandleDelete, this, _1));
}

```

Показанный выше код привязывает функции-обработчики к соответствующим типам HTTP-запросов. Тела этих обработчиков выглядят практически одинаково, так как они лишь делегируют вызовы соответствующим методам класса-хранилища, который выполняет основную работу.

```

void RestDbServiceServer::HandleGet(http_request message) {
    g_dbengine.GET_HANDLER(message);
};

void RestDbServiceServer::HandlePost(http_request message) {
    g_dbengine.POST_HANDLER(message);
};

void RestDbServiceServer::HandleDelete(http_request message) {
    g_dbengine.DEL_HANDLER(message);
}

void RestDbServiceServer::HandlePut(http_request message) {
    g_dbengine.PUT_HANDLER(message);
};

// глобальный объект - экземпляр сервера
std::unique_ptr<RestDbServiceServer> g_http;

// инициализирует сервер
void StartServer(const string_t& address) {
    uri_builder uri(address);
    uri.append_path(U("dbdemo/"));
    auto addr = uri.to_uri().to_string();
    g_http = std::make_unique<RestDbServiceServer>(addr);
}

```



```

g_http->Open().wait();
ucout
    << utility::string_t(U("Сервер стартовал: "))
    << addr
    << std::endl;
}

void ShutDown() {
    g_http->Close().wait();
    return;
}

// запуск системы в целом
int wmain(int argc, wchar_t *argv[])
{
    utility::string_t port = U("34567");
    if (argc == 2) {
        port = argv[1];
    }

    utility::string_t address = U("http://localhost:");
    address.append(port);

    StartServer(address);
    std::cout << "Press ENTER to exit." << std::endl;

    std::string line;
    std::getline(std::cin, line);

    ShutDown();
    return 0;
}

```

Код HTTP-контроллера не отличается от кода, написанного ранее в этой главе для проекта SimpleServer, и приведён здесь исключительно для полноты примера. В целом данный пример иллюстрирует, как предоставить внешним клиентам интерфейс к приложению по протоколу REST.

Таким образом, в этом разделе было рассмотрено, как программировать обработчики для различных типов запросов протокола HTTP. Архитектурный стиль, основанный на микросервисах, предполагает наличие у множества конечных точек с REST-интерфейсами, каждая из которых работает независимо от других (возможно, на разных машинах). Разбиение крупноблочного серверного приложения на множество независимых микросервисов требует высокого мастерства и сильно зависит от решаемых системой задач. Система микросервисов обычно обладает также интерфейсом к внешнему миру, иногда посредством отдельного агрегирующего сервиса. Для реализации логики доступа к микросервисам, лежащей в основе работы агрегирующего сервиса, удобно применять реактивную модель программирования, особенно если учесть асинхронную природу приходящих по сети запросов.

ОБРАЩЕНИЕ К REST-СЕРВИСАМ С ПОМОЩЬЮ БИБЛИОТЕКИ RxCURL

Библиотека RxCurl, созданная Кирком Шупом, первоначально поддерживала лишь запросы типа GET и POST, так как именно они были нужны в приложении для анализа содержимого сети Twitter. Авторы этой книги модифицировали библиотеку, добавив поддержку запросов типа PUT и DELETE. Следующий фрагмент кода демонстрирует работу с запросом типа PUT. Читатель может обратиться к заголовочному файлу `gxcurl.h`, чтобы узнать, как реализована поддержка новых типов запросов.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <map>
#include <chrono>
using namespace std;
using namespace std::chrono;
// использовать библиотеки Curl и Rxcpp
#include <curl/curl.h>
#include <rxcpp/rx.hpp>
using namespace rxcpp;
using namespace rxcpp::rxo;
using namespace rxcpp::rxs;
// модифицированная библиотека RxCurl
#include "gxcurl.h"
using namespace gxcurl;
gxcurl::rxcurl factory;
```



С помощью объекта `factory` и его метода `create` можно отправлять серверу веб-запросы. Методу `create` нужны следующие параметры:


- идентификатор ресурса (URI);
- метод данного запроса;
- дополнительные заголовки запроса;
- тело запроса.

Для удобства сделана следующая функция-обёртка:

```
string HttpCall(
    string url,
    string method,
    std::map<string,string> headers,
    string body)
{
    auto request = factory.create(
        http_request{ url, method, headers, body } |
        rxo::map([](http_response r){ return r.body.complete; }));
```

Здесь создаётся объект, представляющий запрос, и ему назначается функция-обработчик, которая из полученного ответа вычленяет тело в виде тексто-

вой строки. Протокол HTTP вполне допускает присылку ответа по частям. Наш код обеспечивает сборку этих частей воедино, для чего используется наблюдаемый источник данных:



```
// блокирующий вызов
observable<string> response_message;
request.as_blocking().subscribe(
    [&](observable<string> s){response_message = s.sum();},
    [] () {printf("");});
```

В этом фрагменте кода сделана блокирующая подписка на созданный ранее наблюдаемый источник, поставляющий куски ответа на HTTP-запрос. Его обработчик `on_next` склеивает между собой фрагменты ответа и получившийся текст вталкивает в другой наблюдаемый источник. В реальном приложении эта обработка, скорее всего, велась бы асинхронно, что потребовало бы некоторых дополнительных усилий и заметно удлинило бы код.

```
// получить содержимое ответа
string html;
response_message.as_blocking().subscribe(
    [&html] (string temp) { html = temp; },
    [] () { printf(""); });
return html;
}
```

Остаётся показать главную функцию, которая посылает на сервер ряд запросов и выводит на консоль полученные ответы.

```
// привести систему в действие
int main() {
    // задать url сервера и проинициализировать фабрику запросов
    string url = "http://localhost:34567/dbdemo/";
    factory = create_rxcurl();
    // заголовки запроса
    std::map<string,string> headers;
    headers["Content-Type"] = "application/json";
    headers["Cache-Control"] = "no-cache";

    // вызвать метод GET для получения данных
    string html = HttpCall(url, "GET", headers, "");
    cout << html << endl;

    // получить значение по ключу
    string body = string("[\"Praseed\"]\r\n");
    html = HttpCall(url, "POST", headers, body);
    cout << html << endl;

    // добавить значения методом PUT
    body = string(
        "\r\n{\"Praveen\": \"29\", \"Rajesh\": \"41\"}\r\n");
    html = HttpCall(url, "PUT", headers, body);
    cout << html << endl;
```



```

// убедиться, что значения добавились
html = HttpCall(url, "GET", headers, "");
cout << "Новое состояние базы данных" << endl;
cout << html << endl;

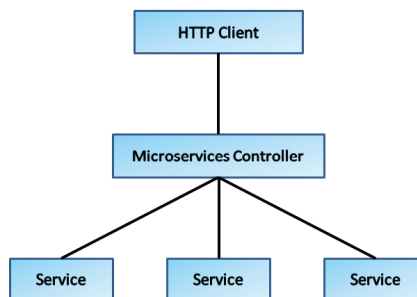
// удалить запись с заданным ключом
body = string("[\"Praseed\"]\r\n");
html = HttpCall(url, "DELETE", headers, body);
cout << "Результат удаления" << html << endl;
html = HttpCall(url, "GET", headers, "");
cout << "Новое состояние базы данных" << endl;
cout << html << endl;
}

```

Как видно из этого кода, реализованная выше функция `HttpCall` удобна для отправки разнообразных запросов и ожидания ответов на них. Этот пример в целом демонстрирует применение библиотеки `RxCpp`. Библиотека позволяет обрабатывать запросы также и в асинхронном стиле.

НЕСКОЛЬКО СЛОВ ОБ АРХИТЕКТУРЕ РЕАКТИВНЫХ МИКРОСЕРВИСОВ

Из предыдущих разделов читатель узнал, как воплотить контроллер микросервиса на языке C++ с помощью библиотеки `REST SDK`. Реализованный нами сервер ассоциативной базы данных вполне можно назвать микросервисом. В реальных системах, построенных на базе микросервисов, каждый из них мог бы быть размещён отдельно от других (возможно, на виртуальной машине или в контейнере), а контроллер микросервисов, обслуживая запросы клиента, обращался бы к этим совершенно независимым друг от друга сервисам. Получив данные от нескольких микросервисов, контроллер должен агрегировать их, чтобы построить ответ для клиента. Типичная архитектура приложения, построенного из микросервисов, показана на следующем рисунке.



На этой диаграмме показано, что HTTP-клиент, реализующий архитектуру `REST`, посылает запрос контроллеру микросервисов. Контроллер, в свою оче-

редь, обращается с различными запросами к трём микросервисам, чтобы из полученных от них данных собрать ответ для клиента. При этом микросервисы могут быть как угодно распределены по физическим и виртуальным машинам.

Как писали Мартан Фаулер и Джеймс Льюис, «термин “Microservice Architecture” получил распространение в последние несколько лет как описание способа дизайна приложений в виде набора независимо развертываемых сервисов. В то время как нет точного описания этого архитектурного стиля, существует некий общий набор характеристик: организация сервисов вокруг бизнес-потребностей, автоматическое развертывание, перенос логики от шины сообщений к приемникам (endpoints) и децентрализованный контроль над языками и данными»¹.

Микросервисы и основанная на них архитектура приложений сами по себе составляют тему, достойную отдельной книги. Задача этой главы состоит лишь в том, чтобы показать, как язык C++ может пригодиться при создании веб-приложений в этом стиле. Изложение темы в этой главе должно подвести читателя к правильной расстановке акцентов. Реактивная модель программирования хорошо подходит для агрегирования информации, полученной асинхронным образом из различных источников, и предоставления клиенту итогового результата её обработки. Именно агрегирование сервисов составляет ключевую задачу, и над этим в первую очередь следует думать читателю.

Говоря о микросервисной архитектуре, нужно хорошо понимать следующие аспекты:

- мелкоблочные сервисы;
- разнородное хранение данных (polyglot persistence);
- независимое развёртывание сервисов;
- оркестровка и хореография сервисов;
- реактивный стиль запросов к веб-сервисам.

Разберём эти темы подробно в следующих разделах.

Мелкоблочные сервисы

В прошлом сервисы, основанные на архитектурах SOA и REST, представляли собой крупные монолиты. В пору зарождения веб-приложений приходилось считаться со временем прохождения запросов и ответов по сети. Чтобы уменьшить количество пересылаемых сообщений, разработчикам часто приходилось изобретать композитные форматы данных, объединяя в одном сообщении данные, различные по своей природе. Поэтому каждая конечная точка распределённой системы, обладающая собственным URI, должна была выполнять множество различных функций, нарушая тем самым принцип разделения обязанностей. Архитектура, построенная на микросервисах, напротив, требует от каждого сервиса выполнения единственной функции, и форматы

¹ Цит. по <https://habr.com/ru/post/249183/>. – Прим. перев.

пересылаемых данных ориентированы именно на неё. Тем самым распределённая система разбивается на многочисленные мелкоблочные сервисы.

Разнородное хранение данных

Под английским словосочетанием «polyglot persistence» понимают такой подход к организации хранения данных, при котором данные в одной системе располагаются в хранилищах разных типов, подходящих под конкретные условия. Это название связано с термином «polyglot programming», означающим подход к программированию, когда каждая часть крупной системы разрабатывается на том языке, который наилучшим образом подходит для задач, возлагаемых на эту часть. Так, например, в составе одной системы могут сочетаться язык Java для серверной части, язык Scala для обработки потоков данных, язык C++ для задач, требующих максимальной эффективности, язык C# для веб-сервера и, конечно же, языки TypeScript и JavaScript для написания кода, выполняемого на стороне клиента. Что же касается хранилищ данных, то в одной системе могут использоваться реляционные, документоориентированные, иерархические СУБД, ассоциативные хранилища или специальные СУБД, ориентированные на хранение временных рядов.

Хорошим примером системы, в которой можно с пользой применить разнородное хранение данных, может служить портал электронной коммерции. Такая система имеет дело с данными нескольких различных категорий, отличающихся как объёмом, так и частотой изменения и массовостью доступа: скажем, списки имеющихся товаров, корзина покупателя, архив выполненных заказов, финансовые отчёты за периоды и по группам товаров. Вместо того чтобы пытаться все эти данные разместить в одном хранилище, стоит для каждой категории данных использовать свою технологию хранения. Таким образом, важной задачей для проектировщика становится выбор наилучшего хранилища данных для каждой подсистемы.

Независимое развёртывание сервисов

Самое большое различие между архитектурой на основе микросервисов и традиционной сервис-ориентированной архитектурой состоит в способе развёртывания компонентов системы. С развитием технологий виртуализации и контейнеризации стало возможным развёртывать любое число сервисов быстро, независимо друг от друга, без какого бы то ни было участия человека. Широкое внедрение практики DevOps заметно способствовало популярности независимо развёртываемых сервисов и приложений. Процесс подготовки к работе новой виртуальной машины под конкретную задачу, включая конфигурирование её процессора, памяти, дисков, сетевых адаптеров, брандмауэра, балансирование нагрузки, масштабирование могут быть полностью автоматизированы и определяться политиками, настроенными на облачном сервисе, таком как AWS или Google Cloud. Имея настроенные политики и сценарии развёртывания, можно на лету создавать готовые к использованию микросервисы.

При разработке распределенных приложений на основе микросервисной архитектуры непременно придется то и дело сталкиваться с технологиями контейнеризации. Подробное изложение вопросов контейнеризации, управления кластерами и практики DevOps выходит далеко за рамки этой книги. Читатель может начать своё знакомство с данной темой с поиска по ключевым словам «docker», «kubernetes», «инфраструктура как код».

Оркестровка и хореография сервисов

Начнём с понятия оркестровки. Речь идёт о том, чтобы объединить ряд сервисов в единую систему по заранее определённой схеме. Логика соединения сервисов описана централизованно. Для повышения надёжности некоторые из составляющих систему сервисов могут быть развёрнуты в нескольких экземплярах. Задача агрегатора состоит в том, чтобы обращаться к каждому из этих дублирующих сервисов независимо и поставлять клиентам агрегированные данные от них.

В случае же хореографии сервисов логика взаимодействия сервисов распределена по всей системе без выделенного управляющего узла. Вызов, адресованный какому-либо сервису, может вызывать многочисленные обращения сервисов друг к другу, перед тем как данные достигнут пункта назначения. Организация взаимодействия микросервисов по принципу хореографии требует больше усилий, по сравнению с оркестровкой. Читатель может самостоятельно найти более подробную информацию по данной теме в сети.

Реактивный стиль запросов к веб-сервисам

Задача обработки веб-запросов хорошо подходит для реактивной модели программирования. Чаще всего в ответ на действие пользователя в графическом интерфейсе приложение посылает некоторый запрос на сервер. Обработывая этот запрос, сервис-агрегатор посылает множество асинхронных запросов микросервисам, ответственным за те или иные подзадачи. Из полученных от них ответов агрегатор собирает свой ответ, который отправляется пользовательскому интерфейсу. Все эти этапы обработки хорошо отображаются на понятия потока сообщений, наблюдаемого источника, фильтра и операции над потоками. В частности, для обработки асинхронных веб-запросов можно пользоваться библиотекой RxCurl.

Итоги

В этой главе было рассмотрено применение реактивной модели программирования для разработки реактивных микросервисов на языке C++. В частности, читатель узнал о программировании с использованием библиотеки REST SDK от корпорации Microsoft. В данной библиотеке воплощена асинхронная модель программирования, основанная на так называемых продолжениях задач. Для создания клиентов, работающих по протоколу REST, можно воспользоваться

библиотекой RxCurl, созданной Кирком Шупом, с некоторыми модификациями, направленными на поддержку запросов типа PUT и DELETE. В этой главе было показано, как в духе реактивного программирования реализовать REST-сервер и работающий с ним клиент.

Следующая глава будет посвящена методам обработки ошибок и исключений с помощью средств, предоставляемых библиотекой RxCpp.



Глава 12

.....

Особые возможности потоков и обработка ошибок

В этой книге рассмотрено уже немало вопросов, касающихся реактивного программирования на современном языке C++ и библиотеки RxCpp. Изложение начиналось со вспомогательных тем, знакомство с которыми необходимо перед погружением в мир реактивного программирования. Первые шесть глав были посвящены главным образом этим предварительным вопросам и постепенно знакомили читателя с основными понятиями и конструкциями реактивного программирования вообще и библиотеки RxCpp в частности. Термин «функциональное реактивное программирование» использовался в широком смысле, как применение методов функционального программирования для написания реактивных программ. Некоторые авторы настаивают на более строгой трактовке термина и не считают, что библиотеки из семейства Rx в полной мере воплощают модель функционального реактивного программирования. В любом случае, программисту нужно сделать над собой усилие и открыть своё сознание для парадигмы декларативного программирования.

Традиционный подход к программированию состоит в том, чтобы изобретать хитроумные структуры данных и алгоритмы их обработки. Эта методология хороша в тех случаях, когда программа манипулирует данными, развёрнутыми в пространстве. Если же ведущую роль в структуризации данных начинает играть время, естественным следствием этого становится асинхронный способ обработки¹. В реактивном программировании сложность структур данных обычно ограничивается одними лишь потоками, операции по обработке данных навешиваются на потоки, а все разнообразные способы обра-

¹ Это утверждение авторов отнюдь не бесспорно. Так, в системах реального времени, где требуется гарантированное время отклика на каждое входящее сообщение, асинхронная обработка с внутренне присущей ей недетерминированностью отнюдь не выглядит наилучшим выбором. Напротив, многие задачи обработки массивов (т. е. структур данных с очевидной пространственной структуризацией) превосходно подходят для распараллеливания: скажем, суммирование, поиск наименьшего элемента, некоторые алгоритмы сортировки. – *Прим. перев.*

ботки элементов данных реализуются посредством единого механизма – оповещения о событии с вызовом заданного действия. Читатель мог убедиться, насколько этот подход упрощает программирование на языке C++ графических интерфейсов, распределённых систем и даже консольных приложений.

В примерах из предыдущих глав для краткости не предусматривалась обработка исключений и, шире, каких-либо ошибок. Это упрощение было сделано намеренно, так как давало возможность сосредоточить внимание на логике приложения, его ключевых деталях и их взаимодействии. Теперь, когда все основополагающие аспекты реактивного программирования рассмотрены, пора сосредоточить внимание на средствах обработки ошибок. Прежде чем погрузиться в обработку ошибок и исключений, необходимо сделать краткий обзор важных характеристик реактивных систем. В этой главе будут рассмотрены следующие вопросы:

- 1) обобщённый обзор характеристик реактивных систем;
- 2) операции для обработки ошибок в библиотеке RxCpp;
- 3) взаимодействие средств обработки ошибок с планировщиком;
- 4) примеры систем, основанных на обработке потоков событий.

ОСНОВНЫЕ ХАРАКТЕРИСТИКИ РЕАКТИВНЫХ СИСТЕМ

Современный мир, как никогда прежде, требует от информационных систем надёжности, масштабируемости и скорости отклика. Понятие реактивного программирования сформировалось именно под давлением этой всё возрастающей потребности. Согласно «Манифесту реактивных систем»¹ (<https://www.reactive-manifesto.org/>), реактивные системы обладают следующими характеристиками.

- **Доступные:** система отвечает своевременно, если это вообще возможно. Доступность является краеугольным камнем удобного и полезного приложения, но, помимо этого, она позволяет быстро обнаруживать проблемы и эффективно их устранять. Доступные системы ориентированы на обеспечение быстрого и согласованного времени отклика, устанавливая надежные верхние границы, чтобы обеспечить стабильное качество обслуживания. Такое предсказуемое поведение, в свою очередь, упрощает обработку ошибок, повышает уверенность конечного пользователя в работоспособности и способствует дальнейшему взаимодействию с системой.
- **Устойчивые:** система остается доступной даже в случае отказов. Это относится не только к высокодоступным, критически важным приложениям – без устойчивости любая система при сбое теряет доступность. Устойчивость достигается за счет репликации, сдерживания, изоляции и делегирования. Эффект от отказов удерживается внутри компонентов, изолируя их друг от друга, что позволяет им выходить из строя и восстанавливаться, не нарушая работу системы в целом. Восстановление каждого компонента делегируется другому (внешнему) модулю, а высокая

¹ Цитируемый ниже русский перевод Манифеста взят с официального сайта. – *Прим. перев.*

доступность обеспечивается за счет репликации там, где это необходимо. Клиент компонента не отвечает за обработку его сбоев.

- **Гибкие:** система остается доступной под разными нагрузками. Реактивные системы способны реагировать на колебания в скорости входящих потоков, увеличивая или уменьшая количество выделенных на их обслуживание ресурсов. Для этого архитектура не должна допускать наличия централизованных узких мест или конкуренции за ресурсы, что позволяет сегментировать или реплицировать компоненты, распределяя между ними входные данные. Реактивные системы поддерживают предсказывающие и реактивные алгоритмы масштабирования, позволяя делать измерения производительности в режиме реального времени. Гибкость достигается применением экономически эффективных аппаратных и программных платформ.
- **Основаны на обмене сообщениями:** реактивные системы используют асинхронный обмен сообщениями, чтобы установить границы между компонентами и обеспечить слабую связанность, изоляцию и прозрачность размещения. Эти границы также позволяют преобразовывать и передавать информацию о сбое в виде сообщений. Открытый обмен сообщениями делает возможными регулирование нагрузки, гибкость и управление потоком, для чего в системе создаются и отслеживаются очереди сообщений и в случае необходимости используется обратное давление. Прозрачность размещения при взаимодействии на основе сообщений позволяет применять к механизму обработки ошибок одни и те же ограничения и семантику как в пределах одного компьютера, так и в масштабах целого кластера. Благодаря неблокирующему взаимодействию принимающая сторона потребляет ресурсы только при активной работе, что позволяет снизить накладные расходы.



Принципы реактивных систем применяются на всех уровнях, что позволяет компоновать их между собой.

В этой главе речь будет идти в основном об устойчивости реактивных систем, для достижения которой нужны средства обработки ошибок.

СРЕДСТВА ОБРАБОТКИ ОШИБОК В БИБЛИОТЕКЕ RxCPP

Ни одна программная система в реальном мире не бывает совершенной. Как отмечалось в предыдущем разделе, устойчивость составляет одно из важнейших качеств реактивной системы. То, насколько хорошо система умеет обрабатывать ошибки и восстанавливаться после сбоев, определяет её успех в будущем. Раннее обнаружение и незаметная извне обработка ошибок делают систему стройной и повышают её доступность для пользователей. По сравнению с императивным подходом, модель реактивного программирования позволяет лучше отделить логику нормальной работы приложения от логики обнаружения ошибки и обработки исключения.

В этой главе будет рассмотрено, как обрабатывать исключения и ошибки с помощью библиотеки RxCpp. В библиотеке RxCpp определено множество операций, позволяющих реагировать на оповещения типа `on_error`. Например, с оповещением об ошибке можно сделать следующее:

- красиво (т. е. выполнив все необходимые завершающие действия) прекратить дальнейшую обработку сообщений из данного источника;
- проигнорировать сигнал ошибки и для дальнейшей выборки данных переключиться на резервный источник;
- проигнорировать ошибку и вместо неполученных данных подставить объект-заглушку;
- проигнорировать ошибку и немедленно перезапустить источник данных;
- проигнорировать ошибку и попытаться перезапустить источник данных, дав ему некоторое время на самовосстановление.

Все эти механизмы обработки возможны благодаря тому, что интерфейс наблюдателя (`observable`) содержит три метода-обработчика:

- для обработки очередного элемента данных;
- для извещения о том, что в источнике больше нет и не будет данных;
- для извещения о том, что в источнике данных произошёл сбой.

Метод `on_error` как раз и предназначен для обработки исключений, которые могут возникнуть как в самом источнике, так и в какой-либо из навешенных на него операций. Напомним сигнатуры трёх методов объекта-наблюдателя:

- `void observer::on_next(T);`
- `void observer::on_completed();`
- `void observer::on_error(std::exception_ptr);`

Выполнение действия в ответ на ошибку

Когда в источнике данных возникает ошибка, её нужно обработать, оставив систему в целом в корректном состоянии. Примеры использования библиотеки

RxCpp, рассмотренные в предыдущих главах, содержали на стороне подписчика лишь обработчики для сценариев `on_next` и `on_completed`. Однако у функции `subscribe` (подписаться) есть ещё один параметр – это функция-обработчик для сценария `on_error`. Рассмотрим простую программу, демонстрирующую подписку не только на данные, но и на ошибки.

```
#include "rxcpp/rx.hpp"
int main() {
    using namespace std;
    using namespace rxcpp;

    // создать источник данных, выдающий ошибку
    auto vals = observable<>::range(1, 3).concat(
        observable<>::error<int>(runtime_error(
            "Ошибка в источнике!")));
    vals.subscribe(
        [] (int v) { printf("OnNext: %d\n", v); },
        [] (exception_ptr ep) {
            printf("OnError: %s\n", util::what(ep).c_str());
        },
        [] () { printf("OnCompleted\n"); });
    return 0;
}
```



Второе из трёх лямбда-выражений, переданных в качестве аргументов методу `subscribe`, представляет собой действие, которое подписчик должен выполнить, реагируя на сигнал ошибки. Результатом выполнения данной программы станет следующий текст:

```
OnNext: 1
OnNext: 2
OnNext: 3
OnError: Ошибка в источнике!
```

В показанном выше примере источник данных сначала генерирует «нормальные» данные, затем сам добавляет к потоку данных сигнал ошибки. Теперь посмотрим, как исключение распространяется вдоль потока данных сквозь промежуточную сущность.

```
#include "rxcpp/rx.hpp"
int main() {
    rxcpp::rxsub::subject<int> sub;
    auto subscriber = sub.get_subscriber();
    auto observable = sub.get_observable();

    observable.subscribe(
        [] (int v) { printf("OnNext: %d\n", v); },
        [] (std::exception_ptr ep) {
            printf(
                "OnError: %s\n",
                rxcpp::util::what(ep).c_str());
        },
        [] () { printf("OnCompleted\n"); });
}
```



Показанный выше фрагмент кода создаёт объект-тему (subject) – объект, обладающий свойствами одновременно наблюдателя и наблюдаемого источника (см. главу 8). На наблюдаемую сторону этого объекта подписываются три функции-обработчика, которые просто печатают соответствующие сообщения на консоль. Другая же сторона объекта, выступающая наблюдателем, используется ниже для того, чтобы наполнять этот поток данными и ошибками.

```
for (int i = 1; i <= 10; ++i) {
    if (i > 5) {
        try {
            std::string().at(1);
        } catch (...) {
            auto eptr = std::current_exception();
            subscriber.on_error(eptr);
        }
        subscriber.on_next(i * 10);
    }
    subscriber.on_completed();
    return 0;
}
```

Вызов метода `on_next` у объекта-подписчика (subscriber) помещает в поток новое значение. Однако поток отвергнет попытку поместить очередное значение, если для него уже выполнялась функция `on_completed`, сигнализирующая о нормальном завершении потока, или функция `on_error`, оповещающая о завершении с ошибкой. Подобным же образом объект-тема проигнорирует попытку нормального завершения потока (вызов метода `on_completed`), если для этого потока уже вызван метод `on_error`. Если поток данных переходит в ошибочное состояние, никакие «нормальные» данные через него уже проходить не могут.

Восстановление после ошибки

Как было показано в предыдущем разделе, возникновение ошибки в потоке данных прекращает дальнейшую передачу данных по нему, а клиенты, подписанные на этот поток, получают возможность выполнить какие-либо действия в качестве реакции на ошибку. Однако в некоторых случаях может оказаться желательным восстановить нормальное функционирование потока данных даже после возникшей в нём ошибки. Для этого служит функция `on_error_resume_next`, её применение иллюстрирует следующий пример.

```
#include "rxcpp/rx.hpp"
int main() {
    using namespace std;
    using namespace rxcpp;

    // поток данных с ошибкой
    auto values = observable<>::range(1, 3).concat(
```

```

observable<>::error<int>(runtime_error(
    "Ошибка в источнике!"))
// восстановление и продолжение потока
.on_error_resume_next([] (exception_ptr ep) {
    printf(
        "Восстановление после: %s\n",
        rxcpp::util::what(ep).c_str());
    return rxcpp::observable<>::range(4,6);
});

values.subscribe(
    [] (int v) { printf("OnNext: %d\n", v); },
    [] (std::exception_ptr ep) {
        printf(
            "OnError: %s\n",
            rxcpp::util::what(ep).c_str());
    },
    [] () { printf("OnCompleted\n"); });
return 0;
}

```

Метод `on_error_resume_next` наблюдаемого источника задаёт функцию-обработчик, которая будет автоматически вызываться на стороне источника при возникновении в нём ошибки. Как показано в данном примере, обработчик возвращает новый поток данных – именно из него будут брать дальнейшие данные. Таким образом, подписчик вообще не заметит ошибки в потоке данных – обработка ошибки и переключение на резервный источник будут осуществлены самим потоком. В результате выполнения представленной выше программы будет напечатан следующий текст:

```

OnNext: 1
OnNext: 2
OnNext: 3
Восстановление после: Ошибка в источнике!
OnNext: 4
OnNext: 5
OnNext: 6
OnCompleted

```

Этот подход позволяет наблюдаемому источнику в случае ошибки не только продолжить генерацию потока данных, но и заменить сигнал ошибки на единственный специальный объект данных. Скажем, в предыдущем примере можно было бы заменить обработчик ошибки следующим образом:

```

.on_error_resume_next([] (exception_ptr ep) {
    printf(
        "Восстановление после: %s\n",
        rxcpp::util::what(ep).c_str());
    return rxcpp::observable<>::just(-1);
});

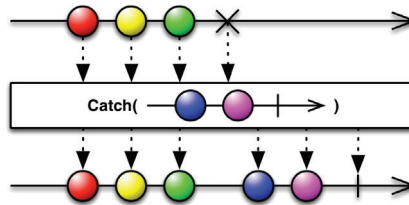
```

После такой замены результат работы программы выглядел бы так:

```
OnNext: 1
OnNext: 2
OnNext: 3
Восстановление после: Ошибка в источнике!
OnNext: -1
OnCompleted
```



Работу операции `on_error_resume_next` иллюстрирует следующая диаграмма.



Эта функция перехватывает ошибку, возникшую в наблюдаемом источнике, и вместо испорченного ошибкой подставляет новый источник данных, тем самым создавая у потребителей иллюзию, что никакой ошибки не было.

Операция `on_error_resume_next` приходит на помощь в различных ситуациях, когда программисту нужно модифицировать распространение ошибки вдоль потока данных. Например, между генерацией первоначальных данных и потреблением данных подписчиками поток может подвергаться различным преобразованиям и фильтрациям. Как разъяснялось в главе 9, это могут быть операции, определённые самим программистом на основе операций, предоставляемых библиотекой. В таких случаях может оказаться полезным использовать операцию `on_error_resume_next` на каждом промежуточном этапе обработки потока данных. Назначаемый этой функцией обработчик может подставить в поток не только «запасную» последовательность данных или специальное значение, но и, в свою очередь, новый сигнал ошибки, который и придёт подписчику. Рассмотрим, например, следующий фрагмент кода:

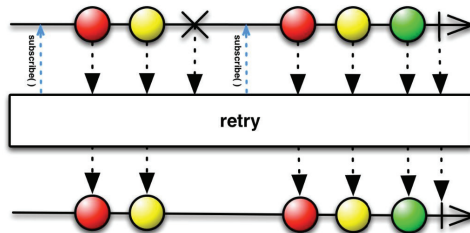
```
auto processed_strm = Source_observable
    .map([] (const string& s) { return fff(s); })
    // преобразовать исключение
    .on_error_resume_next([] (std::exception_ptr) {
        return rxcpp::sources::error<string>(
            runtime_error(rxcpp::util::what(ep).c_str()));
    });
```

К каждому элементу исходного потока данных применяется функция-преобразователь `fff`. Если на каком-либо элементе в ней происходит исключение, оно перехватывается с помощью функции `on_error_resume_next`, и вместо него в поток помещается новое, преобразованное исключение.

Обработка ошибки путём перезапуска источника данных

Часто на практике ошибка в реактивном потоке данных оказывается следствием временной неполадки в генераторе сообщений. В таких случаях хотелось бы иметь возможность выждать, пока сбой не будет устранён на стороне генератора, и затем продолжить нормальную работу системы. В библиотеке RxCpp есть подходящее средство. Лучше всего оно подходит для случаев, когда о последовательности данных заранее известно, что в ней могут возникать кратковременные случайные ошибки.

Операция `retry` (англ. – попытаться заново) реагирует на оповещение `on_error` от источника данных. Вместо того чтобы передавать оповещение об ошибке дальше, подписчикам, эта операция заново подписывается на тот же самый наблюдаемый источник. Это даёт наблюдаемому источнику новый шанс завершить свою работу, без ошибок выдав все данные. Данные, выдаваемые наблюдаемым источником после его перезапуска, снова направляются подписчикам на их обработчики `on_next`. В зависимости от логики работы источника подписчики могут повторно получить данные, которые уже были обработаны ими ранее. Принцип действия данной операции иллюстрирует следующая диаграмма.



Ниже приведён пример программы, в которой используется операция `retry`.

```
#include "rxcpp/rx.hpp"
int main() {
    using namespace rxcpp;
    using namespace std;
    // три элемента, ошибка, повтор сначала
    auto values = observable<>::range(1, 3)
        .concat(observable<>::error<int>(runtime_error(
            "Ошибка в источнике!")))
        .retry()
        .take(5);

    // вывести содержимое потока
    values.subscribe(
        [] (int v) { printf("OnNext: %d\n", v); },
        [] () { printf("OnCompleted\n"); });
    return 0;
}
```



В начале программы создаётся поток данных, в котором после трёх элементов намеренно вызывается ошибка. Обычный метод обработки такого потока обнаружил бы аварийное завершение потока. Однако к потоку применена операция `retry`, которая игнорирует любые ошибки и всякий раз заставляет поток работать сначала. Тем самым для наблюдателя он будет выглядеть бесконечным повторением одних и тех же трёх элементов. Чтобы избежать бесконечной работы программы, ограничим длину потока операцией `take`. Ниже показан результат работы этой программы:

```
OnNext: 1
OnNext: 2
OnNext: 3
OnNext: 1
OnNext: 2
OnCompleted
```

Рассмотренная выше операция перезапускает поток неограниченно много раз, сколько бы ошибок в нём ни возникло. Часто бывает желательно ограничить число таких повторных попыток. Для этого служит перегруженная версия функции `retry`, принимающая один целочисленный аргумент. Её применение показано в следующем примере.

```
#include "rxcpp/rx.hpp"
int main() {
    using namespace rxcpp;
    using namespace std;
    // три элемента, ошибка, повтор сначала
    auto values = observable<>::range(1, 3)
        .concat(observable<>::error<int>(runtime_error(
            "Ошибка в источнике!")))
        .retry(2);

    // вывести содержимое потока
    values.subscribe(
        [] (int v) { printf("OnNext: %d\n", v); },
        [] (std::exception_ptr ep) {
            printf(
                "OnError: %s\n",
                rxcpp::util::what(ep).c_str());
        },
        [] () { printf("OnCompleted\n"); });
    return 0;
}
```

Результат работы этой программы таков:

```
OnNext: 1
OnNext: 2
OnNext: 3
OnNext: 1
OnNext: 2
OnNext: 3
OnError: Ошибка в источнике!
```

Автоматическое выполнение завершающих действий в случае ошибки



Как явствует из предыдущих примеров, при генерации потока данных в библиотеке RxCpp может возникнуть исключение, при этом библиотека обеспечивает корректное завершение потока и оповещение подписчиков о произошедшей ошибке. Операция `finally` может оказаться полезной, если поток данных пользуется некоторыми внешними ресурсами и обязан освободить их при завершении своей работы, будь оно нормальным или аварийным. Очевидно, что на языке C++ написаны миллиарды строк кода различных систем, и многие из них обладают собственными механизмами управления ресурсами. Поэтому для использования технологии реактивного программирования совместно со старым кодом необходимо как-то заставить реактивный код чистить за собой ресурсы, полученные из других модулей. Именно для этого служит операция `finally`.

```
#include "rxcpp/rx.hpp"
int main() {
    using namespace rxcpp;
    using namespace std;
    auto values = observable<>::range(1, 3)
        .concat(observable<>::error<int>(runtime_error(
            "Ошибка в источнике!")))
    // это действие выполнится при завершении потока
    .finally([] () { printf("Завершающее действие\n"); });

    values.subscribe(
        [] (int v) { printf("OnNext: %d\n", v); },
        [] (std::exception_ptr ep) {
            printf(
                "OnError: %s\n",
                rxcpp::util::what(ep).c_str());
        },
        [] () { printf("OnCompleted\n"); });
    return 0;
}
```



Операция `finally` присоединяет к наблюдаемому источнику данных функцию-обработчик и гарантирует, что она будет выполнена после того, как поток завершится (нормальным образом или с ошибкой) и подписчики получают соответствующее оповещение. В результате выполнения этой программы на консоль выводится следующий текст:

```
OnNext: 1
OnNext: 2
OnNext: 3
OnError: Ошибка в источнике!
Завершающее действие
```

Убедимся, что завершающее действие, назначенное для потока, выполняется и в том случае, когда поток завершается без ошибки. Для этого уберём из

кода программы обращение к методу `concat`, добавляющему в поток ошибку. Тогда результат работы программы станет следующим:

```
OnNext: 1
OnNext: 2
OnNext: 3
Завершающее действие
```



ОБРАБОТКА ОШИБОК И ПЛАНИРОВЩИКИ

Планировщики как одна из важнейших составных частей реактивных систем рассматривались в главе 8. Планировщик занимается постановкой генерируемых данных в очередь и их доставкой потребителям для обработки в соответствии с определённой дисциплиной координации выполнения. А именно обработка может осуществляться в текущем потоке выполнения, в главном цикле обработки событий библиотеки RxCpp или в новом потоке. Программист может в известной степени управлять работой планировщика посредством операций `observe_on`, `subscribe_on` и некоторых других. В качестве аргумента эти операции принимают дисциплину координации. По умолчанию библиотека RxCpp работает в однопоточном режиме. Программисту нужно явно указывать, если операции требуется выполнять в отдельном потоке. Рассмотрим следующий код:

```
#include "rxcpp/rx.hpp"
#include <iostream>
#include <thread>

int main() {
    auto values = rxcpp::observable<>::range(1, 4)
        .map([] (int v) { return v*v; })
        .concat(rxcpp::observable<>::error<int>(
            std::runtime_error("Ошибка в источнике!")));

    // узнать идентификатор главного потока
    std::cout
        << "Главный поток => "
        << std::this_thread::get_id()
        << std::endl;
```

Здесь создаётся поток данных на основе заданного диапазона, затем каждое число из этого потока возводится в квадрат. В конец потока намеренно добавлена генерация ошибки. Это позволит продемонстрировать, как механизмы обработки ошибок взаимодействуют с планировщиками в библиотеке RxCpp.

```
// наблюдатель работает в другом потоке
values
    .observe_on(rxcpp::synchronize_new_thread())
    .as_blocking()
    .subscribe(
        [] (int v) {
            std::cout
```

```

        << "Поток наблюдателя => "
        << std::this_thread::get_id()
        << " "
        << v
        << std::endl;
    },
    [] (std::exception_ptr ep) {
        printf(
            "OnError: %s\n",
            rxcpp::util::what(ep).c_str());
    },
    [] () { std::cout << "OnCompleted" << std::endl; });

// снова напечатать идентификатор главного потока
std::cout
    << "Главный поток => "
    << std::this_thread::get_id()
    << std::endl;
return 0;
}

```

Благодаря операции `observe_on` обработка данных из этого источника происходит в отдельном потоке выполнения. Как и в предыдущих примерах, подписчик содержит функцию-обработчик ошибок. Эта программа должна напечатать текст, подобный следующему (конечно же, идентификаторы потоков выполнения будут различаться):

```

Главный поток => 5776
Поток наблюдателя => 12184 1
Поток наблюдателя => 12184 4
Поток наблюдателя => 12184 9
Поток наблюдателя => 12184 16
OnError: Ошибка в источнике!
Главный поток => 5776

```

Посмотрим теперь, как ведёт себя программа, если к источнику подключить двух наблюдателей, каждый из которых работает в своём потоке.

```

#include "rxcpp/rx.hpp"
#include <mutex>
std::mutex printMutex;

int main() {
    rxcpp::rxsub::subject<int> sub;
    auto subscriber = sub.get_subscriber();
    auto observable1 = sub.get_observable();
    auto observable2 = sub.get_observable();

```



В этом фрагменте кода создаётся объект-тема, т. е. объект, выступающий одновременно наблюдаемым источником и наблюдателем. Из объекта создаются один интерфейс наблюдателя и два интерфейса наблюдаемого источника, каждому из которых предстоит работать в своём потоке.

```

auto onNext = [] (int v) {
    std::lock_guard<std::mutex> lock(printMutex);
    std::cout << "Поток источника => "
        << std::this_thread::get_id()
        << "\tOnNext: " << v << std::endl;
};

auto onError = [] (std::exception_ptr ep) {
    std::lock_guard<std::mutex> lock(printMutex);
    std::cout << " Поток источника => "
        << std::this_thread::get_id()
        << "\tOnError: "
        << rxcpp::util::what(ep).c_str() << std::endl;
};

```

Эти две лямбда-функции будут в дальнейшем подписаны на оповещения о данных и об ошибках. Чтобы символы, выводимые ими на консоль из разных потоков, не смешивались друг с другом, доступ к стандартному устройству вывода синхронизирован с помощью глобального семафора.

```

// подписка в отдельном потоке
observable1.
    observe_on(rxcpp::synchronize_new_thread()).
    subscribe(onNext, onError,
        [] () { printf("OnCompleted\n"); });

// ещё одна подписка в другом потоке
observable2.
    observe_on(rxcpp::synchronize_event_loop()).
    subscribe(onNext, onError,
        [] () { printf("OnCompleted\n"); });

```

На два наблюдаемых источника, полученных из объекта-темы, подписываются два наблюдателя, причём каждый наблюдатель работает в своём потоке. А именно для первого наблюдателя создаётся отдельный поток, тогда как второй наблюдатель работает в цикле обработки событий библиотеки RxCpp.

```

// наполнение потока значениями и ошибками
for (int i = 1; i <= 10; ++i) {
    if (i > 5) {
        try {
            std::string().at(1);
        }
        catch (...) {
            auto eptr = std::current_exception();
            subscriber.on_error(eptr);
        }
    }
    subscriber.on_next(i * 10);
}
subscriber.on_completed();

```

```
// подождать две секунды
rxcpp::observable<>::timer(std::chrono::milliseconds(2000)).
    subscribe([&](long) {});
return 0;
}
```

В этой части программы данные вталкиваются в наблюдаемый источник через связанный с ним интерфейс наблюдателя. Также этот генератор намеренно провоцирует исключение и передаёт его в поток данных. В результате работы этой программы получается следующий текст, из которого видно, как механизмы обработки ошибок взаимодействуют с планировщиком.

```
Поток источника => 2644   OnNext: 10
Поток источника => 2304   OnNext: 10
Поток источника => 2644   OnNext: 20
Поток источника => 2304   OnNext: 20
Поток источника => 2644   OnNext: 30
Поток источника => 2304   OnNext: 30
Поток источника => 2644   OnNext: 40
Поток источника => 2304   OnNext: 40
Поток источника => 2304   OnNext: 50
Поток источника => 2304   OnError: invalid string position
Поток источника => 2644   OnNext: 50
Поток источника => 2644   OnError: invalid string position
```

Приведённый выше пример показывает, как происходит рассылка оповещений двум различным подписчикам, подключенным к одному общему генератору данных через промежуточный объект-тему и работающим в разных потоках. Теперь посмотрим, как планировщик пересылает сигналы об ошибках в случае операции `subscribe_on`.

```
#include "rxcpp/rx.hpp"
#include <thread>
#include <mutex>
std::mutex printMutex;

int main() {
    // создание потоков данных
    auto values1 = rxcpp::observable<>::range(1, 4)
        .transform([](int v) { return v * v; });
    auto values2 = rxcpp::observable<>::range(5, 9)
        .transform([](int v) { return v * v; });
    .concat(rxcpp::observable<>::error<int>(
        std::runtime_error("Ошибка в источнике!")));
```



Здесь показано создание двух наблюдаемых источников данных, причём второй источник в конце генерирует сигнал ошибки.

```
// запланировать на выполнение в отдельном потоке
auto s1 = values1.subscribe_on(
    rxcpp::observe_on_event_loop());
// запланировать в ещё одном потоке
```

```
auto s2 = values2.subscribe_on(
    rxcpp::synchronize_new_thread());
```

Этот фрагмент кода заставляет планировщик выполнять генерацию данных для этих двух источников в двух отдельных потоках выполнения. А именно данные для первого источника генерируются в потоке цикла обработки событий, а для второго источника создаётся новый поток.

```
auto onNext = [](int v) {
    std::lock_guard<std::mutex> lock(printMutex);
    std::cout << "Поток => "
                << std::this_thread::get_id()
                << "\tOnNext: " << v << std::endl;
};

auto onError = [](std::exception_ptr ep) {
    std::lock_guard<std::mutex> lock(printMutex);
    std::cout << "Поток => "
                << std::this_thread::get_id()
                << "\tOnError: "
                << rxcpp::util::what(ep).c_str() << std::endl;
};
```

Эти две лямбда-функции будут использованы ниже в качестве обработчиков, подписанных на оповещения (как о данных, так и об ошибках) от наблюдаемого источника. Вывод в консоль защищён семафором от одновременного выполнения параллельными потоками. В оставшейся части программы новый наблюдаемый источник строится путём соединения первых двух, и на него подписывается один наблюдатель.

```
// подписка на объединённый источник данных
s1.merge(s2).as_blocking().subscribe(
    onNext, onError,
    []() { std::cout << "OnCompleted" << std::endl; });

// напечатать идентификатор главного потока
std::cout << "Главный поток => "
            << std::this_thread::get_id()
            << std::endl;
return 0;
}
```

Результат работы этой программы (не считая конкретных значений идентификаторов и порядка выполнения потоков) будет таким:

```
Поток => 12380  OnNext: 1
Поток => 9076   OnNext: 25
Поток => 12380  OnNext: 4
Поток => 9076   OnNext: 36
Поток => 12380  OnNext: 9
Поток => 12380  OnNext: 16
Поток => 9076   OnNext: 49
```



```

Поток => 9076      OnNext: 64
Поток => 9076      OnNext: 81
Поток => 9076      OnError: Ошибка в источнике!
Главный поток => 10692

```

ПРИМЕРЫ ОБРАБОТКИ ПОТОКОВ СОБЫТИЙ

Прежде чем завершать эту главу, рассмотрим на паре примеров разработку событийно-управляемых систем с помощью библиотеки RxCpp. Эти примеры должны продемонстрировать, насколько эффективной может оказаться библиотека RxCpp при решении задач из реального мира. Наши примеры будут заниматься агрегированием данных в поток и обработкой прикладных событий.

Агрегирование потоков данных

В этом разделе элементами потока данных будут объекты пользовательского типа, моделирующего понятие сотрудника. Основная задача приложения состоит в том, чтобы сгруппировать сотрудников по должности и по зарплате.

```

#include "rxcpp/rx.hpp"
namespace Rx {
    using namespace rxcpp;
    using namespace rxcpp::sources;
    using namespace rxcpp::subjects;
    using namespace rxcpp::util;
}

using namespace std;

struct Employee {
    string name;
    string role;
    int salary;
};

```

В показанном выше фрагменте объявляется удобный для частого использования краткий псевдоним для нескольких пространств имён из библиотеки RxCpp и структура данных, представляющая информацию о сотруднике. У этого структурного типа все поля открыты, а зарплата сделана целым числом.

```

int main() {
    Rx::subject<Employee> employees;

    // группировать по зарплате
    auto role_sal = employees
        .get_observable()
        .group_by(
            [](Employee& e) { return e.role; },
            [](Employee& e) { return e.salary; });
}

```

В функции `main` создаётся объект-тема с целью запустить *горячий* источник данных типа `Employee`. Данные, получаемые из этого источника, будут группироваться по должности и зарплате. Для этого используется определённая в библиотеке `RxCpp` операция `group_by`, её результатом является наблюдаемый источник наблюдаемых источников, каждый из которых, в свою очередь, содержит элементы исходного источника данных, имеющие одинаковое значение ключа.

```
// комбинированная свёртка по трём операциям:
// наименьшее, наибольшее, среднее
auto result = role_sal
    .map([] (Rx::grouped_observable<string, int> group) {
        return group
            .count()
            .combine_latest(
                [=](int count, int min, int max, double average) {
                    return make_tuple(group.get_key(), count, min, max, average);
                },
                group.min(),
                group.max(),
                group.map([] (int salary) -> double {
                    return salary;
                }).average());
    })
    .merge();
```

Созданный ранее поток групп, объединённых по критериям должности и зарплаты, подвергается здесь дальнейшему преобразованию: для каждой должности вычисляется наименьшая, наибольшая и средняя зарплата. Лямбда-функция, стоящая под управлением операции `combine_latest`, вызывается тогда, когда известны значения всех её аргументов. В данном случае это означает, что когда становится готова группа (объект `group`), к ней независимо друг от друга применяются три операции-свёртки: `min` для поиска наименьшего значения, `max` для поиска наибольшего и `average` для нахождения среднего. Когда эти три значения вычислены, вызывается эта лямбда-функция. Таким образом, она вызывается по одному разу для каждой должности и набор всех сотрудников, имеющих эту должность, сворачивает до одного объекта со сводными данными по всей должности. Далее, поскольку всё это преобразование стоит под управлением функции `map` и применяется к потоку групп, её результатом становится наблюдаемый источник наблюдаемых источников, т. е. объектов типа

```
observable<tuple<string, int, int, int, double>>
```

Наконец, операция слияния `merge` соединяет все эти наблюдаемые источники в один. Следовательно, результат всего выражения есть наблюдаемый источник объектов типа

```
tuple<string, int, int, int, double>
```

Операция слияния нужна в том числе и для того, чтобы предотвратить возможную потерю данных: промежуточные наблюдаемые источники, полученные в результате группировки, являются горячими, и их данные безвозвратно теряются, если нет подписчика, готового их получить.

```
// отобразить агрегированные данные
result
    .subscribe(Rx::apply_to(
        [](string role, int count, int min, int max, double avg) {
            std::cout
                << role.c_str()
                << ":\tчисленность "
                << count
                << ", зарплаты ["
                << min
                << "- "
                << max
                << "], средняя "
                << avg
                << endl;
        }
    ));

// тестовые данные на вход
Rx::observable<>::from(
    Employee{ "Джон", "Инженер", 60000 },
    Employee{ "Тирион", "Менеджер", 120000 },
    Employee{ "Арья", "Инженер", 92000 },
    Employee{ "Санса", "Менеджер", 150000 },
    Employee{ "Серсея", "Бухгалтер", 76000 },
    Employee{ "Джейми", "Инженер", 52000 }
).subscribe(employees.get_subscriber());

return 0;
}
```

На построенный выше поток данных подписывается наблюдатель, который выводит результаты агрегирования на консоль. Наконец, чтобы привести систему в действие, в исходный поток данных помещаются учетные записи сотрудников. В реальном приложении это могут быть данные, вычитываемые из файла, получаемые по сети или вырабатываемые параллельным потоком выполнения. Запуск данной программы должен дать следующий результат:

```
Бухгалтер: численность 1, зарплаты [76000-76000], средняя 76000
Инженер:   численность 3, зарплаты [52000-92000], средняя 68000
Менеджер:  численность 2, зарплаты [120000-150000], средняя 135000
```

Событийно-управляемое приложение

Этот пример представляет собой консольное приложение, которое обрабатывает события, представляющие примитивные операции пользовательского интерфейса. Для обработки используется библиотека RxCpp. Это приложение

можно было бы сделать и с графическим пользовательским интерфейсом, но для краткости кода оставим его консольным.

```
#include <rxcpp/rx.hpp>
#include <cassert>
#include <cctype>
#include <clocale>

namespace Rx {
    using namespace rxcpp;
    using namespace rxcpp::sources;
    using namespace rxcpp::operators;
    using namespace rxcpp::util;
    using namespace rxcpp::subjects;
}
```

```
using namespace Rx;
using namespace std::chrono;
```

```
// коды событий
enum class AppEvent {
    Active,
    Inactive,
    Data,
    Close,
    Finish,
    Other
};
```



В начале листинга расположены, как обычно, директивы включения заголовочных файлов, псевдонимы для пространств имён и объявление перечислимого типа, представляющего коды различных событий, с которыми имеет дело программа.

```
int main()
{
    //-----
    // A или a - Active   - активный
    // I или i - Inactive - неактивный
    // D или d - Data     - данные
    // C или c - Close    - закрыть
    // F или f - Finish   - завершить
    // прочие - Other     - прочие
    auto events = Rx::observable<>::create<AppEvent>(
        [](Rx::subscriber<AppEvent> dest) {
            std::cout << "Введите команду:\n";
            for (;;) {
                int key = std::cin.get();
                AppEvent current_event = AppEvent::Other;

                switch (std::tolower(key)) {
                    case 'a': current_event = AppEvent::Active; break;
```

```

    case 'i': current_event = AppEvent::Inactive; break;
    case 'd': current_event = AppEvent::Data; break;
    case 'c': current_event = AppEvent::Close; break;
    case 'f': current_event = AppEvent::Finish; break;
    default: current_event = AppEvent::Other;
  }

  if (current_event == AppEvent::Finish) {
    dest.on_completed();
    break;
  }
  else {
    dest.on_next(current_event);
  }
}
}).
on_error_resume_next([](std::exception_ptr ep) {
    return rxcp::observable<>::just(AppEvent::Finish);
}).
publish();

```

В этом фрагменте кода создаётся наблюдаемый источник данных типа `AppEvent`, т. е. поток кодов событий. Для первоначальной генерации данных используется ввод с консоли. Лямбда-функция в бесконечном цикле ожидает ввода символа с клавиатуры и превращает символ в код события. Эта лямбда-функция выполняет в данном консольном приложении ту же роль, которую в графических приложениях играет главный цикл обработки событий. Операция `publish` превращает холодный источник данных в горячий и устраняет зависимость источника данных от подключенных к нему наблюдателей. Это также означает, что опоздавшим подписчикам всегда будет доставляться самое свежее событие, а не вся предыстория.

```

// Фильтры событий:
// вход в активный режим
auto appActive = events.
    filter([](AppEvent const& event) {
        return event == AppEvent::Active;
    });

// выход из активного режима
auto appInactive = events.
    filter([](AppEvent const& event) {
        return event == AppEvent::Inactive;
    });

// передача данных
auto appData = events.
    filter([](AppEvent const& event) {
        return event == AppEvent::Data;
    });

```

```
// закрытие приложения
auto appClose = events.
    filter([](AppEvent const& event) {
        return event == AppEvent::Close;
    });
```



Выше создано несколько дополнительных источников данных, полученных путём отбора из исходного источника только событий определённого типа. Например, объект `appActive` представляет собой поток событий, состоящий исключительно из пришедших от пользователя событий типа `AppEvent::Active`.

```
auto dataFromApp = appActive
    .map([=] (AppEvent const& event) {
        std::cout
            << "**Вход в активный режим**\n"
            << std::flush;
        // собрать все события передачи данных
        // до выхода из активного режима
        return appData
            .take_until(appInactive)
            .finally([] () {
                std::cout << "**Выход из активного**\n";
            });
    })
    .switch_on_next() // обрабатывать только свежие события
    .take_until(appClose) // останов по событиям Finish/Close
    .finally([]() {
        std::cout << "**Завершение приложения**\n";
    });

dataFromApp.subscribe(
    [](AppEvent const& event) {
        std::cout << "**Данные приложения**\n" << std::flush;
    });

events.connect();

return 0;
}
```



Получив от пользовательского интерфейса событие `AppEvent::Active`, приложение переходит в активный режим и начинает принимать некие данные, что в нашем примере смоделировано событием `AppEvent::Data`, и делает это до тех пор, пока не получит событие `AppEvent::Inactive`, снова переводящее приложение в неактивный режим. Если в будущем приложение ещё раз получит событие `AppEvent::Active`, оно снова начнёт принимать данные. Если же от пользовательского интерфейса приходит событие `AppEvent::Close` или `AppEvent::Finish`, приложение завершается, однако при этом успевает выполнить определённые действия – например, освободить системные ресурсы.

Итоги



В этой главе речь шла об обработке ошибок средствами библиотеки RxCpp, а также о некоторых усложнённых подходах к обработке потоков событий. Главу открывал обзор ключевых характеристик, присущих реактивным системам в общем случае, затем особое внимание было уделено одной из них – устойчивости, для обеспечения которой требуются развитые средства обработки ошибок и сбоев. Рассматривался обработчик `on_error` – один из трёх, составляющих интерфейс реактивного подписчика. Затем были разобраны операции `on_error_resume_next`, `retry`, позволяющие продолжить функционирование подписчиков, несмотря на ошибку, возникшую в источнике данных. Также было рассказано об операции `finally`, которая гарантирует выполнение определённых завершающих действий независимо от того, завершился поток нормальным образом или с ошибкой. В заключение были рассмотрены две программы, иллюстрирующие усложнённые приёмы обработки потоков данных. Первая из них занималась разбиением на группы и последующим агрегированием потока объектов, а вторая имитировала работу сложного приложения, принимающего разнообразные команды.



Предметный указатель

A

accumulate, 105
ACE, 243
adopt_lock, 90
all, 213
amb, 213
Android, 270
any, 65
async, 100, 104
ATL, 23
atomic<>, 117
atomic<bool>, 113
atomic_flag, 111, 121
auto, 44
average, 182, 213

B

Boost.Range, 149
buffer, 211

C

C#, 44
catch, 212
clear, 111
combine_latest, 212
compare_exchange_strong, 114, 116
compare_exchange_weak, 114, 116
concat, 213
condition_variable, 72, 91, 96
condition_variable_any, 91
connect, 23, 214, 228
contains, 213
copy_if, 147
count, 182, 213
create, 210
curl, 274, 275, 276

D

deadlock, 87
debounce, 211
decltype, 45



default_if_empty, 213
defer, 210
DELETE, 270, 273, 277, 282, 290
detach, 76, 80
distinct, 211

E

element_at, 211
empty, 210
exchange, 116

F

F#, 159
fetch_add, 116, 123
fetch_sub, 116
filter, 211
finally, 212
first, 211
flat_map, 211
from, 210
function, 60
future, 72, 100, 101, 104

G

GET, 270, 272, 275, 277, 282, 285, 290
group_by, 211

H

Haskell, 44, 159
HTTP, 269, 272, 275, 276, 278, 292

I

ignore_elements, 211
interval, 210
IObservable, 26, 36, 67
IObserver, 26, 36, 67
iterator, 66

J

join, 75, 80
joinable, 76
JSON, 269, 278, 284
just, 210

L

last, 211
 launch, 105
 libcurl, 269, 276, 277
 Linux, 22, 70, 217, 222, 253, 270, 275
 Lisp, 61
 list, 147
 livelock, 98
 load, 116, 120
 lock, 89
 lock_guard, 72, 86, 90, 92, 94, 95
 lvalue, 48

M

macOS, 22, 70, 253, 270, 275
 map, 211
 max, 182, 213
 memory_order_acq_rel, 115, 118, 122
 memory_order_acquire, 112, 115, 118, 122
 memory_order_consume, 118
 memory_order_relaxed, 112, 115, 118, 122
 memory_order_release, 112, 115, 117, 118
 memory_order_seq_cst, 115, 118
 merge, 212
 min, 182, 213
 ML, 159
 move, 52, 81
 mutex, 72, 85, 91

N

never, 210
 notify_one, 94

O

observe_on, 212
 OMG, 244
 optional, 65

P

packaged_task, 102, 104
 POSE, 243
 POSIX, 19, 70, 73, 128, 156, 253, 275
 POST, 270, 282, 290
 postman, 274
 promise, 72, 100, 101
 publish, 214
 PUT, 270–277, 282, 286, 290

Q

QApplication, 223, 225
 QCloseEvent, 220
 QCoreApplication, 221
 QDialog, 227, 228
 QEvent, 220, 237
 QKeyEvent, 220
 QLabel, 225, 226, 230, 236
 QMetaClassInfo, 218
 QMetaEnum, 218
 QMetaMethod, 218
 QMetaObject, 218
 QMetaProperty, 218
 QMetaType, 218
 QMouseEvent, 225
 QMoveEvent, 220
 QObject, 218, 219, 222, 236
 Q_OBJECT, 22, 23, 222, 226–228
 QObjectCleanupHandler, 218
 QPointer, 218
 QSignalMapper, 218
 Qt, 22, 32, 37, 215–217, 221, 222, 232, 233, 245, 268
 QTimerEvent, 220
 QVariant, 218
 QVBoxLayout, 231
 QWidget, 220, 236

R

race condition, 83
 RaftLib, 170
 RAII, 86, 87
 range, 210
 Range-v3, 149
 reduce, 213
 ref, 79
 ref_count, 214
 repeat, 210
 replay, 214
 REST, 269, 282, 290
 retry, 212
 rvalue, 49
 RxCpp, 36, 39, 61, 127, 152–178, 182–184, 191, 194, 199–216, 232–241, 255, 256, 268, 292, 300–315
 RxCurl, 269, 277, 282, 290, 295

S

sample, 211
 Scala, 44
 scan, 211
 scope, 212
 sequence_equal, 213
 shared_future, 101
 shared_ptr, 52, 101, 123
 skip, 211
 skip_last, 211
 skip_until, 213
 skip_while, 213
 Spreadsheet, 168, 170
 stack, 93
 start_with, 212
 static_cast, 52
 store, 116, 120
 Streams, 159
 Streamulus, 162, 166, 167, 168
 subscribe, 212
 subscribe_on, 212
 sum, 182, 213
 switch_on_next, 212

T

take, 211
 take_last, 211
 take_until, 213
 take_while, 213
 terminate, 82
 test_and_set, 111, 120
 thread, 72, 80, 99
 throw, 210
 timer, 210

U

unique_lock, 72, 92, 96
 unique_ptr, 52, 101
 Unix, 61, 137, 183, 270
 UNIX, 156

V

variant, 64, 143
 vector, 147

W

weak_ptr, 52
 WebSocket, 195, 270

window, 211

Windows, 17, 61, 73, 215, 217, 223, 253, 270, 275

Z

zip, 212

А

Абстрактное синтаксическое
 дерево, 135
 Абстракция нулевой стоимости, 40
 Адвайта-веданта, 134
 Активный объект, 260
 Ананд, 134
 Аппликативный порядок
 вычислений, 158
 Атомарная операция
 запись, 111
 общая характеристика, 108
 чтение, 111
 чтение-модификация-запись, 111
 Атомарный тип, 108
 общая характеристика, 108
 указатель, 116
 целочисленный, 115

Б

Банда четырёх, 127, 133, 135, 142, 149
 Брахман, 134

В

Вариадический шаблон, 46
 Взаимозаменяемость, 43
 Вталкивание данных, 28
 Втягивание данных, 28
 Вывод типов, 44
 Выразительность, 40

Г

Гонка потоков, 83, 84, 85, 107

Д

Двойная диспетчеризация, 137, 138, 250, 255
 Декоратор, 149

З

Замыкание, 58



И

Инвариант, 83

К

Карринг, 57

Композиция функций, 57, 61

Критическая секция, 84

Л

Лямбда-функция, 54, 79

М

Многопоточное программирование, 70

Модель памяти, 106

Мьютекс, 84

Н

Наблюдаемый источник, 26, 27, 31, 127–136, 149, 151, 172–184, 193–195, 200, 203, 206, 210–213, 234–240, 248, 255–257, 265, 278, 291, 295, 302–307, 310–317

Наблюдатель, 12, 26–28, 31–36, 40, 65–67, 127–132, 136, 149, 150, 153, 172–186, 191–196, 199, 200, 206, 236, 237, 300, 302, 309–312, 317

Неблокирующая структура данных, 124

О

Обёртка над функцией, 60

Обещание, 100

Обработчик, 18, 24, 25, 67, 129, 138, 140, 153, 157, 164, 166, 172, 174, 206, 219–221, 227, 228, 233, 269, 272, 273, 278, 284–290, 301–305, 309

Оповещение, 26, 150

Ослабленный порядок доступа, 122

П

Параллельное программирование, 70, 71

Планировщик, 151, 172–177, 183–185, 192, 194, 203–208, 260, 308, 311, 312

Подписка, 150

Подписчик, 26, 28, 32, 67, 128–132, 150, 172, 174, 177–180, 185, 186, 194–209, 233, 234, 239, 240, 245, 301–319

Порядок доступа к памяти, 118

Последовательная согласованность, 119

Р

Разглаживание композита, 142

С

Сат, 134

Семантика захвата и освобождения, 120

Семантика перемещения, 50

Семафор, 84

Сигнал, 217–232, 238

Слот, 217–222, 225, 228

Событие, 26, 150

Состояние гонок, 83

Сумма типов, 64

Т

Тупик, 87

динамический, 98

У

Умный указатель, 52

Условная переменная, 76, 91, 93, 98, 108, 124

Ф

Функциональный объект, 55

Функция высшего порядка, 146

Фьючерс, 100

Ч

Частичное применение, 57

Чит, 134

Ш

Шаблон проектирования, 127

итератор, 128, 135, 142

композит, 128, 135, 142

наблюдатель, 127, 128, 131–136, 149

посетитель, 128, 135, 138, 142, 143



Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planet.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Прасид Пай, Питер Абрахам

Реактивное программирование на C++

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Винник В. Ю.*

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «PT Serif». Печать офсетная.

Усл. печ. л. 26,33. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**