THE EXPERT'S VOICE® IN C

C Quick Syntax Reference

Mikael Olsson

Apress®

www.it-ebooks.info

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

www.it-ebooks.info

Contents at a Glance

About the Author xi

About the Technical Reviewer xiii

Introduction xv

■ Chapter 1: Hello World 1

■ Chapter 2: Compile and Run 3

■ Chapter 3: Variables 5

■ Chapter 4: Operators 15

■ Chapter 5: Pointers 19

■ Chapter 6: Arrays 21

■ Chapter 7: String 23

■ Chapter 8: Conditionals 27

■ Chapter 9: Loops 29

■ Chapter 10: Functions 31

■ Chapter 11: Typedef 37

■ Chapter 12: Enum 39

■ Chapter 13: Struct 43

■ Chapter 14: Union 47

■ Chapter 15: Type Conversions 49

■ Chapter 16: Storage Classes 51

www.it-ebooks.info

Chapter 17: Constants 55

Chapter 18: Preprocessor 57

Chapter 19: Memory Management 63

Chapter 20: Command Line Arguments 67

Chapter 21: Headers 69

Index 73

www.it-ebooks.info

Introduction

The C programming language is a general-purpose, middle-level language originally developed by Dennis M. Ritchie at Bell Labs. It was created over the period 1969 through 1973 for the development of the UNIX operating system, which had previously been written in assembly language. The name C was chosen because many of its features derived from an earlier language called B. Whereas the B language is no longer in common use, C became and still remains one of the most popular and influential programming languages in use today.

Although C is a general-purpose language it is most often used for systems programming. This includes software that controls the computer hardware directly, such as drivers, operating systems, and software for embedded microprocessors. C can also be used for writing applications, which run on top of system software. However, it has largely been superseded in that domain by higher-level languages, such as C++, Objective-C, C#, Swift, and Java. The features of these and many other languages are heavily influenced by C, as can be seen in some of their names.

The development of C was a major milestone in computer science as it was the first widely successful middle-level language for system development. The foremost reasons for its success were that the language was concise, fast, and powerful. It offered comparable speed to assembly with far improved usability. The high-level constructs of the language allowed programmers to focus mainly on the software's design, while its low-level capabilities still provided direct access to the hardware when needed, as assembly had done. Furthermore, the language is relatively simple to understand with few keywords and what many consider to be an elegant syntax.

Another major reason for the success of C was its portability. Unlike assembly the C language is platform independent. A standards-compliant C program can therefore be compiled for a wide variety of computer systems with few changes to its source code. Moreover, the C compiler was small and easy to port to different CPU architectures, which together with the language's popularity has made C compilers available on most computer systems.

C versions

In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as K&R C. This description was succeeded in 1989 when the American National Standards Institute (ANSI) provided a comprehensive definition of C known as ANSI C or C89. In the following year the same specification was adopted as an international standard by the International Organization for Standardization and became known as ISO C90 or just C90. C has since undergone three more revisions by ISO (successively adopted by ANSI) with further language extensions, including C95, C99, and most recently C11, which is the latest ANSI standard for the C programming language.

www.it-ebooks.info

CHAPTER 1

Hello World

To begin programming in C you need a text editor and a C compiler. You can get both at the same time by installing an Integrated Development Environment (IDE) that includes support for C. A good choice is Microsoft's Visual Studio Community Edition, which is a free version of Visual Studio that is available from Microsoft's website.[1] This IDE has built-in support for the C89 standard and also includes many features of C99 as of the 2013 version.

Some other popular cross-platform IDEs include Eclipse CDT, Code::Blocks, and CodeLite. Alternatively, you can develop using a simple text editor – such as Notepad – although this is less convenient than using an IDE. If you choose to do so, just create an empty document with a .c file extension and open it in the editor of your choice. By convention, the .c extension is used for files that contain source code for C programs.

Creating a Project

After installing Visual Studio, go ahead and launch the program. You then need to create a project, which will manage the C source files and other resources. Go to File ➤ New ➤ Project to display the New Project window. From there select the Visual C++ template type in the left frame. Then select the Win32 Console Application template in the right frame. At the bottom of the window you can configure the name and location of the project. When you are finished, click the OK button, and another dialog box will appear titled Win32 Application Wizard. Click next, and a couple of application settings will be displayed. Leave the application type as Console application and check the Empty project checkbox. Then click Finish to let the wizard create your empty project.

Adding a Source File

You have now created a C/C++ project. In the Solution Explorer panel (View ➤ Solution Explorer) you can see that the project consists of three empty folders: Header Files, Resource Files, and Source Files. Right-click on the Source Files folder and select Add ➤ New Item. From the Add New Item dialog box choose the C++ File (.cpp) template. Give this source file the name "myapp.c." The .c file extension will make the file compile in C instead of C++. Click the Add button, and the empty C file will be added to your project and opened for you.

[1] http://www.visualstudio.com.

www.it-ebooks.info

Hello World

The first thing to add to the source file is the `main` function. This is the entry point of the program, and the code inside of the curly brackets is what will be executed when the program runs. The brackets, along with their content, is referred to as a code block, or just a block.

```
int main(void) {}
```

Your first application will simply output the text "Hello World" to the screen. Before this can be done the stdio.h header needs to be included. This header provides input and output functionality for the program, and is one of the standard libraries that come with all C/C++ compilers. What the `#include` directive does is to effectively replace the line with everything in the specified header before the file is compiled.

```
#include <stdio.h>
int main(void) {}
```

With stdio.h included you gain access to several new functions, including the `printf` function that is used for printing text – in this case to a console window. To call this function you type its name followed by a set of parentheses that includes the text string that will be displayed. The string is delimited by double quotes, and the whole statement is followed by a semicolon. The semicolon is used in C to mark the end of a code statement.

```
#include <stdio.h>

int main(void) {
  printf("Hello World");
  return 0;
}
```

The `main` function here ends with a `return` statement, which returns a status code as the program exits. This can be useful if the intent is for your program to be executed by another program. The status code can then signal to the caller the success or failure of your program to complete its function. By convention, the return code zero is used to indicate that a program or function has executed successfully.

The C89 standard requires the `return` statement to be present, but following C90 the statement became optional. As of C90 the compiler will automatically include the `return` statement if it is omitted. For brevity the statement will be left out from future code examples.

IntelliSense

When writing code in Visual Studio a window called IntelliSense will pop up wherever there are multiple predetermined alternatives from which to choose. This window can be also brought up manually at any time by pressing Ctrl+Space to provide quick access to any code entities you are able to use within your program. This is a very powerful feature that you should learn to make good use of.

www.it-ebooks.info

CHAPTER 2

Compile and Run

Before a program can be run, the source code has to be translated into an executable format by a compiler. This step transforms the human-readable source code into binary machine code, which is a sequence of instructions that can be executed by a computer.

Visual Studio Compilation

Continuing from the last chapter, the Hello World program is now complete and ready to be compiled and run. You can do this by going to the Build menu and clicking on Debug ➤ Start Without Debugging (Ctrl+F5). Visual Studio then compiles and runs the application that displays the text in a console window.

If you select Start Debugging (F5) from the Debug menu instead, the console window displaying Hello World will close as soon as the `main` function is finished. To prevent this, you can add a call to the `getchar` function at the end of `main`. This function, included with the stdio.h header, will read a character from the keyboard and thereby prevent the program from exiting until the return key is pressed.

```
#include <stdio.h>

int main(void) {
  printf("Hello World");
  getchar();
}
```

Console Compilation

As an alternative to using an IDE you can also compile source files from a terminal window as long as you have a C compiler. For example, on a Linux machine you can use the GNU C compiler, which is available on virtually all Unix systems, including Linux and the BSD family, as part of the GNU Compiler Collection (GCC). This compiler can also be installed on Windows by downloading MinGW[1] or on Mac as part of the Xcode development environment.[2]

[1] http://www.mingw.org.
[2] https://developer.apple.com/xcode/.

www.it-ebooks.info

To use the GNU compiler you type its name "gcc" in a terminal window and give it the input and output filenames as arguments. It then produces an executable file, which when run gives the same result as one compiled under Windows in Visual Studio.

```
gcc myapp.c -o myapp.exe
./myapp.exe
Hello World
```

Comments

Comments are used to insert notes into the source code. They have no effect on the end program and are meant only to enhance the readability of the code, both for you and for other developers. The C89 standard featured only one comment notation, a multiline comment delimited by `/*` and `*/`.

```
/* multi-line
   comment */
```

The C99 standard added the single-line comment, which starts with `//` and extends to the end of the line. This comment was standardized since it was a convenient feature found in many other programming languages, such as C++. Many C compilers also started to support the single-line comment long before the C99 standard was formalized.

```
// single-line comment
```

Keep in mind that whitespace characters – such as comments, spaces, and tabs – are generally ignored by the compiler. This allows you a lot of freedom in how to format your code.

www.it-ebooks.info

CHAPTER 3

Variables

Variables are used for storing data during program execution.

Data Types

Depending on what data you need to store, there are several different kinds of data types. The simple types in C consist of four integer types: three floating-point types as well as the `char` type.

Data Type	Size (byte)	Description
`char`	1	Integer or character
`short`	2	Integer
`int`	4	
`long`	4 or 8	
`long long`	8	
`float`	4	Floating-point number
`double`	8	
`long double`	8 or 16	

In C, the exact size of the data types is not fixed. The sizes shown in the previous table are those commonly found on 32-bit and 64-bit systems. The C standard only specifies the minimum range that is guaranteed to be supported. The minimum size for `char` is 8 bits, for `short` and `int` it is 16 bits, for `long` it is 32 bits and `long long` must contain at least 64 bits. Most modern compilers make `int` 32 bits, which nearly universally means 4 bytes. Each integer type in the table must also be at least as large as the one preceding it. The same applies to the floating-point types where each one must provide at least as much precision as the preceding one.

www.it-ebooks.info

Declaring Variables

Before a variable can be used, it first has to be declared. To declare a variable you start with the data type you want the variable to hold followed by an identifier, which is the name of the variable.

```
int myInt;
```

The identifier can consist of letters, numbers, and underscores, but it cannot start with a number. It also cannot contain spaces or special characters and must not be a reserved keyword.

```
int _myInt32; /* allowed */
int 32Int;    /* incorrect (starts with number) */
int my Int;   /* incorrect (contains space) */
int Int@32;   /* incorrect (contains special character) */
int int;      /* incorrect (reserved keyword) */
```

Note that C is a case sensitive programming language, so uppercase and lowercase letters have different meanings.

Assigning Variables

To assign a value to a declared variable the equal sign is used, which is known as the assignment operator (=). This is called assigning or initializing the variable.

```
myInt = 50;
```

The declaration and assignment can be combined into a single statement. When a variable is assigned a value it then becomes *defined*.

```
int myInt = 50;
```

If you need to create more than one variable of the same type there is a shorthand way of doing it using the comma operator (,).

```
int x = 1, y = 2, z;
```

Once a variable has been defined (declared and assigned), you can use it by simply referencing the variable's name: for example, to copy the value to another variable.

```
int a = x;
```


www.it-ebooks.info

Printing Variables

In addition to strings the `printf` function can be used to print values and variables to the standard output stream. This is done by embedding format specifiers into the string where the value is to be printed. Each specifier must be matched by a corresponding argument to `printf` of the correct type, as seen in the following example.

```
#include <stdio.h>

int main() {
  int x = 5;
  printf("x is %d and 2+3 is %d", x, 2+3);
}
```

The `%d` specifier displays an integer of the `char`, `short` or `int` type. Other commonly used format specifiers are seen in the following table.

Specifier	Output
`%d` or `%i`	`char`, `short`, or `int`
`%c`	character
`%s`	string of characters
`%f`	`float` or `double`
`%Lf`	`long double`
`%ld`	`long int`
`%lld`	`long long int`
`%u`	unsigned `char`, `short` or `int`
`%lu`	unsigned `long int`
`%llu`	unsigned `long long int`
%p	pointer address

For more information on `printf` and other standard library functions, you can visit the C library reference on cplusplus.com.[1]

[1] http://www.cplusplus.com/reference/clibrary/.

www.it-ebooks.info

Integer Types

There are four native integer (whole number) types you can use depending on how large a number you need the variable to hold. Typical ranges on 32-bit and 64-bit systems are given below.

```
char  myChar  = 0; /* -128   to +127 */
short myShort = 0; /* -32768 to +32767 */
int   myInt   = 0; /* -2^31  to +2^31-1 */
long  myLong  = 0; /* -2^31  to +2^31-1 */
```

C99 added support for the `long long` data type, which is guaranteed to be at least 64 bits in size.

```
long long myLL = 0; /* -2^63 to +2^63-1 */
```

To determine the exact size of a data type you can use the `sizeof` operator. This operator returns the number of bytes that a type occupies in the system you are compiling for. The type returned is `size_t`, which is an alias for an integer type. The specifier `%zu` was introduced in C99 as a portable way to format this type with `printf`. Visual Studio does not support this specifier and uses `%Iu` instead.

```
#include <stdio.h>

int main(void) {
  size_t s = sizeof(int);
  printf("%zu", s); /* "4" (C99) */
  printf("%Iu", s); /* "4" (Visual Studio) */
}
```

In addition to standard decimal notation, integers can also be assigned by using octal or hexadecimal notation. The following values all represent the same number, which in decimal notation is 50.

```
int myDec = 50    /* decimal notation */
int myOct = 062;  /* octal notation (0) */
int myHex = 0x32; /* hexadecimal notation (0x) */
```

Signed and Unsigned

By default, all integer types in C are signed and may therefore contain both positive and negative values. This can be explicitly declared using the `signed` keyword.

```
signed char  myChar;   /* -128   to +127 */
signed short myShort;  /* -32768 to +32767 */
signed int   myInt;    /* -2^31  to +2^31-1 */
signed long  myLong;   /* -2^31  to +2^31-1 */
signed long long myLL; /* -2^63  to +2^63-1 */
```


www.it-ebooks.info

If only positive values need to be stored the integer types can be declared as unsigned to double their upper range.

```
unsigned char  uChar;   /* 0 to 255 */
unsigned short uShort;  /* 0 to 65535 */
unsigned int   uInt;    /* 0 to 2^32-1 */
unsigned long  uLong;   /* 0 to 2^32-1 */
unsigned long long uLL; /* 0 to 2^64-1 */
```

When an unsigned value is printed the specifier %u is used for the unsigned char, short, and int types. The unsigned long type is specified with %lu and unsigned long long with %llu.

```
unsigned int uInt = 0;
printf("%u", uInt); /* "0" */
```

The signed and unsigned keywords may be used as types on their own, in which case the int type is assumed by the compiler.

```
unsigned uInt; /* unsigned int */
signed sInt;   /* signed int */
```

In the same way, the short and long data types are abbreviations of short int and long int.

```
short myShort; /* short int */
long myLong;   /* long int */
```

Sized Integers

As mentioned before, the actual sizes of the integer types are implementation dependent. For more precise specification of size the C99 standard introduced a number of exact-width integer types. They can be enabled by including the stdint.h standard header.

```
#include <stdint.h>

/* Signed exact-width integers */
int8_t  iSmall;  /* 8 bits */
int16_t iMedium; /* 16 bits */
int32_t iLarge;  /* 32 bits */
int64_t iHuge;   /* 64 bits */
```


www.it-ebooks.info

Unsigned versions of these types are available as well. Like the signed versions, these exact-width integer types are guaranteed to have the same number of bits across all implementations.

```
/* Unsigned exact-width integers */
uint8_t  uSmall;  /* 8 bits */
uint16_t uMedium; /* 16 bits */
uint32_t uLarge;  /* 32 bits */
uint64_t uHuge;   /* 64 bits */
```

It is recommended to use sized integers when available, to more easily keep track of the range of your integer variables and to enhance the portability of your programs. Compilers that comply with standards prior to C99 may provide sized integers with different type names. Visual Studio, for example, has built-in support for the following signed exact-width integers.

```
/* Visual Studio signed exact-width integers */
__int8  iSmall;  /* 8 bits */
__int16 iMedium; /* 16 bits */
__int32 iLarge;  /* 32 bits */
__int64 iHuge;   /* 64 bits */
```

Floating-Point Types

The floating-point types can store real numbers with different levels of precision.

```
float myFloat;    /* ~7 digits */
double myDouble;  /* ~15 digits */
long double myLD; /* typically same as double */
```

The precision shown above refers to the total number of digits. A `float` can accurately represent about 7 digits, whereas a `double` can handle around 15 of them.

```
float myFloat = 12345.678;
printf("%f", myFloat); /* "12345.677734" */
```

When printing a floating-point number you can limit the decimal places to, for instance, two in the following way.

```
printf("%.2f", myFloat); /* "12345.68" */
```


www.it-ebooks.info

Floating-point numbers can be expressed using decimal, exponential, or hexadecimal notation. Exponential (scientific) notation is used by adding E or e followed by the decimal exponent, while the hexadecimal floating-point notation uses P or p to specify the binary exponent. Support for the hexadecimal notation was not standardized until C99.

```
double fDec = 1.23;
double fExp = 3e2;   /* 3*10^2 = 300 */
double fHex = 0xAp2; /* 10*2^2 = 40 */
```

Literal Suffixes

An integer literal (constant) is normally treated as an `int` by the compiler, or a larger type if needed to fit the value. Suffixes can be added to the literal to change this evaluation. With integers the suffix can be a combination of U and L, for unsigned and `long` respectively. C99 also added the LL suffix for the `long long` type. The order and casing of these letters do not matter.

```
int i = 10;
long l = 10L;
unsigned long ul = 10UL;
```

A floating-point literal is treated as a `double`. The `F` or `f` suffix can be used to specify that a literal is of the `float` type instead. Likewise, the `L` or `l` suffix specifies the `long double` type.

```
float f = 1.23F;
double d = 1.23;
long double ld = 1.23L;
```

The compiler implicitly converts literals to whichever type is necessary, so this type distinction for literals is usually not necessary. If the F suffix is left out when assigning to a `float` variable the compiler may give a warning since the conversion from `double` to `float` involves a loss of precision.

Char Type

The `char` type is commonly used to represent ASCII characters. Such character constants are enclosed in single quotes and can be stored in a variable of `char` type.

```
char c = 'x'; /* assigns 120 (ASCII for x) */
```


www.it-ebooks.info

When the `char` is printed with the `%c` format specifier the ASCII character is displayed.

```
printf("%c", c); /* "x" */
```

Use the `%d` specifier to instead display the numerical value.

```
printf("%d", c); /* "120" */
```

Bool Type

C99 introduced a `_Bool` type to increase compatibility with C++. Variables of this type can store a Boolean value, which is a value that can only be either 1 (true) or 0 (false).

```
_Bool b = 0; /* false value */
```

The type `_Bool` is usually accessed via its alias name `bool` defined by the standard header stdbool.h. This header also defines the macros `true` and `false` as aliases for 1 and 0.

```
#include <stdbool.h>

bool b = true; /* true value */
```

Variable Scope

The scope of a variable refers to the region of code within which it is possible to use that variable. Variables in C may be declared both globally and locally. A global variable is declared outside of any code blocks and is accessible from anywhere after it has been declared. A local variable, on the other hand, is declared inside of a function and will only be accessible within that function after it has been declared. The lifetime of a local variable is also limited. A global variable will remain allocated for the duration of the program, while a local variable will be destroyed when its function has finished executing.

```
int globalVar; /* global variable */

int main(void) {
  int localVar; /* local variable */
}
```


www.it-ebooks.info

The default values for these variables are also different. Global variables are automatically initialized to zero by the compiler, whereas local variables are not initialized at all. Uninitialized local variables will therefore contain whatever garbage is already present in that memory location.

```
int globalVar; /* initialized to 0 */

int main(void) {
  int localVar; /* uninitialized */
}
```

Using uninitialized variables is a common programming mistake that can produce unexpected results. It is therefore a good idea to always give your local variables an initial value when they are declared.

```
int main(void) {
  int localVar = 0; /* initialized to 0 */
}
```

In C89, local variables must be declared before any other statements within their scope. The later C99 standard changed this to allow variables to be declared anywhere within a function's scope, which can be more intuitive.

```
int main(void) {
  int var1;
  /* Other statements */
  int var2; /* C99 only */
}
```


www.it-ebooks.info

CHAPTER 4

Operators

A numerical operator is a symbol that makes the program perform a specific mathematical or logical manipulation. The numerical operators in C can be grouped into five types: arithmetic, assignment, comparison, logical, and bitwise operators.

Arithmetic Operators

There are four basic arithmetic operators, as well as the modulus operator (%), which is used to obtain the division remainder.

```
float x = 3 + 2; /* 5 - addition */
      x = 3 - 2; /* 1 - subtraction */
      x = 3 * 2; /* 6 - multiplication */
      x = 3 / 2; /* 1 - division */
      x = 3 % 2; /* 1 - modulus (division remainder) */
```

Notice that the division sign gives an incorrect result. This is because it operates on two integer values and will therefore truncate the result and return an integer. To get the correct value, one of the numbers must be explicitly converted to a floating-point number.

```
x = 3 / (float)2; /* 1.5 */
```

Assignment Operators

The second group is the assignment operators. Most important, it is the assignment operator (=) itself, which assigns a value to a variable.

Combined Assignment Operators

A common use of the assignment and arithmetic operators is to operate on a variable and then to save the result back into that same variable. These operations can be shortened with the combined assignment operators.

www.it-ebooks.info

```
int x = 0;
    x += 5; /* x = x+5; */
    x -= 5; /* x = x-5; */
    x *= 5; /* x = x*5; */
    x /= 5; /* x = x/5; */
    x %= 5; /* x = x%5; */
```

Increment and Decrement Operators

Another common operation is to increment or decrement a variable by one. This can be simplified with the increment (++) and decrement (--) operators.

```
x++; /* x = x+1; */
x--; /* x = x-1; */
```

Both of these can be used either before or after a variable.

```
x++; /* post-increment */
x--; /* post-decrement */
++x; /* pre-increment */
--x; /* pre-decrement */
```

The result on the variable is the same whichever is used. The difference is that the post-operator returns the original value before it changes the variable, while the pre-operator changes the variable first and then returns the value.

```
int x, y;
x = 5; y = x++; /* y=5, x=6 */
x = 5; y = ++x; /* y=6, x=6 */
```

Comparison Operators

The comparison operators compare two values and return either true or false, represented as 1 or 0. They are mainly used to specify conditions, which are expressions that evaluate to either true or false.

```
int x = (2 == 3); /* 0 - equal to */
    x = (2 != 3); /* 1 - not equal to */
    x = (2 > 3);  /* 0 - greater than */
    x = (2 < 3);  /* 1 - less than */
    x = (2 >= 3); /* 0 - greater than or equal to */
    x = (2 <= 3); /* 1 - less than or equal to */
```


www.it-ebooks.info

Logical Operators

The logical operators are often used together with the comparison operators. Logical and (&&) evaluates to true if both the left and right sides are true, and logical or (||) is true if either the left or right side is true. For inverting a Boolean result there is the logical not (!) operator. Note that for both "logical and" and "logical or" the right-hand side will not be evaluated if the result is already determined by the left-hand side.

```
int x = (1 && 0); /* 0 - logical and */
    x = (1 || 0); /* 1 - logical or  */
    x = !(1);     /* 0 - logical not */
```

Recall that as of C99 the stdbool.h header can be included to make use of the `bool` type to store Boolean values. The header also defines the constants `true` and `false` to represent 1 and 0, which allows the previous example to be rewritten as seen here.

```
#include <stdbool.h>
/* ... */
bool x = (true && false); /* false - logical and */
     x = (true || false); /* true  - logical or  */
     x = !(true);         /* false - logical not */
```

Bitwise Operators

The bitwise operators can manipulate individual bits inside an integer. For example, the bitwise left shift operator (<<) moves all bits to the left with the specified number of steps.

```
int x = 5 & 4;  /* 101 & 100 =  100 (4) - and */
    x = 5 | 4;  /* 101 | 100 =  101 (5) - or */
    x = 5 ^ 4;  /* 101 ^ 100 =  001 (1) - xor */
    x = 4 << 1; /* 100 << 1  = 1000 (8) - left shift */
    x = 4 >> 1; /* 100 >> 1  =   10 (2) - right shift */
    x = ~4;     /* ~00000100 = 11111011 (-5) - invert */
```

The bitwise operators also have combined assignment operators.

```
int x = 5; x & = 4; /* 101 & 100 =  100 (4) - and */
    x = 5; x | = 4; /* 101 | 100 =  101 (5) - or */
    x = 5; x ^ = 4; /* 101 ^ 100 =  001 (1) - xor */
    x = 4; x <<= 1; /* 100 << 1  = 1000 (8) - left shift */
    x = 4; x >>= 1; /* 100 >> 1  =   10 (2) - right shift */
```


www.it-ebooks.info

Operator Precedence

In C, expressions are normally evaluated from left to right. However, when an expression contains multiple operators, the precedence of those operators decides the order in which they are evaluated. The order of precedence can be seen in the following table, where the operator with the lowest precedence will be evaluated first. This same order also applies to many other languages, such as C++ and C#.

Pre	Operator	Pre	Operator
1	`() [] . -> x++ x--`	8	`&`
2	`! ~ ++x --x (type) sizeof * &`	9	`^`
3	`* / %`	10	`|`
4	`+ -`	11	`&&`
5	`<< >>`	12	`||`
6	`< <= > >=`	13	`= op=`
7	`== !=`	14	`,`

To give an example, multiplication binds harder than addition and will therefore be evaluated first in the following line of code.

```
int x = 4 + 3 * 2; /* 10 */
```

This can be clarified by enclosing the part of the expression that will be evaluated first in parentheses. As seen in the table, parentheses have the highest precedence of all operators.

```
int x = 4 + (3 * 2); /* 10 */
```


www.it-ebooks.info

CHAPTER 5

Pointers

A pointer is a variable that contains the memory address of another variable, called the pointee.

Creating Pointers

Pointers are declared as any other variable, except that an asterisk (*) is placed between the data type and the pointer's name. The data type used determines what type of memory it will point to.

```
int* p; /* pointer to an integer */
int *q; /* alternative syntax */
```

A pointer can point to a variable of the same type by prefixing that variable with an ampersand, in order to retrieve its address and assign it to the pointer. The ampersand is known as the address-of operator (&).

```
int i = 10;
p = &i; /* address of i assigned to p */
```

Dereferencing Pointers

The pointer now contains the memory address to the integer variable. Referencing the pointer will retrieve this address. To obtain the actual value stored in that address, the pointer must be prefixed with an asterisk, known as the dereference operator (*).

```
printf("Address of i: %p \n", p); /* ex. 0017FF1C */
printf("Value of i: %d",  *p); /* 10 */
```

When writing to the pointer the same method is used. Without the asterisk the pointer is assigned a new memory address, and with the asterisk the actual value of the variable pointed to will be updated.

```
p = &i;  /* address of i assigned to p */
*p = 20; /* value of i changed through p */
```


www.it-ebooks.info

If a second pointer is created and assigned the value of the first pointer, it will then get a copy of the first pointer's memory address.

```
int* p2 = p; /* copy address stored in p */
```

Pointing to a Pointer

Sometimes it can be useful to have a pointer that can point to another pointer. This is done by declaring a pointer with two asterisks and then assigning it the address of the pointer that it will reference. This way, when the address stored in the first pointer changes, the second pointer can follow that change.

```
int** r = &p; /* pointer to pointer */
```

Referencing the second pointer now gives the address of the first pointer. Dereferencing the second pointer gives the address of the variable, and dereferencing it again gives the value of the variable.

```
printf("Address of p: %p \n",  r); /* ex. 0017FF28 */
printf("Address of i: %p \n", *r); /* ex. 0017FF1C */
printf("Value of i: %d",  **r); /* 20 */
```

Null Pointer

A pointer should be set to zero when it is not assigned to a valid address. Such a pointer is called a *null pointer* . Doing this allows you to check whether the pointer can be safely dereferenced, because a valid pointer will never be zero. This check is necessary for places in the code where you may not know whether a pointer is valid or not.

```
if (p != 0) { *p = 10; } /* check for null pointer */
```

The constant `NULL` can also be used to signify a null pointer. `NULL` is typically defined as zero in C, making the choice of which to use a matter of preference. The constant is defined by several standard library files, including stdio.h and stddef.h.

```
#include <stdio.h>
/* ... */
if (p != NULL) { *p = 10; }
```


www.it-ebooks.info

CHAPTER 6

Arrays

An array is a data structure used for storing a collection of values that all have the same data type.

Array Declaration and Allocation

To declare an array you start as you would a normal variable declaration, but in addition, append a set of square brackets following the array's name. The brackets contain the number of elements in the array. The default values for these elements are the same as for variables – elements in global arrays are initialized to their default values, and elements in local arrays remain uninitialized.

```
int myArray[3]; /* integer array with 3 elements */
```

Array Assignment

To assign values to the elements, you can reference them one at a time by placing the element's index inside the square brackets, starting with zero.

```
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
```

Alternatively, you can assign values at the same time as the array is declared by enclosing them in curly brackets. The specified array length may optionally be left out to let the array size be decided by the number of values assigned.

```
int myArray[3] = { 1, 2, 3 };
int myArray[] = { 1, 2, 3 }; /* alternative */
```

Once the array elements are initialized, they can be accessed by referencing the index of the element you want.

```
printf("%d", myArray[0]); /* 1 */
```


www.it-ebooks.info

Multi-Dimensional Arrays

Arrays can be made multi-dimensional by adding more sets of square brackets. As with single-dimensional arrays, they can either be filled in one at a time or all at once during the declaration.

```
int mArray[2][2] = { { 0, 1 }, { 2, 3 } };
mArray[0][0] = 0;
mArray[0][1] = 1;
```

The extra curly brackets are optional, but including them is good practice since it makes the code easier to understand.

```
int mArray[2][2] = { 0, 1, 2, 3 }; /* alternative */
```

Arrays and Pointers

Any array in C is actually a constant pointer to the first element in the array. Therefore, the referencing of array elements can be made just as well with pointer arithmetic. By incrementing the pointer by one, you move to the next element in the array, because changes to a pointer's address are implicitly multiplied by the size of the pointer's data type.

```
*(myArray+1) = 10; /* myArray[1] = 10; */
```

Pointer arithmetic is an advanced feature that should be used with care. The four arithmetic operators that can be used with pointers include: `+`, `-`, `++`, and `--`.

```
int* ptr = &myArray;
printf("Address of myArray[0]: %p \n", ptr); /* ex. 0028FF14 */
ptr++;
printf("Address of myArray[1]: %p", ptr); /* ex. 0028FF18 */
```

Array Size

Just as with any other pointer it is possible to exceed the valid range of an array and thereby rewrite some adjacent memory. This should always be avoided since it can lead to unexpected results or crash the program.

```
int myArray[2] = { 1, 2 };
myArray[2] = 3; /* out of bounds */
```

To determine the length of a regular (statically allocated) array, the `sizeof` operator can be used.

```
int length = sizeof(myArray) / sizeof(int); /* 2 */
```


www.it-ebooks.info

CHAPTER 7

String

A string consists of an array of characters and is delimited by double quotes. There is no string type for storing strings in C. Instead, strings are commonly assigned to a character array as shown here.

```
char myString[] = "Hi";
```

Strings in C are terminated with a null character `\0`, which is used to know where the string ends. The null character is added automatically by the compiler for quoted strings, as in the previous example. The same statement can also be written using regular array initialization syntax, in which case the null character needs to be explicitly included.

```
char myString[3] = { 'H', 'i', '\0' };
```

Both statements produce the same result: a char array with three elements. Note that individual characters are delimited by single quotes and not double quotes. To print a string the format specifier `%s` is used with the `printf` function, which outputs the literals of the string until the null character is encountered.

```
printf("%s", myString); /* "Hi" */
```

As an alternative to the character array, a char pointer may be set to point to a string. The string is then automatically stored in the compiled file, giving the pointer a location to point to. In most compilers this location is a read-only block, so unlike the char array the characters in this string cannot be changed.

```
char* ptr = "Hi";
printf("%s", ptr); /* "Hi" */
```

Escape Characters

To add new lines into a string the escape character `\n` is used for representing a line break.

```
printf("First line\nSecond line");
```


www.it-ebooks.info

This backslash notation is used to write special characters that are difficult or impossible to type on a regular keyboard. In addition to the new line and null characters, there are several other such characters as seen in the following table.

Character	Meaning	Character	Meaning
`\n`	newline	`\f`	form feed
`\t`	horizontal tab	`\a`	alert sound
`\v`	vertical tab	`\'`	single quote
`\b`	backspace	`\"`	double quote
`\r`	carriage return	`\\`	backslash
`\0`	null character	`\?`	question mark
`\000`	octal number (1-3 digits)	`\xhh`	hexadecimal number

Any one of the 128 ASCII characters can be expressed by writing a backslash followed by the ASCII code for that character, represented as either an octal or hexadecimal number. This is illustrated below where the new line character is represented in three different ways.

```
char line = '\n';   /* escape code */
     line = '\012'; /* octal notation */
     line = '\x0A'; /* hexadecimal notation */
```

String Functions

Because strings in C are arrays, the only way to make changes to them is to change each element in the array. To simplify common string operations the standard header string.h includes a collection of functions for manipulating null terminated strings. Consider the code below that will be used as the template for the following string examples.

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char s1[12] = "Hello";
  char s2[12] = "World";
  int result;
}
```


www.it-ebooks.info

The string function `strcat` (string concatenation) appends the second string onto the first string. For this to work, it is important that the destination is large enough to hold the entire string.

```
/* Append s2 to s1 */
strcat(s1, s2); /* s1 = "HelloWorld" */
```

Another string function is `strcpy`, which copies the characters in the second string into the first string. The function stops when the terminating null character for the second argument is reached.

```
/* Copy s1 into s2 */
strcpy(s2, s1); /* s2 = "HelloWorld" */
```

A string can be compared with another string using the `strcmp` function. If all characters match, the function returns zero.

```
/* Compare s1 and s2 */
result = strcmp(s1, s2); /* 0 (equal) */
```

When manipulating strings it is important to take their lengths into account to avoid overwriting any adjacent memory. The length of a string stored in a char array can be found with the `strlen` function. With regular (statically allocated) strings the allocated size can also be retrieved using the `sizeof` operator.

```
/* Length of s1 (excluding null char) */
result = strlen(s1); /* 10 */

/* Allocated size for s1 */
result = sizeof(s1); /* 12 */
```


www.it-ebooks.info

CHAPTER 8

Conditionals

Conditional statements are used to execute different code blocks based on different conditions.

If Statement

The `if` statement will only execute if the expression inside the parentheses is evaluated to true. In C, this does not have to be a Boolean expression. It can be any expression that evaluates to a number, in which case zero is `false` and all other numbers are `true`.

```
if (x < 1) {
  printf("x < 1");
}
```

To test for other conditions, the `if` statement can be extended by any number of `else/if` clauses.

```
else if (x > 1) {
  printf("x > 1");
}
```

The `if` statement can have one `else` clause at the end, which will execute if all previous conditions are false.

```
else {
  printf("x == 1");
}
```


www.it-ebooks.info

As for the curly brackets, they can be left out if only a single statement needs to be executed conditionally. However, it is considered good practice to always include them since they improve readability.

```
if (x < 1)
  printf("x < 1");
else if (x > 1)
  printf("x > 1");
else
  printf("x == 1");
```

Switch Statement

The `switch` statement checks for equality between an integer and a series of `case` labels, and then passes execution to the matching case. It may contain any number of `case` clauses, and it can end with a default label for handling all other cases.

```
switch (x) {
  case 0: printf("x is 0"); break;
  case 1: printf("x is 1"); break;
  default: printf("x is not 0 or 1"); break;
}
```

Note that the statements after each `case` label end with the `break` keyword to skip the rest of the `switch`. If the `break` is left out, execution will fall through to the next case, which can be useful if several cases need to be evaluated in the same way.

Ternary Operator

In addition to the `if` and `switch` statements there is the ternary operator (`?:`) that can replace a single `if/else` clause. This operator takes three expressions. If the first one is true then the second expression is evaluated and returned; and if it is false, the third one is evaluated and returned.

```
x = (x < 0.5) ? 0 : 1; /* ternary operator (?:) */
```


www.it-ebooks.info

CHAPTER 9

Loops

There are three looping structures in C, all of which are used to execute a specific code block multiple times. Just as with the conditional `if` statement, the curly brackets for the loops can be left out if there is only one statement in the code block.

While Loop

The `while` loop runs through the code block only if its condition is true, and will continue looping for as long as the condition remains true. Bear in mind that the condition is only checked at the start of each iteration.

```
int i = 0;
while (i < 10) {
  printf("%d", i++); /* 0-9 */
}
```

Do-While Loop

The `do-while` loop works in the same way as the `while` loop, except that it checks the condition after the code block. It will therefore always run through the code block at least once. Notice that this loop ends with a semicolon.

```
int j = 0;
do {
  printf("%d", j++); /* 0-9 */
} while (j < 10);
```

For Loop

The `for` loop is used to run through a code block a specific number of times. It uses three parameters. The first one initializes a counter and is always executed once before the loop. The second parameter holds the condition for the loop and is checked before each iteration. The third parameter contains the increment of the counter and is executed at the end of each iteration.

www.it-ebooks.info

```
int k;
for (k = 0; k < 10; k++) {
  printf("%d", k); /* 0-9 */
}
```

Since the C99 standard the first parameter may contain a declaration, typically a counter variable. The scope of this variable is limited to the `for` loop.

```
for (int k = 0; k < 10; k++) {
  printf("%d", k); /* 0-9 */
}
```

The `for` loop has several variations. One such variation is to split the first and third parameters into several statements by using the comma operator.

```
int k, m;
for (k = 0, m = 0; k < 10; k++, m--) {
  printf("%d", k+m); /* 000... (10x) */
}
```

Another option is to leave out any one of the parameters. If all parameters are left out, it becomes a never-ending loop, unless there is another exit condition defined.

```
for (;;) { /* infinite loop */ }
```

Break and Continue

There are two jump statements that can be used inside loops: `break` and `continue`. The `break` keyword ends the loop structure, and `continue` skips the rest of the current iteration and continues at the beginning of the next iteration.

```
int i;
for (i = 0; i < 10; i++)
{
  if (i == 2) continue;   /* start next iteration */
  else if (i == 5) break; /* end loop */
  printf("%d", i);        /* "0134" */
}
```

Goto Statement

A third jump statement that may be useful to know of is `goto`, which performs an unconditional jump to a specified label within the same function. This instruction is generally never used since it tends to make the flow of execution difficult to follow.

```
goto myLabel; /* jump to label */
/* ... */
myLabel:      /* label declaration */
```


www.it-ebooks.info

CHAPTER 10

Functions

Functions are reusable code blocks that will only execute when called. They allow developers to divide their programs into smaller parts that are easier to understand and reuse.

Defining Functions

A function can be created by typing `void` followed by the function's name, a set of parentheses containing another `void`, and a code block. The first use of the `void` keyword specifies that this function will not return a value. The second `void` inside the parentheses means that the function does not accept any arguments.

```
void myFunction(void) {
  printf("Hello World");
}
```

Calling Functions

The previous function will print out a text message when it is called. To invoke it, the function's name is specified followed by an empty set of parentheses.

```
int main(void) {
  myFunction(); /* "Hello World" */
}
```

Function Parameters

The parentheses that follow the function's name are used for passing arguments to the function. To do this the corresponding parameters must first be added to the function's parameter list.

```
void sum(int a, int b) {
  int sum = a + b;
  printf("%d", sum);
}
```


www.it-ebooks.info

A function can be defined to take any number of arguments, and they can have any data types. Just ensure the function is called with the same types and number of arguments. In this example, the function accepts two integer arguments and displays their sum.

```
sum(2, 3); /* "5" */
```

To be precise, *parameters* appear in function definitions, while *arguments* appear in function calls. However, the two terms are sometimes used interchangeably.

Void Parameter

In C, functions that leave out the `void` keyword from their parameter list are allowed to accept an unknown number of arguments. This is different from C++, where leaving out `void` means the same as including it: that the function takes no arguments. Therefore, to have the compiler ensure that no arguments are mistakenly passed to a parameterless function, it is necessary in C to include `void` in the parameter list.

```
/* Accepts no arguments */
void foo(void) {}

/* Accepts an unknown number of arguments */
void bar() {}
```

As of C99, the use of an empty parameter list has been deprecated and results in a warning from the compiler.

Return Statement

A function can return a value. The `void` keyword before the function's name is then replaced with the data type the function will return, and the `return` keyword is added to the function's body followed by an argument of the specified return type.

```
int getSum(int a, int b) {
  return a + b;
}
```

Return is a jump statement that causes the function to exit and return the specified value to the place where the function was called. To illustrate, the previous function can be passed as an argument to the `printf` function since it evaluates to an integer.

```
printf("%d", getSum(5, 10)); /* "15" */
```


www.it-ebooks.info

The `return` statement can also be used in a void function as a way to exit the function before the end block is reached.

```
void dummy(void) { return; }
```

The `main` function must be set to return an `int` type, but including an explicit return value is only required in the C89 standard. As of C90 the compiler will automatically add a `return` statement to the end of the `main` function if no such statement is present, and with the C99 standard this implicit return value is guaranteed to be zero.

```
int main(void) {
  return 0; /* optional */
}
```

Forward Declaration

An important thing to keep in mind in C is that a function must be declared before it can be called. This can either be achieved by placing the function's implementation before any references to it, or by adding a declaration of the function before it is called. This kind of forward declaration is known as a *prototype* and provides the compiler with the information needed to allow the function to be used before it has been defined.

```
void myFunction(int a); /* prototype */

int main(void) {
  myFunction(0);
}

void myFunction(int a) {}
```

The parameter names do not need to be included in the prototype; only the data types are required.

```
void myFunction(int);
```

In early versions of C an undeclared function that is referenced is implicitly declared as a function that returns an `int` and takes an unspecified number of parameters. Relying on this behavior is not recommended and usually results in a warning from the compiler. As of C99 this feature has been removed and will instead result in an error.

```
int main(void) {
  foo();
}

/* Warning: implicit declaration of foo */
int foo() { return 0; }
```


www.it-ebooks.info

Variable Parameter Lists

A function can be defined to accept a variable number of arguments, similar to the `printf` function. The parameter list of such a function must end with an ellipsis (...) and there must be at least one additional parameter. An `int` parameter is typically included to let the function know the number of extra arguments that are passed to it.

In the following example the function accepts a variable number of arguments that are summed up and returned to the caller. To access these arguments the stdarg.h header file is included. This header defines a new type, called `va_list`, and three functions that operate on variables of this type: `va_start`, `va_arg`, and `va_end`.

```
#include <stdio.h>
#include <stdarg.h>

int sum(int num, ...) {
  va_list args; /* variable argument list */
  int sum = 0, i = 0;

  va_start(args, num); /* initialize argument list */

  for (i = 0; i < num; i++) /* loop through arguments */
    sum += va_arg(args, int); /* get next argument */

  va_end(args); /* free memory */
  return sum;
}

int main(void) {
  printf("Sum of 1+2+3 = %d", sum(3,1,2,3)); /* 6 */
}
```

In contrast to C++, C does not allow function overloading or default parameter values. However, variable parameter lists can be used to implement functions that behave in similar ways.

Pass by Value

Variables are by default passed by value. This means that only a copy of the value is passed to the function. Therefore, changing the parameter in any way will not affect the original variable, and passing large variables back and forth can have a negative impact on performance.

```
#include <stdio.h>

void set(int i) { i = 1; }
```


www.it-ebooks.info

```
int main(void) {
  int x = 0;
  set(x);
  printf("%d", x); /* "0" */
}
```

Pass by Address

The alternative to passing by value is to use pointer syntax to instead pass the variable by address. When an argument is passed by address the parameter can be changed or replaced, and the change will affect the original variable.

```
void set(int* i) { *i = 1; }

int main(void) {
  int x = 0;
  set(&x);
  printf("%d", x); /* "1" */
}
```

Recall that arrays are in effect hidden pointers. As such they will automatically be passed by address, as shown in the following example.

```
void set(int a[]) { a[0] = 1; }

int main(void) {
  int x[] = { 0 };
  set(x);
  printf("%d", x[0]); /* "1" */
}
```

Return by Value or Address

In addition to passing variables by value or address, a variable may also be returned in one of these two ways. By default a function returns by value, in which case a copy of the value is returned to the caller.

```
int byVal(int i) { return i + 1; }

int main(void) {
  int a = 10;
  printf("%d", byVal(a)); /* "11" */
}
```


www.it-ebooks.info

To instead return by address the dereference operator is appended to the function's return type. The function must then return a variable and not an expression or literal, as is allowed when returning by value. The variable returned should never be a local variable since the memory to these variables is released when the function ends. Instead, return by address is commonly used to return an argument that has also been passed to the function by address.

```
int* byAdr(int* i) { (*i)++; return i; }

int main(void) {
  int a = 10;
  int *p = byAdr(&a);
  printf("%d", *p); /* "11" */
}
```

Inline Functions

When calling a function it is important to keep in mind that a certain performance overhead occurs. To potentially remove this overhead the programmer can recommend that the compiler inlines the calls to a specific function by using the `inline` function modifier. This keyword was added in the C99 standard. It is most suited for use with small functions that are called inside loops, as shown in the following example. Larger functions should not be inlined since this can significantly increase the size of the code, which may instead decrease performance.

```
inline int increment(int a) { return ++a; }

int main(void) {
  int i;
  for(i = 0; i < 100;) {
    i = increment(i);
  }
}
```

Note that the `inline` keyword is only a recommendation. The compiler may in its attempts to optimize the code choose to ignore this recommendation, and it may also inline functions that do not have the `inline` modifier.

www.it-ebooks.info

CHAPTER 11

Typedef

An alias for a type can be created using the `typedef` keyword followed by the type and alias name. By convention, uppercase letters are commonly used for these definitions.

```
typedef unsigned char BYTE;
```

Once defined, the alias can be used as a synonym for its specified type.

```
BYTE b; /* unsigned char */
```

`typedef` does not only work for existing types, but can also include a definition of a user-defined type – such as a struct, union, or enum. This can make a complex type easier to understand.

```
typedef struct { int points; } score;
score a, b, c;
a.points = 10;
```

If used properly a type alias can simplify a long or confusing type name, making the code easier to understand. Another benefit they provide is the ability to change the definition of a type from a single location, which can help make a program more portable.

www.it-ebooks.info

CHAPTER 12

Enum

Enum is a user-defined type consisting of a fixed list of named constants. In the following example, the enumeration type is called `color` and contains three constants: `RED`, `GREEN` and `BLUE`.

```
enum color { RED, GREEN, BLUE };
```

The `color` type can be used to create variables that may hold one of these constant values. In C, the variable declaration must be preceded by `enum`, whereas this is optional in C++.

```
int main(void) {
  enum color c = RED;
}
```

Enum variables may also be declared when the enum is defined, by placing the variable names before the final semicolon. This position is known as the declarator list.

```
enum color { RED, GREEN, BLUE } c, d;
```

Enum Example

The `switch` statement provides a good example of when enumerations can be useful. Compared to using ordinary constants, the enumeration has the advantage that it lets the programmer clearly specify what values a variable should be allowed to contain.

```
switch(c) {
  case RED:   break;
  case GREEN: break;
  case BLUE:  break;
}
```


www.it-ebooks.info

Bear in mind that enums in C are not typesafe, unlike their C++ equivalent. It is up to the programmer to ensure that enum types and constants are used correctly, as most compilers will not enforce this. Enums in C simply provide a way to group a set of integer constants and have them be automatically numbered. For this purpose the enum identifier is not strictly necessary and may optionally be omitted.

```
enum { RED, GREEN, BLUE } c;
```

Enum Constant Values

Enumerated constants are of the `int` type. Usually there is no need to know the underlying values that these constants represent, but in some cases it can be useful. By default, the first constant in the enum list has the value zero and each successive constant is one value higher.

```
enum color {
  RED   /* 0 */
  GREEN /* 1 */
  BLUE  /* 2 */
};
```

These default values can be overridden by assigning values to the constants. The values can be computed and do not have to be unique.

```
enum color {
  RED   = 5,          /* 5 */
  GREEN = RED,        /* 5 */
  BLUE  = GREEN + 2,  /* 7 */
  ORANGE              /* 8 */
};
```

Enum Conversions

The compiler can implicitly convert an enumerated constant to an integer. An integer can also be converted back into an enum variable.

```
int i = RED;
enum color c = i;
```

Some compilers warn when an integer is assigned to an enum variable since this makes it possible to assign a value that is not one of its specified constants. To suppress this warning an explicit type cast can be used.

```
enum color c = (enum color)i;
```


www.it-ebooks.info

Enum Scope

An enum does not have to be declared globally. It can also be placed locally within a function, in which case it will only be usable within that function after where it has been defined.

```
/* Global enum */
enum speed { SLOW, NORMAL, FAST };

int main(void) {
  /* Local enum */
  enum color { RED, GREEN, BLUE };
}
```


www.it-ebooks.info

CHAPTER 13

Struct

A struct or structure is a user-defined type used for grouping a collection of related variables under a single name. To define a structure you use the `struct` keyword, followed by an optional identifier and a code block containing variable declarations. The definition of this new type is ended with a semicolon.

```
struct point {
  int x, y;
};
```

Unlike arrays, structs allow data items of different kinds to be combined. Structs may contain variables, pointers, arrays, or other user-defined types. In contrast to C++, structs in C may not contain functions.

Struct Objects

To declare a variable of a struct type, the `struct` keyword is followed by the type name and the variable identifier. Variables of struct type are commonly referred to as objects or instances.

```
int main(void) {
  struct point p; /* object declaration */
}
```

Objects may also be created when the struct is defined, by placing the object names before the final semicolon. This position is called the declarator list. If the optional struct identifier is left out, this becomes the only way to create objects of the struct type. Such a struct without an identifier is called an unnamed struct and provides a way for programmers to prevent any more instances of the type from being created.

```
struct /* unnamed struct */
{
  int x, y;
} a, b; /* object declarations */
```


www.it-ebooks.info

It is common in C to use `typedef` when defining structures. This aliasing removes the need to include the `struct` keyword when declaring objects of the struct type, resulting in the shorter syntax used in C++.

```
typedef struct point point;

struct point {
  int x, y;
};

int main(void) {
  point p; /* struct omitted */
}
```

Member Access

Variables of a struct type are called fields or members. These fields are accessed using the member of operator (`.`) prefixed by the object name. Fields of an object are by default undefined, so it is important to assign them a value before they are read.

```
int main(void) {
  point p;
  p.x = 1;
  p.y = 2;
}
```

Similar to an array, struct objects may also be initialized when they are declared by enclosing the values in curly brackets. The values are then assigned in order to the corresponding members of the struct. This way of assigning values to a composite type is known as aggregate initialization.

```
struct point {
  int x, y;
} r = { 1, 2 }; /* assigns x and y */

int main(void) {
  point p = { 1, 2 };
}
```

C99 introduced designated initializers, which allow structures to be initialized in any order by specifying the names of the fields. Any omitted fields will be automatically initialized to 0.

```
int main(void) {
  point p = { .y = 2, .x = 1 };
}
```


www.it-ebooks.info

Struct Pointers

A struct is a value type, not a reference type like an array. As such, any assignment or argument passing for objects will copy the field values and not the object reference. This is different from many modern languages where composite types are automatically assigned and passed by reference.

```
int main(void) {
  point p = { 1, 2 };
  point r = p; /* copies field values */
}
```

For large structures the performance cost of this copy operation may be significant. Therefore it is common to use pointers when passing objects to functions, to avoid having to copy and return the whole object.

```
void init_struct(point* a) {
  (*a).x = 1;
  (*a).y = 2;
}

int main(void) {
  point p;
  init_struct(&p);
}
```

As shown in this example, the pointer must be dereferenced before the member of the operator can be used to access the fields. Since this operation is so common there is a syntactical shortcut available, known as the infix operator (`->`), which automatically dereferences the pointer.

```
point p;
point* r = &p;
r->x = 1; /* same as (*r).x = 1; */
```


www.it-ebooks.info

Bit Fields

The C programming language offers a way to optimize memory use within a struct type by allowing the bit length of integer fields to be specified. Such a field is called a bit field, and its length is set by placing a colon after the field name followed by the number of bits. The length must be less than or equal to the bit length of the specified type.

```
struct my_bits
{
  unsigned short f1 : 1;
  unsigned short f2 : 1;
  unsigned short id : 10;
} a;
```

Bit fields are packed as compactly as possible, while keeping in mind that the size of an object needs to be a multiple of the size of the types it contain. In this case the needed 12 bits will require 16 bits to be reserved for the object, as that is the size of the short type. Had bit fields not been used, the 3 shorts and consequently the struct would occupy 48 bits instead.

```
int main(void) {
  printf("%d bytes", sizeof(a)); /* "2 bytes" */
}
```

This feature is useful when programming for embedded systems, where hardware resources may be very constrained.

www.it-ebooks.info

CHAPTER 14

Union

The union type is identical to the struct type, except that all fields share the same memory position. Therefore, the size of a union is the size of the largest field it contains. In the following code this is the integer field, which is 4 bytes large.

```
union mix {
  char c;  /* 1 byte */
  short s; /* 2 bytes */
  int i;   /* 4 bytes */
};
```

Given this memory sharing, the union type can only be used to store one value at a time, because changing one field will overwrite the value of the others.

```
int main(void) {
  union mix m;
  m.c = 0xFF; /* set first 8 bits */
  m.s = 0;    /* reset first 16 bits */
}
```

The benefit of a union, in addition to efficient memory usage, is that it provides multiple ways of using the same memory location. For example, the following union has three data members that allow access to the same group of 4 bytes in different ways.

```
union mix {
  char c[4];                  /* 4 bytes */
  struct { short hi, lo; } s; /* 4 bytes */
  int i;                      /* 4 bytes */
} m;
```


www.it-ebooks.info

The `int` field accesses all 4 bytes at once, the struct 2 bytes at a time, and the `char` array allows each byte to be referenced individually. The bit pattern for this is illustrated in the next example. Keep in mind that the internal order of bytes for primitive data types is not defined in C. Because of this, the order of the 4 bytes that make up the `int` may be reversed on some platforms.

```
m.i=0xFF00F00F; /* 11111111 00000000 11110000 00001111 */
m.s.lo;         /* 11111111 00000000                  */
m.s.hi;         /*                   11110000 00001111 */
m.c[3];         /* 11111111                            */
m.c[2];         /*          00000000                   */
m.c[1];         /*                   11110000          */
m.c[0];         /*                            00001111 */
```


www.it-ebooks.info

CHAPTER 15

Type Conversions

Converting an expression from one type to another is known as type casting or type conversion. This can be done either implicitly by the compiler or explicitly with code.

Implicit Conversions

An implicit conversion is performed automatically by the compiler when an expression needs to be converted into one of its compatible types. For example, any conversions between the primitive data types can be done implicitly.

```
long l = 5;   /* int -> long */
double d = l; /* long -> double */
```

These implicit conversions can also take place within an expression, allowing you to mix different primitive types together. When types of different sizes are involved, the result will be of the larger type, so an `int` and `double` will produce a `double` value.

```
double d = 5 + 2.5; /* int -> double */
```

Implicit conversions of primitive types can be further grouped into two kinds: *promotion* and *demotion*. Promotion occurs when an expression gets implicitly converted into a larger type, and demotion occurs when converting an expression to a smaller type.

```
/* Promotion */
long l = 5;   /* int promoted to long */
double d = l; /* long promoted to double */

/* Demotion */
int i = 10.5; /* warning: possible loss of data */
char c = i;   /* warning: possible loss of data */
```

Because a demotion can result in the loss of information, these conversions generate a warning on many compilers. If the potential information loss is intentional, the warning can be suppressed by using an explicit cast.

www.it-ebooks.info

Explicit Conversions

An explicit cast is performed by placing the desired data type in parentheses to the left of the expression that needs to be converted.

```
int i = (int)10.5; /* double demoted to int */
char c = (char)i;  /* int demoted to char */
```

Keep in mind that casting a variable only makes it temporarily evaluate as a different type; it does not change the variable's underlying type.

www.it-ebooks.info

CHAPTER 16

Storage Classes

Every variable has a storage class that determines its scope and lifetime. These storage classes include the following: auto, register, extern, and static. Each of these classes is also a keyword that can be placed before the data type to determine to which storage class a variable belongs.

Auto

The default storage class for local variables is auto, which can be explicitly specified with the `auto` keyword. Memory for automatic variables is allocated when the code block is entered and freed upon exit. The scope of these variables is local to the block in which they are declared, as well as any nested blocks.

```
int main(void) {
  auto int localVar; /* auto variable */
}
```

Register

The register storage class hints to the compiler that a local variable will be heavily used and should therefore be kept in a CPU register instead of RAM memory to provide quicker access. Variables of the register storage class cannot use the address-of operator (&) since registers do not have memory addresses. They also cannot be larger than the register size, which is usually the same as the processor's word size.

```
int main(void) {
  register int counter; /* register variable */
}
```

Use of the `register` keyword has become deprecated since modern compilers are automatically able to optimize which variables should be stored in registers.

www.it-ebooks.info

External

The external storage class, specified with the `extern` keyword, is used to reference a variable or function defined in another compilation unit. A compilation unit consists of a source file plus any included header files. Functions default to the external storage class, so marking function prototypes with `extern` is optional.

```
/* app.c */
extern void foo(void); /* declared function */
int main(void) {
  foo(); /* external function call */
}

/* func.c */
void foo(void) {} /* defined function */
```

When `extern` is used with a global variable it becomes declared but not defined, so no memory is allocated for it. This tells the compiler that the variable is defined elsewhere. As with functions, it is necessary to declare global variables before they can be used in a compilation unit outside the one containing the definition.

```
/* app.c */
int globalVar; /* defined variable */

int main(void) {
  globalVar = 1;
}

/* func.c */
extern int globalVar; /* declared variable */

int foo(void) {
  globalVar++;
}
```

Keep in mind that a global variable or function may be declared externally multiple times in a program, but they may be defined only once.

Static

The static storage class restricts the scope of a global variable or function to the compilation unit that defines it. The lifetime of static entities is to last for the whole program duration, which is the same as entities belonging to the external storage class.

```
/* Only visible within this compilation unit */
static int myInt;
static void myFunc(void) {}
```


www.it-ebooks.info

Local variables may be declared as static to make the function preserve the variable for the duration of the program. A static local variable is only initialized once – when execution first reaches the declaration – and that declaration is then ignored every subsequent time the execution passes through.

```
/* Store number of calls to this function */
void myFunc(void) {
  static int count = 0;
  count++;
}
```

Knowing that a code entity can only be accessed and altered within a limited scope simplifies debugging as it reduces potential dependencies between compilation units. Therefore, it is a good idea to declare all global variables and functions as static, unless they have an actual need to be exposed outside of their own compilation unit.

Volatile

Another type modifier in C is `volatile`. This modifier tells the compiler that a variable's value may be changed by something external to the program and that the value must therefore be reread from memory every time it is accessed. Like const, the `volatile` modifier can appear either before or after the type, and it can be used together with a storage class modifier.

```
volatile int var; /* recommended order */
int volatile var; /* alternative order */
```

In the following example the function waits for a variable to be set by some external event. Without the `volatile` modifier the compiler may decide to optimize this loop condition by replacing it with an infinite loop, as it assumes the variable is never changed.

```
volatile int ext = 0;

void poll(void) {
  while(ext == 0) {}
}
```

Global variables should be declared volatile if their value is shared and can be changed externally. This can occur because an interrupt service routine modifies the variable, or because it is changed by another thread in a multi-threaded application. A third use case for volatile is with memory-mapped peripheral devices, which can change hardware registers outside of the program's control.

www.it-ebooks.info

CHAPTER 17

Constants

A constant is a variable with a value that cannot be changed once it has been assigned. This allows the compiler to enforce that a variable's value is not changed anywhere in the code by mistake.

Constant Variables

A variable can be made into a constant by adding the `const` keyword either before or after the data type. This modifier makes the variable read-only, and it must therefore be assigned a value at the same time as it is declared. Attempting to change the constant anywhere else results in a compile-time error.

```
const int var = 5;   /* recommended order */
int const var2 = 10; /* alternative order */
```

Constant Pointers

When it comes to pointers, `const` can be used in two ways. First, the pointer can be made constant, which means that it cannot be changed to point to another location.

```
int myPointee;
int* const p = &myPointee; /* constant pointer */
```

Second, the pointee can be declared constant. This means that the variable pointed to cannot be modified through this pointer.

```
const int* q = &var; /* constant pointee */
```

It is possible to declare both the pointer and the pointee as constant to make them both read-only.

```
const int* const r = &var; /* constant pointer & pointee */
```


www.it-ebooks.info

Referencing a constant from a non-constant pointer will produce a warning or error on most compilers. This is because such an assignment makes it possible to accidentally rewrite the constant's value.

```
int* s = &var; /* error: const to non-const assignment */
```

Constant Parameters

Function parameters can be made constant to prevent them from being altered within the function. The main benefit of this is to let programmers know that the function leaves its pointer arguments untouched. When used consistently, it can also provide information about which functions can be expected to modify their pointer arguments.

```
void foo(const int* x) {
  if (x != NULL) {
    int i = *x; /* allowed */
    *x = 1;     /* compile-time error */
  }
}
```

Constant Guideline

In general, it is a good idea to always declare variables constant if they do not need to be modified. This ensures that the variables are not changed anywhere in the program by mistake, which in turn can help prevent bugs.

www.it-ebooks.info

CHAPTER 18

Preprocessor

The preprocessor is a text substitution tool that modifies the source code before it is compiled. This modification is done according to the preprocessor directives that are included in the source files. The directives are easily distinguished from normal programming code in that they all start with a hash sign (#). They must always appear as the first non-whitespace character on a line and do not need to end with a semicolon. The following table shows the preprocessor directives available in C along with their functions.

Directive	Description
`#include`	File include
`#define #undef`	Define macro Undefine macro
`#ifdef #ifndef`	If macro defined If macro not defined
`#if #elif #else #endif`	If Else if Else End if
`#line #error #pragma`	Set line number Abort compilation Set compiler option

Including Source Files

The `#include` directive inserts the contents of a file into the current source file. Its most common use is to include header files (.h), both user-defined and library ones. Library header files are enclosed between angle brackets (<>). This tells the preprocessor to search for the header in the default directory where it is configured to look for standard header files.

```
#include <stdio.h> /* search library directory */
```

Header files that you create for your own program are enclosed within double quotes ("""). The preprocessor will then search for the file in the same directory as the current file. In case the header is not found there, the preprocessor will then search among the standard header files.

```
#include "myfile.h" /* search current, then default */
```


www.it-ebooks.info

The double quoted form can also be used to specify an absolute or relative path to the file.

```
#include "c:\myfile.h" /* absolute path */
#include "..\myfile.h" /* relative path */
```

Define

Another important directive is #define, which is used to create compile-time constants, also called macros. After the directive, the name of the constant is specified followed by what it will be replaced by.

```
#define PI 3.14 /* macro definition */
```

The preprocessor will go through the code and change any occurrences of this constant with whatever comes after it in its definition until the end of the line.

```
float f = PI; /* f = 3.14 */
```

By convention, constants should be named in uppercase letters with each word separated by an underscore. That way they are easy to spot when reading the source code.

Undefine

A #define directive should not be used to directly override a previously defined macro. Doing so will give a compiler warning, unless the macro definitions are the same. In order to redefine an existing macro, it first needs to be undefined using the #undef directive. Attempting to undefine a macro that is not currently defined will not generate a warning.

```
#undef PI /* undefine */
#undef PI /* allowed */
```

Predefined Macros

There are a number of macros that are predefined by the compiler. To distinguish them from other macros, their names begin and end with two underscores. The standard macros that all ANSI C compliant compilers include are listed in the following table.

www.it-ebooks.info

Directive	Description
__FILE__	The name and path for the current file.
__LINE__	The current line number.
__DATE__	The compilation date in MM DD YYYY format.
__TIME__	This compilation time in HH:MM:SS format.
__func__	The name of the current function. Added in C99.
__STDC__	Defined as 1 if the compiler complies with the ANSI C standard.

A common use for predefined macros is to provide debugging information. To give an example, the following error message includes the file name and line number where the message occurs.

```
printf("Error in %s at line %d", __FILE__, __LINE__);
```

Macro Functions

A macro can be made to take arguments. This allows them to define compile-time functions. For example, the following macro function gives the square of its argument.

```
#define SQUARE(x) ((x)*(x))
```

The macro function is called just as if it was a regular C function. Keep in mind that for this kind of function to work, the arguments must be known at compile time.

```
int x = SQUARE(2); /* 4 */
```

Note the extra parentheses in the macro definition that are used to avoid problems with operator precedence. Without the parentheses the following example would give an incorrect result, as the multiplication would then be carried out before the addition.

```
#define SQUARE(x) x*x

int main(void) {
  int x = SQUARE(1+1); /* 1+1*1+1 = 3 */
}
```

To break a macro function across several lines the backslash character can be used. This will escape the newline character that marks the end of a preprocessor directive. For this to work there must not be any whitespace after the backslash.

```
#define MAX(a,b)  \
            a>b ? \
            a:b
```


www.it-ebooks.info

Although macros can be powerful, they tend to make the code more difficult to read and debug. Macros should therefore only be used when they are necessary and should always be kept short. C code such as constant variables, enums, and inline functions can often accomplish the same goal more efficiently and safely than `#define` directives can.

Conditional Compilation

The directives used for conditional compilation can include or exclude part of the source code if a certain condition is met. First, there is the `#if` and `#endif` directives, which specifies a section of code that will only be included if the condition after the `#if` directive is true. Note that this condition must evaluate to a constant expression.

```
#define DEBUG_LEVEL 3

#if DEBUG_LEVEL > 2
 /* ... */
#endif
```

Just as with the C if statement, any number of `#elif` (else if) directives and one final `#else` directive can be included.

```
#if DEBUG_LEVEL > 2
 /* ... */
#elif DEBUG_LEVEL == 2
 /* ... */
#else
 /* ... */
#endif
```

Conditional compilation also provides a useful means of temporarily commenting out large blocks of code for testing purposes. This often cannot be done with the regular multi-line comment since they cannot be nested.

```
#if 0
 /* Removed from compilation */
#endif
```

Compile if Defined

Sometimes, a section of code should only be compiled if a certain macro has been defined, irrespective of its value. For this purpose two special operators can be used: `defined` and `!defined` (not defined).

www.it-ebooks.info

```
#define DEBUG

#if defined DEBUG
 /* ... */
#elif !defined DEBUG
 /* ... */
#endif
```

The same effect can also be achieved using the directives `#ifdef` and `#ifndef` respectively. The `#ifdef` section is only compiled if the specified macro has been previously defined. Note that a macro is considered defined even if it has not been given a value.

```
#ifdef DEBUG
 /* ... */
#endif
#ifndef DEBUG
 /* ... */
#endif
```

Error and Warning

When the `#error` directive is encountered the compilation is aborted. This directive can be useful, for example, to determine whether or not a certain line of code is being compiled. It can optionally take a parameter that specifies the description of the generated compilation error.

```
#error "Compilation aborted"
```

Many C compilers also include the non-standard directive `#warning`. This directive displays a warning message without halting the compilation.

```
#warning "Function X is deprecated, use Y instead"
```

Line

A less commonly used directive is `#line`, which can change the line number that is displayed when an error occurs during compilation. Following this directive the line number will as usual be increased by one for each successive line. The directive can take an optional string parameter that sets the filename that will be shown when an error occurs.

```
#line 5 "myapp.c"
```


www.it-ebooks.info

Pragma

The last standard directive is #pragma, or pragmatic information. This directive is used to specify options to the compiler; and as such, they are vendor specific. To give an example, #pragma message can be used with many compilers to output a string to the build window.

```
/* Show compiler message */
#pragma message "Compiling " __FILE__ "..."
```


www.it-ebooks.info

CHAPTER 19

Memory Management

In the examples so far, the programs have only had as much memory available as has been declared for the variables at compile time. This is referred to as static allocation. If any additional memory is needed at runtime it becomes necessary to use dynamic allocation. The C standard library provides several functions for managing dynamically allocated memory, including: `malloc`, `free`, and `realloc`. These functions are found in the stdlib.h header file.

Malloc

The `malloc` function takes a size in bytes and returns a pointer to a block of free memory of that size. This dynamically allocated memory is uninitialized and can only be accessed through pointers.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  /* Dynamic memory allocation */
  char* ptr = malloc(sizeof(char) * 5);
}
```

The `sizeof` operator is used to get the number of bytes for the given data type on the current system. This number is here multiplied by five to allocate a block large enough to contain exactly five chars, provided that the system has that much free memory available. If it does not, `malloc` returns null to signal that it has failed to allocate the memory, in which case the function may need to signal to its caller that it too has failed.

```
char* ptr = malloc(sizeof(char) * 5);

if (ptr == NULL) {
  /* No memory allocated, exit function */
  return -1;
}
```


www.it-ebooks.info

By convention, functions that return a pointer, such as `malloc`, uses the value NULL to indicate failure. This is different from functions that return a value, which traditionally use 0 to indicate success and -1 to signal failure. Additional return values can be used to give the user of the function more detailed information about the return state.

Free

An important thing to remember about dynamic allocation is that this memory will not be released when the pointer goes out of scope, as with local variables. Instead, the memory has to be manually released with the function `free`, which releases the memory block at the specified address.

```
free(ptr); /* release allocated memory */
```

This allows you to control the lifetime of a dynamically allocated object, but it also means that you are responsible for freeing that memory once it is no longer needed. Forgetting to free dynamic memory will give the program unwanted memory leaks, because that memory will stay allocated until the program shuts down.

As mentioned before, the pointer to the released memory should be set to NULL immediately to show that it is no longer set to a valid reference. This is especially important for pointers that are repeatedly allocated and freed within a program.

```
ptr = NULL; /* null pointer */
```

It is interesting to note that the function `free` only accepts one argument, the starting address for the memory block. The actual size of the block does not need to be provided. This is because the implementation of `malloc` and `free` keeps track of the size of each block as it is allocated, typically by storing it next to the block.

Realloc

An allocated memory block can be resized with the `realloc` function. This function takes two arguments: the pointer to a previously allocated memory block and the new total size requested. If the pointer passed to `realloc` is NULL then the function behaves as `malloc`.

```
/* Increase size of memory block */
char* new_ptr = realloc(ptr, sizeof(char) * 10);
```

The return value is here stored in a new pointer, in case `realloc` fails to allocate the extra memory and returns NULL. This prevents the only reference to the previously allocated memory block from being lost, which would lead to a memory leak.

```
/* On failure, free memory and exit */
if (new_ptr == NULL) {
  free(ptr);
```


www.it-ebooks.info

```
  return -1;
}
/* On success, update pointer */
else {
  ptr = new_ptr;
}
```

Void Pointer

It is sometimes necessary to use pointers without regard to the type they reference. This is achieved by specifying the pointer type as `void*`, known as a void pointer. A void pointer can store the address of any type of variable and can be cast to any pointer type, making them useful as a universal pointer. This type is what allows the `free` function to accept any pointer argument, and how `malloc` returns a pointer that can be cast to any pointer type. The following example illustrates how the void pointer can be used to change the values of two variables of different types.

```
int i;
char c;

void *vptr = &i;
*((int*)vptr) = 1;

vptr = &c;
*((char*)vptr) = 'a';
```

Note that a void pointer may not be dereferenced without first casting it to the appropriate pointer type. The compiler is unable to check that this type cast is valid, which is why void pointers should be used with care.

www.it-ebooks.info

CHAPTER 20

Command Line Arguments

When a C program is executed it can accept arguments that are passed to it from the command line. These command line arguments are useful for controlling the behavior of the program from outside the code. The arguments are passed to the `main` function, which is then set to accept two parameters as shown here.

```
int main(int argc, char* argv[]) {}
```

The integer `argc` is the number of arguments passed, and `argv` is a pointer array to the supplied command line arguments. The first argument `argv[0]` is the name of the program, so the argument count in `argc` begins with 1 when no arguments are passed. If the program name is not available on the host environment, the first element will be an empty string instead. A simple program that prints all arguments is given here.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  int i;
  for(i=0; i < argc; i++) {
    printf("Argument %d is: %s\n", i, argv[i]);
  }
}
```

When a program is executed from a terminal window, the string arguments are listed after the filename, separated by spaces. If the argument itself contains a space it can be delimited by double quotes. In the following example, the previous program is compiled and executed with two arguments.

```
gcc myapp.c -o myapp.exe
./myapp.exe test "Hello World"
Argument 0 is myapp.exe
Argument 1 is test
Argument 2 is Hello World
```


www.it-ebooks.info

CHAPTER 21

Headers

As a project grows it is common to split the code up into different source files. When this happens the interface and implementation are generally separated. The interface is placed in a header file, which commonly has the same name as the source file and a .h file extension. This header file contains forward declarations for the source file entities that need to be accessible to other compilation units in the project.

Why to Use Headers

C requires everything to be declared before it can be used. It is not enough to just compile the necessary source files in the same project. For example, if a function is placed in func.c, and a second file named app.c in the same project tries to call it, the compiler will report that it cannot find the function (or, prior to C99, that it has implicitly declared it).

```
/* func.c */
void myFunc(void) {
  /* ... */
}

/* app.c */
int main(void) {
  myFunc(); /* error: myFunc identifier not found */
}
```

To make this work as intended, the function's prototype has to be included in app.c.

```
/* app.c */
void myFunc(void); /* prototype */

int main(void) {
  myFunc(); /* ok */
}
```


www.it-ebooks.info

This can be made more convenient if the prototype is placed in a header file named func.h and this header is included in app.c through the use of the `#include` directive. This way, when changes are made to func.c, there is no need to update the prototypes in app.c. Furthermore, any source file that wants to use the shared code in func.c can just include this one header.

```
/* func.h */
void myFunc(void); /* prototype */

/* app.c */
#include "func.h"
```

What to Include in Headers

As far as the compiler is concerned there is no difference between a header file and a source file. The distinction is only conceptual. The key idea is that the header should contain the interface of the implementation file – that is, the code that other source files will need to use. This may include shared macros, constants, and type definitions, as those shown here.

```
/* app.h - Interface */
#define DEBUG 0
const double PI = 3.14;
typedef unsigned long ulong;
```

The header can also contain prototypes of the shared functions defined in the source file. Internal functions used only within the source file should be left out of the header, to keep them private from the rest of the program.

```
void myFunc(void); /* prototype */
```

Additionally, shared global variables are typically declared as extern in the header, while their definitions lay in the source file.

```
/* app.h */
extern int myGlobal;

/* app.c */
int myGlobal = 0;
```

It should be noted that the use of shared global variables is discouraged. This is because the larger a program becomes, the more difficult it is to keep track of which functions access and modify these variables. The preferred method is to instead pass variables to functions only as needed, in order to minimize the scope of those variables.

www.it-ebooks.info

The header should not include any executable statements, with one exception. A shared function that is declared as `inline` needs to be defined in the header. Otherwise, the compiler will not have the definition necessary for inlining the function available to it.

```
/* app.h */
inline void inlineFunc(void) {}
```

If a header requires other headers it is common to include those files as well, to make the header stand alone. This ensures that everything needed is included in the correct order, solving potential dependency problems for every source file that needs the header.

```
/* app.h */
#include <stddef.h>
void mySize(size_t);
```

Note that since headers mainly contain declarations, any extra headers included should not affect the size of the program, although they may slow down the compilation.

Include Guards

An important thing to bear in mind when using header files is that a code entity, such as a constant, typedef, or enum, may only be defined once in every project. Consequently, including the same header file more than once will often result in compilation errors. The standard way to prevent this is to use a so-called include guard. An include guard is created by enclosing the body of the header in a `#ifndef` section that checks for a macro specific to that header file. Only when the macro is not defined is the file included and the macro is then defined, which effectively prevents the file from being included again.

```
/* app.h */
#ifndef APP_H
#define APP_H
/* ... */
#endif
```


www.it-ebooks.info

Index

A

Arithmetic operators, 15
Arrays
    array assignment, 21
    array size, 22
    declaration and allocation, 21
    multi-dimensional, 22
    and pointers, 22
Assignment operators, 15

B

Bit field, 46
Bitwise operators, 17

C

Command line arguments, 67
Comparison operators, 16
Compile and run
    comments, 4
    console compilation, 3–4
    visual studio compilation, 3
Constants
    guideline, 56
    parameters, 56
    pointers, 55–56
    variable, 55

D

Declarator list, 39
Dereference operator, 19–20
Do-while loop, 29

E

Enum
    color, 39
    constant values, 40
    conversions, 40
    scope, 41
    switch statement, 39
    variables, 39
Explicit conversions, 50

F

For loop, 29
Free function, 64
Functions
    calling, 31
    definition, 31
    forward declaration, 33
    inline, 36
    parameters, 31
    pass by address, 35
    pass by value, 34
    return by value/address, 35
    return statement, 32
    variable parameter lists, 34
    void parameter, 32

G

GNU Compiler Collection (GCC), 3–4
Goto statement, 30

H

Headers
    function's prototype, 69
    include guards, 71

www.it-ebooks.info

Headers (*cont.*)
interface and implementation, 69
shared global variables, 70
Hello World
integrated development environment (IDE), 1
IntelliSense, 2
printf function, 2
project, 1
return statement, 2
source file, 1

I, J, K

If statement, 27–28
Implicit conversions, 49
Increment and decrement operators, 16
Integrated development environment (IDE), 1
IntelliSense, 2

L

Logical operators, 17
Loops
break and continue, 30
do-while, 29
for, 29
goto statement, 30
while, 29

M

Malloc function, 63–64
Memory management
description, 63
free function, 64
malloc function, 63–64
realloc function, 64–65
void pointer, 65

N

Null pointer, 20

O

Operator precedence, 18

P, Q

Pointers
address-of operator, 19
data type, 19
dereferencing, 19–20
null, 20
pointing to pointer, 20
Preprocessor
ANSI C compliant compilers, 58
conditional compilation, 60
define macro (#define), 58
description, 57
error and warning, 61
file include (#include), 57
If macro defined (#ifdef), 61
line, 61
macro functions, 59–60
pragma, 62
undefine macro (#undef), 58

R

Realloc function, 64–65

S

Storage classes
auto, 51
external, 52
register, 51
static, 52–53
volatile, 53
String
character array, 23
escape character, 23
null character, 23
strcat (string concatenation), 25
strcpy (string copy), 25
strlen function, 25
Struct/structure
bit field, 46
member access, 44
objects, 43–44
pointers, 45
user-defined type, 43
Switch statement, 28

www.it-ebooks.info

T

Ternary operator, 28
Type conversions
    explicit, 50
    implicit, 49
Typedef, 37

U

Union, 47–48

V

Variables
    assigning, 6
    bool type, 12
    char type, 11
    data types, 5
    declaration, 6
    floating-point types, 10
    global, 13
    integer size, 9–10
    integer types, 8
    literal suffixes, 11
    local, 12–13
    printf function, 7–8
    signed and unsigned, 8–9
Void pointer, 65

W, X, Y, Z

While loop, 29

www.it-ebooks.info

C Quick Syntax Reference

Mikael Olsson

Apress®

www.it-ebooks.info

C Quick Syntax Reference

Copyright © 2015 by Mikael Olsson

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4302-6499-6

ISBN-13 (electronic): 978-1-4302-6500-9

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Michael Thomas
Editorial Board: Steve Anglin, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Steve Weiss
Coordinating Editor: Mark Powers
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

www.it-ebooks.info

Contents

About the Author xi

About the Technical Reviewer xiii

Introduction xv

Chapter 1: Hello World 1

Creating a Project 1

Adding a Source File 1

Hello World 2

IntelliSense 2

Chapter 2: Compile and Run 3

Visual Studio Compilation 3

Console Compilation 3

Comments 4

Chapter 3: Variables 5

Data Types 5

Declaring Variables 6

Assigning Variables 6

Printing Variables 7

Integer Types 8

Signed and Unsigned 8

Sized Integers 9

Floating-Point Types 10

www.it-ebooks.info

Literal Suffixes 11
Char Type 11
Bool Type 12
Variable Scope 12

Chapter 4: Operators 15

Arithmetic Operators 15
Assignment Operators 15
Combined Assignment Operators 15
Increment and Decrement Operators 16
Comparison Operators 16
Logical Operators 17
Bitwise Operators 17
Operator Precedence 18

Chapter 5: Pointers 19

Creating Pointers 19
Dereferencing Pointers 19
Pointing to a Pointer 20
Null Pointer 20

Chapter 6: Arrays 21

Array Declaration and Allocation 21
Array Assignment 21
Multi-Dimensional Arrays 22
Arrays and Pointers 22
Array Size 22

www.it-ebooks.info

Chapter 7: String 23
Escape Characters 23
String Functions 24
Chapter 8: Conditionals 27
If Statement 27
Switch Statement 28
Ternary Operator 28
Chapter 9: Loops 29
While Loop 29
Do-While Loop 29
For Loop 29
Break and Continue 30
Goto Statement 30
Chapter 10: Functions 31
Defining Functions 31
Calling Functions 31
Function Parameters 31
Void Parameter 32
Return Statement 32
Forward Declaration 33
Variable Parameter Lists 34
Pass by Value 34
Pass by Address 35
Return by Value or Address 35
Inline Functions 36

www.it-ebooks.info

Chapter 11: Typedef **37**

Chapter 12: Enum **39**

Enum Example 39

Enum Constant Values 40

Enum Conversions 40

Enum Scope 41

Chapter 13: Struct **43**

Struct Objects 43

Member Access 44

Struct Pointers 45

Bit Fields 46

Chapter 14: Union **47**

Chapter 15: Type Conversions **49**

Implicit Conversions 49

Explicit Conversions 50

Chapter 16: Storage Classes **51**

Auto 51

Register 51

External 52

Static 52

Volatile 53

Chapter 17: Constants **55**

Constant Variables 55

Constant Pointers 55

Constant Parameters 56

Constant Guideline 56

www.it-ebooks.info

Chapter 18: Preprocessor 57
Including Source Files 57
Define 58
Undefine 58
Predefined Macros 58
Macro Functions 59
Conditional Compilation 60
Compile if Defined 60
Error and Warning 61
Line 61
Pragma 62
Chapter 19: Memory Management 63
Malloc 63
Free 64
Realloc 64
Void Pointer 65
Chapter 20: Command Line Arguments 67
Chapter 21: Headers 69
Why to Use Headers 69
What to Include in Headers 70
Include Guards 71

Index 73

www.it-ebooks.info

About the Author

Mikael Olsson is a professional web entrepreneur, programmer, and author. He works for an R&D company in Finland where he specializes in software development. In his spare time he writes books and creates websites that summarize various fields of interest. The books he writes are focused on teaching their subject in the most efficient way possible, by explaining only what is relevant and practical without any unnecessary repetition or theory. The portal to his online businesses and other websites is `Siforia.com`.

www.it-ebooks.info

About the Technical Reviewer

Michael Thomas has worked in software development for over 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has over 10 years of experience working with mobile devices. His current focus is in the medical sector using mobile devices to accelerate information transfer between patients and health care providers.

www.it-ebooks.info