Beginning C

From Novice to Professional, Fourth Edition

Ivor Horton

Apress®

Beginning C: From Novice to Professional, Fourth Edition

Copyright © 2006 by Ivor Horton

ISBN-13 (pbk): 978-59059-735-4

ISBN-10 (pbk): 1-59059-735-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matthew Moodie
Technical Reviewer: Stan Lippman
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole LeClerc
Copy Editor: Jennifer Whipple
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Susan Glinert
Proofreader: Lori Bring
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 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 `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/Download section.

This book is for the latest member of the family, Henry James Gilbey, who joined us on July 14, 2006. He hasn't shown much interest in programming so far, but he did smile when I asked him about it so I expect he will.

Contents at a Glance

Contents

About the Author

IVOR HORTON started out as a mathematician, but after graduating he was lured into messing around with computers by a well-known manufacturer. He has spent many happy years programming occasionally useful applications in a variety of languages as well as teaching scientists and engineers to do likewise. He has extensive experience in applying computers to problems in engineering design and manufacturing operations. He is the author of a number of tutorial books on programming in C, C++, and Java. When not writing programming books or providing advice to others, he leads a life of leisure.

Acknowledgments

I'd like to thank Gary Cornell for encouraging me to produce this new updated edition of *Beginning C: From Novice to Professional.* I'm particularly grateful to Stan Lippman for taking the time to cast his critical eye over the entire draft text; he did not pull any punches in his extensive review comments and the book is surely better as a result. My thanks to all the people at Apress, who have done their usual outstandingly professional job of converting my initial text with all its imperfections into this finished product. Any imperfections that remain are undoubtedly mine.

My sincere thanks to those readers of previous editions of this book who took the trouble to point out my mistakes and identify areas that could be better explained. I also greatly appreciate all those who wrote or e-mailed just to say how much they enjoyed the book or how it helped them get started in programming.

Last and certainly not least I'd like to thank my wife, Eve, who still provides limitless love, support, and encouragement for whatever I choose to do, and always understands when I can't quite make it to dinner on time.

Introduction

Welcome to *Beginning C: From Novice to Professional, Fourth Edition.* With this book you can become a competent C programmer. In many ways, C is an ideal language with which to learn programming. C is a very compact language, so there isn't a lot of syntax to learn before you can write real applications. In spite of its conciseness and ease, it's also an extremely powerful language that's still widely used by professionals. The power of C is such that it is used for programming at all levels, from device drivers and operating system components to large-scale applications. C compilers are available for virtually every kind of computer, so when you've learned C, you'll be equipped to program in just about any context. Finally, once you know C, you have an excellent base from which you can build an understanding of the object-oriented C++.

My objective in this book is to minimize what I think are the three main hurdles the aspiring programmer must face: coming to grips with the jargon that pervades every programming language, understanding how to *use* the language elements (as opposed to merely knowing what they are), and appreciating how the language is applied in a practical context.

Jargon is an invaluable and virtually indispensable means of communication for the expert professional as well as the competent amateur, so it can't be avoided. My approach is to ensure that you understand the jargon and get comfortable using it in context. In this way, you'll be able to more effectively use the documentation that comes along with most programming products, and also feel comfortable reading and learning from the literature that surrounds most programming languages.

Comprehending the syntax and effects of the language elements is obviously an essential part of learning a language, but appreciating *how* the language features work and *how* they are used is equally important. Rather than just using code fragments, I always provide you with practical working examples that show the relationship of each language feature to specific problems. These examples can then provide a basis for you to experiment and see the effects of changing the code in various ways.

Your understanding of programming in context needs to go beyond the mechanics of applying individual language elements. To help you gain this understanding, I conclude most chapters with a more complex program that applies what you've learned in the chapter. These programs will help you gain the competence and confidence to develop your own applications, and provide you with insight into how you can apply language elements in combination and on a larger scale. Most important, they'll give you an idea of what's involved in designing real programs and managing real code.

It's important to realize a few things that are true for learning any programming language. First, there *is* quite a lot to learn, but this means you'll gain a greater sense of satisfaction when you've mastered it. Second, it's great fun, so you really will enjoy it. Third, you can only learn programming by doing it, and this book helps you along the way. Finally, it's much easier than you think, so you positively *can* do it.

How to Use This Book

Because I believe in the hands-on approach, you'll write your first programs almost immediately. Every chapter has several programs that put a theory into practice, and these examples are key to the book. I advise you to type in and run all the examples that appear in the text because the very act of typing in programs is a tremendous aid to remembering the language elements. You should also attempt all the exercises that appear at the end of each chapter. When you get a program to work for the first time—particularly when you're trying to solve your own problems—you'll find that the great sense of accomplishment and progress make it all worthwhile.

We will start off at a gentle pace, but we'll gain momentum as we get further into the subject. Each chapter will cover quite a lot of ground, so take your time and make sure you understand everything before moving on. Experimenting with the code and trying out your own ideas is an important part of the learning process. Try modifying the programs and see what else you can make them do—that's when it gets really interesting. And don't be afraid to try things out—if you don't understand how something works, just type in a few variations and see what happens. A good approach is to read each chapter through, get an idea of its scope, and then go back and work through all the examples.

You might find some of the end-of-chapter programs quite difficult. Don't worry if it's not all completely clear on the first try. There are bound to be bits that you find difficult to understand at first, because they often apply what you've learned to rather complicated problems. And if you really get stuck, you can skip the end-of-chapter programs, move on to the next chapter, and come back to them later. You can even go through the entire book without worrying about them. The point of these programs is that they're a useful resource for you—even after you've finished the book.

Who This Book Is For

Beginning C: From Novice to Professional, Fourth Edition is designed to teach you how to write useful programs as quickly and easily as possible. This is the tutorial for you if

- You're a newcomer to programming but you want to plunge straight into the C language and learn about programming and writing C programs right from the start.
- You've done a little bit of programming before, so you understand the concepts behind it—maybe you've used BASIC or PASCAL. Now you're keen to learn C and develop your programming skills further.

This book doesn't assume any previous programming knowledge on your part, but it does move quickly from the basics to the real meat of the subject. By the end of *Beginning C*, you'll have a thorough grounding in programming the C language.

What You Need to Use This Book

To use this book, you'll need a computer with a C compiler and library installed so that you can execute the examples, and a program text editor for preparing your source code files. The compiler you use should provide good support for the International Standard for the C language, ISO/IEC 9899. You'll also need an editor for creating and modifying your code. You can use any plain text editor such as Notepad or vi to create your source program files. However, you'll get along better if your editor is designed for editing C code.

To get the most out of this book you need the willingness to learn, the desire to succeed, and the determination to continue when things are unclear and you can't see the way ahead. Almost everyone gets a little lost somewhere along the way when learning programming for the first time. When you find you are struggling to grasp some aspect of C, just keep at it—the fog will surely disperse and you'll wonder why you didn't understand the topic in the first place. You might believe that doing all this is going to be difficult, but I think you'll be surprised by how much you can achieve in a relatively short time. I'll help you to start experimenting on your own and become a successful programmer.

Conventions Used

I use a number of different styles of text and layout in the book to help differentiate between the different kinds of information. For the most part, their meanings will be obvious. Program code will appear like this:

```
int main(void)
{
  printf("\nBeginning C");
  return 0;
}
```

When a code fragment is a modified version of a previous instance, I show the lines that have changed in bold type like this:

```
int main(void)
{
  printf("\nBeginning C by Ivor Horton");
  return 0;
}
```

When code appears in the text, it has a different typestyle that looks like this: `double`.

I'll use different types of "brackets" in the program code. They aren't interchangeable, and their differences are very important. I'll refer to the symbols ( ) as **parentheses**, the symbols { } as **braces**, and the symbols [ ] as **square brackets**.

Important new words in the text are shown in **bold** type.

Code from the Book

All the code from the book and solutions to the exercises are available for download from the Apress web site at `http://www.apress.com`.

CHAPTER 1

■■■

Programming in C

C is a powerful and compact computer language that allows you to write programs that specify exactly what you want your computer to do. You're in charge: you create a program, which is just a set of instructions, and your computer will follow them.

Programming in C isn't difficult, as you're about to find out. I'm going to teach you all the fundamentals of C programming in an enjoyable and easy-to-understand way, and by the end of this chapter you'll have written your first few C programs. It's as easy as that!

In this chapter you'll learn the following:

- How to create C programs
- How C programs are organized
- How to write your own program to display text on the screen

Creating C Programs

There are four fundamental stages, or processes, in the creation of any C program:

- Editing
- Compiling
- Linking
- Executing

You'll soon know all these processes like the back of your hand (you'll be doing them so easily and so often), but first let's consider what each process is and how it contributes to the creation of a C program.

Editing

This is the process of creating and modifying C **source code**—the name given to the program instructions you write. Some C compilers come with a specific editor that can provide a lot of assistance in managing your programs. In fact, an editor often provides a complete environment for writing, managing, developing, and testing your programs. This is sometimes called an **integrated development environment**, or IDE.

You can also use other editors to create your source files, but they must store the code as plain text without any extra formatting data embedded in it. In general, if you have a compiler system with an editor included, it will provide a lot of features that make it easier to write and organize your source programs. There will usually be automatic facilities for laying out the program text appropriately, and color highlighting for important language elements, which not only makes your code more readable but also provides a clear indicator when you make errors when keying in such words.

If you're working in UNIX or Linux, the most common text editor is the vi editor. Alternately you might prefer to use the emacs editor.

On a PC you could use one of the many freeware and shareware programming editors. These will often provide a lot of help in ensuring your code is correct with syntax highlighting and autoindenting of your code. Don't use a word processor such as Microsoft Word, as these aren't suitable for producing program code because of the extra formatting information they store along with the text. Of course, you also have the option of purchasing one of the professionally created programming development environments that support C, such as those from Borland or Microsoft, in which case you will have very extensive editing capabilities. Before parting with your cash though, it's a good idea to check that the level of C that is supported is approximate to the current C standard. With some of the products out there that are primarily aimed at C++ developers, C has been left behind somewhat. A further possibility is to get the emacs editor for Windows. emacs is the editor of choice for some programming professionals.

Compiling

The compiler converts your source code into machine language and detects and reports errors in the compilation process. The input to this stage is the file you produce during your editing, which is usually referred to as a **source file**.

The compiler can detect a wide range of errors that are due to invalid or unrecognized program code, as well as structural errors where, for example, part of a program can never be executed. The output from the compiler is known as **object code** and is stored in files called **object files**, which usually have names with the extension `.obj` in the Microsoft Windows environment, or `.o` in the Linux/UNIX environment. The compiler can detect several different kinds of errors during the translation process, and most of these will prevent the object file from being created.

The result of a successful compilation is a file with the same name as that used for the source file, but with the `.o` or `.obj` extension.

If you're working in UNIX, at the command line, the standard command to compile your C programs will be `cc` (or the GNU's Not UNIX (GNU) compiler, which is `gcc`). You can use it like this:

```
cc -c myprog.c
```

where `myprog.c` is the program you want to compile. Note that if you omit the `-c` flag, your program will automatically be linked as well. The result of a successful compilation will be an object file.

Most C compilers will have a standard compile option, whether it's from the command line (such as `cc myprog.c`) or a menu option from within an IDE (where you'll find a Compile menu option).

Linking

The linker combines the various modules generated by the compiler from source code files, adds required code modules from program libraries supplied as part of C, and welds everything into an executable whole. The linker can also detect and report errors, for example, if part of your program is missing or a nonexistent library component is referenced.

In practice, if your program is of any significant size, it will consist of several separate source code files, which can then be linked. A large program may be difficult to write in one working session, and it may be impossible to work with as a single file. By breaking it up into a number of smaller source files that each provide a coherent part of what the whole program does, you can make the development of the program a whole lot easier. The source files can be compiled separately, which makes eliminating simple typographical errors a bit easier. Furthermore, the whole program can usually be developed incrementally. The set of source files that make up the program will usually be integrated under a **project name**, which is used to refer to the whole program.

Program libraries support and extend the C language by providing routines to carry out operations that aren't part of the language. For example, libraries contain routines that support operations such as performing input and output, calculating a square root, comparing two character strings, or obtaining date and time information.

A failure during the linking phase means that once again you have to go back and edit your source code. Success on the other hand will produce an executable file. In a Microsoft Windows environment, this executable file will have an `.exe` extension; in UNIX, there will be no such extension, but the file will be of an executable type.

Many IDEs also have a Build option, which will compile and link your program in one step. This option will usually be found, within an IDE, in the Compile menu; alternatively, it may have a menu of its own.

Executing

The execution stage is where you run your program, having completed all the previous processes successfully. Unfortunately, this stage can also generate a wide variety of error conditions that can include producing the wrong output or just sitting there and doing nothing, perhaps crashing your computer for good measure. In all cases, it's back to the editing process to check your source code.

Now for the good news: this is the stage where, at last, you get to see your computer doing exactly what you told it to do! In UNIX and Linux you can just enter the name of the file that has been compiled and linked to execute the program. In most IDEs, you'll find an appropriate menu command that allows you to run or execute your compiled program. This Run or Execute option may have a menu of its own, or you may find it under the Compile menu option. In Windows, you can run the `.exe` file for your program as you would any other executable.

The processes of editing, compiling, linking, and executing are essentially the same for developing programs in any environment and with any compiled language. Figure 1-1 summarizes how you would typically pass through processes as you create your own C programs.

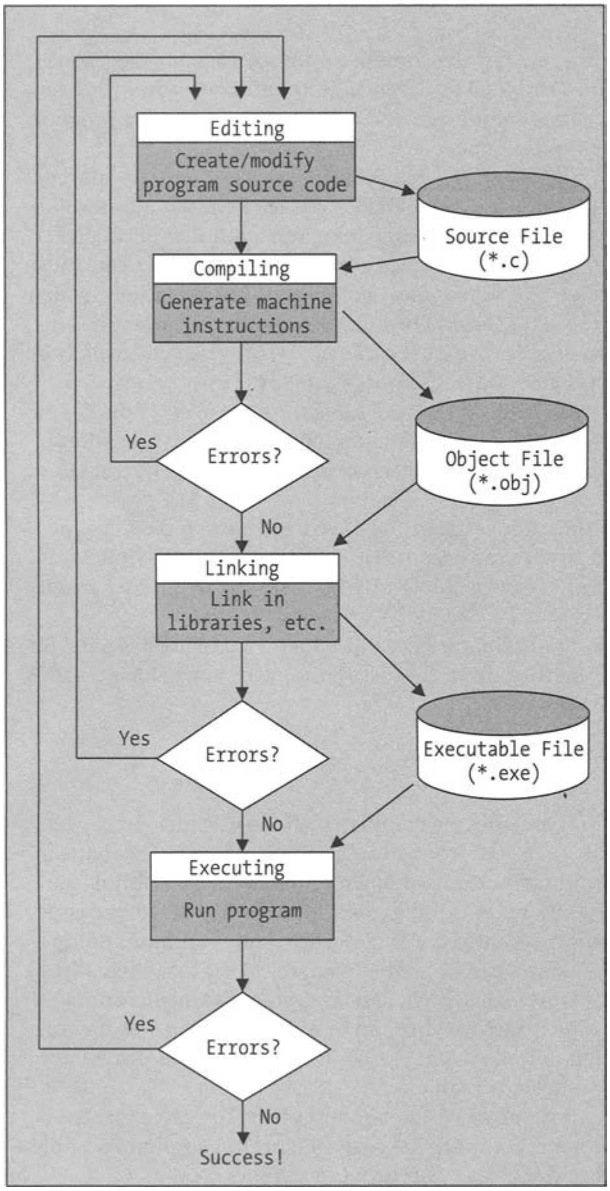

Figure 1-1. *Creating and executing a program*

Creating Your First Program

Let's step through the processes of creating a simple C program, from entering the program itself to executing it. Don't worry if what you type doesn't mean anything to you at this stage—I'll explain everything as we go along.

TRY IT OUT: AN EXAMPLE C PROGRAM

Run your editor, and type in the following program exactly as it's written. Be careful to use the punctuation exactly as you see here. The brackets used on the fourth and last lines are braces—the curly ones {}, not the square ones [] or the round ones ()—it really does matter. Also, make sure you put the slashes the right way (/), as later you'll be using the backslash (\) as well. Don't forget the semicolon (;).

```
/* Program 1.1 Your Very First C Program - Displaying Hello World */
#include <stdio.h>

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

When you've entered the preceding source code, save the program as `hello.c`. You can use whatever name you like instead of `hello`, but the extension must be `.c`. This extension is the common convention when you write C programs and identifies the contents of the file as C source code. Most compilers will expect the source file to have the extension `.c`, and if it doesn't, the compiler may refuse to process it.

Next you'll compile your program as described in the "Compiling" section previously in this chapter and link all the pieces necessary to create an executable program as discussed in the previous "Linking" section. This is typically carried out in a single operation, and once the source code has been compiled successfully, the linker will add code from the standard libraries that your program needs and create the single executable file for your program.

Finally, you can execute your program. Remember that you can do this in several ways. There is the usual method of double-clicking the `.exe` file from Windows Explorer if you're using Windows, but you will be better off opening a command-line window because the window showing the output will disappear when execution is complete. On all platforms, you can run your program from the command line. Just start a command-line session, change the current directory to the one that contains the executable file for your program, and then enter the program name to run it.

If everything worked without producing any error messages, you've done it! This is your first program, and you should see the following message on the screen:

```
Hello world!
```

Editing Your First Program

You could try altering the same program to display something else on the screen. For example, you might want to try editing the program to read like this:

```
/* Program 1.2 Your Second C Program */
#include<stdio.h>

int main(void)
{
  printf("If at first you don\'t succeed, try, try, try again!");
  return 0;
}
```

The \' sequence in the middle of the text to be displayed is called an **escape sequence**. Here it's a special way of including a single quote in the text because single quotes are usually used to indicate where a character constant begins and ends. You'll learn more about escape sequences in the "Control Characters" section later in this chapter. You can try recompiling the program, relinking it, and running it again once you've altered the source. With a following wind and a bit of luck you have now edited your first program. You've written a program using the editor, edited it, and compiled, linked, and executed it.

Dealing with Errors

To err is human, so there's no need to be embarrassed about making mistakes. Fortunately computers don't generally make mistakes themselves and they're actually very good at indicating where we've slipped up. Sooner or later your compiler is going to present you with a list (sometimes a list that's longer than you want) of the mistakes that are in your source code. You'll usually get an indication of the statements that are in error. When this happens, you must return to the editing stage, find out what's wrong with the incorrect code, and fix it.

Keep in mind that one error can result in error messages for subsequent statements that may actually be correct. This usually happens with statements that refer to something that is supposed to be defined by a statement containing an error. Of course, if a statement that defines something has an error, then what was supposed to be defined won't be.

Let's step through what happens when your source code is incorrect by creating an error in your program. Edit your second program example, removing the semicolon (;) at the end of the line with printf() in it, as shown here:

```
/* Program 1.2 Your Second C Program */
#include<stdio.h>

int main(void)
{
  printf("If at first you don\'t succeed, try, try, try again!")
  return 0;
}
```

If you now try to compile this program, you'll see an error message that will vary slightly depending on which compiler you're using. A typical error message is as follows:

```
Syntax error : missing ';' before '}'
HELLO.C - 1 error(s), 0 warning(s)
```

Here, the compiler is able to determine precisely what the error is, and where. There really should be a semicolon at the end of that printf() line. As you start writing your own programs, you'll probably get a lot of errors during compilation that are caused by simple punctuation mistakes. It's so easy to forget a comma or a bracket, or to just press the wrong key. Don't worry about this; a lot of experienced programmers make exactly the same mistakes—even after years of practice.

As I said earlier, just one mistake can sometimes result in a whole stream of abuse from your compiler, as it throws you a multitude of different things that it doesn't like. Don't get put off by the number of errors reported. After you consider the messages carefully, the basic approach is to go back and edit your source code to fix what you can, ignoring the errors that you can't understand. Then have another go at compiling the source file. With luck, you'll get fewer errors the next time around.

To correct your example program, just go back to your editor and reenter the semicolon. Recompile, check for any other errors, and your program is fit to be run again.

Dissecting a Simple Program

Now that you've written and compiled your first program, let's go through another that's very similar and see what the individual lines of code do. Have a look at this program:

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
#include <stdio.h>

int main(void)
{
  printf("Beware the Ides of March!");
  return 0;
}
```

This is virtually identical to your first program. Even so, you could do with the practice, so use your editor to enter this example and see what happens when you compile and run it. If you type it in accurately, compile it, and run it, you should get the following output:

```
Beware the Ides of March!
```

Comments

Look at the first line of code in the preceding example:

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
```

This isn't actually part of the program code, in that it isn't telling the computer to do anything. It's simply a **comment**, and it's there to remind you, or someone else reading your code, what the program does. Anything between /* and */ is treated as a comment. As soon as your compiler finds /* in your source file, it will simply ignore anything that follows (even if the text looks like program code) until it finds the matching */ that marks the end of the comment. This may be on the same line, or it can be several lines further on.

You should try to get into the habit of documenting your programs, using comments as you go along. Your programs will, of course, work without comments, but when you write longer programs you may not remember what they do or how they work. Put in enough comments to ensure that a month from now you (and any other programmer) can understand the aim of the program and how it works.

As I said, comments don't have to be in a line of their own. A comment is everything between /* and */, wherever /* and */ are in your code. Let's add some more comments to the program:

```
/* Program 1.3 Another Simple C Program - Displaying a Quotation */
#include <stdio.h>      /* This is a preprocessor directive    */

int main(void)          /* This identifies the function main() */
{                       /* This marks the beginning of main()  */
  printf("Beware the Ides of March!");  /* This line displays a quotation */
  return 0;             /* This returns control to the operating system */
}                       /* This marks the end of main()            */
```

You can see that using comments can be a very useful way of explaining what's going on in the program. You can place comments wherever you want in your program, and you can use them to explain the general objectives of the code as well as the specifics of how the code works. A single comment can spread over several lines; everything from the /* to the */ will be treated as a comment and ignored by the compiler. Here's how you could use a single comment to identify the author of the code and to assert your copyright:

```
/*
 * Written by Ivor Horton
 * Copyright 2006
 */
```

This is one comment spread over four lines. I have used asterisks to mark the beginning of each line of text here but they are not obligatory, just part of the comment as I wrote it. You can use anything you like to improve the readability of a comment, but don't forget that */ will be interpreted as the end of the comment.

Preprocessing Directives

Look at the following line of code:

```
#include <stdio.h>      /* This is a preprocessor directive    */
```

This is not strictly part of the executable program, but it is essential in this case—in fact, the program won't work without it. The symbol # indicates this is a **preprocessing directive**, which is an instruction to your compiler to do something before compiling the source code. The compiler handles these directives during an initial preprocessing phase before the compilation process starts. There are quite a few preprocessing directives, and they're usually placed at the beginning of the program source file.

In this case, the compiler is instructed to "include" in your program the contents of the file with the name `stdio.h`. This file is called a **header file**, because it's usually included at the head of a program. In this case the header file defines information about some of the functions that are provided by the standard C library but, in general, header files specify information that the compiler uses to integrate any predefined functions or other global objects with a program, so you'll be creating your own header files for use with your programs. In this case, because you're using the `printf()` function from the standard library, you have to include the `stdio.h` header file. This is because `stdio.h` contains the information that the compiler needs to understand what `printf()` means, as well as other functions that deal with input and output. As such, its name, `stdio`, is short for **standard input/output**. All header files in C have file names with the extension `.h`. You'll use other C header files later in the book.

Note Although the header file names are not case sensitive, it's common practice to write them in `#include` directives in lowercase letters.

Every C compiler that conforms to the international standard (ISO/IEC 9899) for the language will have a set of standard header files supplied with it. These header files primarily contain declarations relating to standard library functions that are available with C. Although all C compilers that conform with the standard will support the same set of standard library functions and will have the same set of standard header files available, there may be extra library functions provided with a particular compiler that may not be available with other compilers, and these will typically provide functionality that is specific to the type of computer on which the compiler runs.

Defining the main() Function

The next five statements define the function `main()`:

```
int main(void)          /* This identifies the function main() */
{                       /* This marks the beginning of main()  */
  printf("Beware the Ides of March!");  /* This line displays a quotation */
  return 0;             /* This returns control to the operating system */
}                       /* This marks the end of main()             */
```

A **function** is just a named block of code between braces that carries out some specific set of operations. Every C program consists of one or more functions, and every C program must contain a function called `main()`—the reason being that a program will always start execution from the beginning of this function. So imagine that you've created, compiled, and linked a file called `progname.exe`. When you execute this program, the operating system calls the function `main()` for the program.

The first line of the definition for the function `main()` is as follows:

```
int main(void)          /* This identifies the function main() */
```

This defines the start of the function `main()`. Notice that there is *no* semicolon at the end of the line. The first line identifying this as the function `main()` has the word `int` at the beginning. What appears here defines the type of value to be returned by the function, and the word `int` signifies that the function `main()` returns an integer value. The integer value that is returned when the execution of `main()` ends represents a code that is returned to the operating system that indicates the program state. You end execution of the `main()` function and specify the value to be returned in the statement:

```
  return 0;             /* This returns control to the operating system */
```

This is a **return** statement that ends execution of the `main()` function and returns that value 0 to the operating system. You return a zero value from `main()` to indicate that the program terminated normally; a nonzero value would indicate an abnormal return, which means, in other words, things were not as they should be when the program ended.

The parentheses that immediately follow the name of the function, `main`, enclose a definition of what information is to be transferred to `main()` when it starts executing. In this example, however, you can see that there's the word `void` between the parentheses, and this signifies that no data can be transferred to `main()`. Later, you'll see how data is transferred to `main()` and to other functions in a program.

The function `main()` can call other functions, which in turn may call further functions, and so on. For every function that's called, you have the opportunity to pass some information to it within the parentheses that follow its name. A function will stop execution when a return statement in the body of the function is reached, and control will then transfer to the calling function (or the operating system in the case of the function `main()`).

Keywords

In C, a **keyword** is a word with special significance, so you shouldn't use keywords for any other purposes in your program. For this reason, keywords are also referred to as **reserved words.** In the preceding example, `int` is a keyword and `void` and `return` are also keywords. C has several keywords, and you'll become familiar with more of them as you learn more of the language. You'll find a complete list of C keywords in Appendix C.