C Made Easy

001335

Herbert Schildt

中国科学院微电子中心 藏书

001335

C Made Easy

Osborne **McGraw-Hill**
Berkeley, California

Published by
Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne **McGraw-Hill** at the above address.

C MADE EASY

Copyright © 1985 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

1234567890 DODO 898765

ISBN 0-07-881178-3

Jon Erickson, Acquisitions Editor
Lorraine Aochi, Technical Editor
Raymond Lauzzana, Technical Reviewer
Ted Gartner, Copy Editor
Judy Wohlfrom, Text Design
Deborah Wilson, Composition
Yashi Okita, Cover Design

INTRODUCTION

The purpose of this book is to teach you the C programming language. Because programming is learned best by doing, it is strongly recommended that you have access to a C compiler. There are several available for most microcomputers, including excellent ones for the IBM PC and compatibles. The examples presented in the book will compile correctly and without errors on virtually any C compiler. However, minor variances can occur among different compilers, so it is best to check your user manual first.

This book assumes that you have some knowledge of programming. You should understand the general concepts of variables, assignment statements, and loops. Don't worry, your programming experience need not be extensive.

Because BASIC is generally included with the computer when you purchase it, you are probably acquainted with it. BASIC has become a common language for program examples because it is so widely known. Expecially in the earlier chapters of this book, BASIC and C examples will be used side by side to help you better understand aspects of the C language. Sometimes, seeing a statement written in a language that you are learning is worth more than pages of explanation. However, knowledge of BASIC is not required. Even if you don't know BASIC, this book will still be excellent for learning the C programming language.

If you have not yet purchased a compiler, it is strongly recommended that you buy one that is *UNIX-compatible*, because the function library that comes with it will be similar to the one described in this book.

In the early examples, data will be input to programs using either the **getchar( )** function, which is found with most compilers, or the **getnum( )** function, which is developed in the text as an easy way to input decimal numbers.

The examples in this book were compiled and run using the Aztec C compiler for the IBM PC. The examples will also compile and run using the SuperSoft C compiler, with the exception that the floating-point examples may need to be changed slighty to accommodate differences in the SuperSoft implementation. In general, any version 7, UNIX-compatible compiler will compile and run the programs in this book.

此为试读，需要完整PDF请访问：www.ertongbook.com

CONTENTS

Introducing C

CHAPTER 1

C is often called a middle-level computer language. *Middle-level* does not have a negative meaning: it does not mean that C is less powerful, harder to use, or less developed than a high-level language such as BASIC or Pascal; nor does it mean that C is similar to a low-level language, such as assembly language (often called *assembler*), which is simply a symbolic representation of the actual machine code a computer can read. C is a middle-level language because it combines elements of a high-level language with the functionalism of assembler. Table 1-1 shows the levels of various computer languages, including C.

A middle-level language gives programmers a minimal set of control and data-manipulation statements that they can use to define high-level constructs. In contrast, a high-level language is designed to try to give programmers everything they could possibly want already built into the language. A low-level language forces programmers to define all program functions directly because nothing is built-in. One approach is not inherently better than the other; each has its specific application. Middle-level languages are sometimes thought of as building-block languages because the

Table 1-1. Levels of Computer Languages

High Level	Middle Level	Low Level
Ada	C	Assembler
BASIC	FORTH	
COBOL		
FORTRAN		
Pascal		

programmer first creates the routines to perform all the program's necessary functions and then puts them together.

C allows—indeed needs—the programmer to define routines to perform high-level commands. These routines are called *functions* and are very important in the C language. You can easily tailor a library of C functions to perform tasks that are carried out by your program. In this sense, you can personalize C to fit your needs.

As a middle-level language C manipulates the bits, bytes, and addresses the computer functions with. Unlike BASIC, a high-level language that can operate directly on strings of characters to perform a multitude of string functions, C can operate directly on characters. In BASIC, there are built-in statements to read and write disk files. In C, these procedures are performed by functions that are not part of the C language proper, but are provided in the C standard library. These functions are special routines written in C that perform these operations. For example, the PRINT statement in BASIC has no direct parallel in C. However, there is a function called **printf ()** in your C compiler's standard function library that the manufacturer provided.

C does have its benefits. It has very few statements to remember—only 28 keywords. (The IBM PC version of BASIC has 159.) This means that C compilers can be written reasonably easily, so there is generally one available for your machine. Since C operates on the same data types as the computer, the code output from a C compiler is efficient and fast. C can be used in place of assembler for most tasks.

C code is very portable. *Portability* means software written for one type of computer can be adapted to another type. For example, if a program written for an Apple II+ can be easily moved to an IBM PC, that program is portable. Portability is important if you plan to use a new computer with a different processor. Most application programs will only need to be recompiled with a C compiler written for the new processor. This can save countless hours and dollars.

Uses of C

C was first used for system programming. *System programming* refers to a class of programs that either are part of or work closely with the operating system of the computer. System programs make the computer capable of performing useful work. These are examples of system programs that are often written in C:

- Operating systems
- Assemblers
- Print spoolers
- Modem programs
- Language interpreters
- Language compilers
- Text editors
- Network drivers
- Data bases
- Utilities

There are several reasons why C is used for system programming. System programs often must run very quickly. Programs compiled by C compilers can run almost as fast as those written in assembler. In the past, most system software had to be written in assembly language because none of the available computer languages could create programs that ran fast enough. Writing in assembly language is hard, tedious work. Since C code can be written more quickly than assembly code, using C reduces costs tremendously.

Another reason that C is frequently used for system programming is that it is a programmer's language. Professional programmers seem to be attracted to C because it lacks restrictions and easily manipulates bits, bytes, and addresses. The system programmer needs C's direct control of the I/O and memory management functions. C also allows a program to reflect the personality of the programmer.

Because programmers like to program in C, it has in recent years also been used as a general-purpose programming language. C is very readable. Once you are familiar with C, you can follow the precise flow and logic of a program and easily verify the general operation of subroutines. C program listings look clear; in contrast, a language like BASIC looks cluttered and confusing. Perhaps the best reason that C has become a general-purpose language is that it is simply fun to use.

C as A Structured Language

C is a structured language, as are Ada and Pascal. BASIC, COBOL, and FORTRAN are nonstructured languages. The most distinguishing feature of a structured language is that it uses blocks. A *block* is a set of statements that are logically connected. For example, imagine an IF statement that, if successful, will execute five discrete statements. If these statements can be grouped together and referenced easily, they form a block.

A structured language gives you a variety of programming possibilities. It supports the concept of subroutines with local variables. A *local variable* is simply a variable that is known only to the subroutine in which it is defined. A structured language also supports several loop constructs, such as the **while**, **do-while**, and **for** constructs. (The use of the **goto** is either prohibited or discouraged and is not the common form of program control in the same way it is in BASIC or FORTRAN.) A structured language allows separately compiled subroutines to be used without being in the program proper. This means that you can create a subroutine library of useful, tested functions that can be accessed by any program you write. A structured language allows you to indent statements and does not require a strict field concept as in FORTRAN.

Structured languages tend to be more modern, while the nonstructured are older. In fact, a characteristic of an old computer language is that it is not structured. Because of their clarity, structured languages are not only easier to program in but also much easier to maintain.

Although you may be able to think of nonstructured languages that still satisfy the requirements of a structured language (such as advanced BASICs), a structured language is based on the compartmentalization of function and data: that is, the reduction of each task to its own subroutine or

此为试读，需要完整PDF请访问：www.ertongbook.com

block of code. As you learn the C programming language, the difference between a structured and nonstructured language will become quite clear.

Interpreters Versus Compilers

The terms *interpreter* and *compiler* refer to how a program is executed. In theory any programming language can be either compiled or interpreted, but some languages are usually executed one way or the other. However, the way a program is executed is not defined by the language in which it is written. Interpreters and compilers are simply sophisticated programs that operate on your program source code. *Source code* is the program text that you write.

An interpreter reads the source code of your program one line at a time and performs the specific instructions contained in that line. A compiler reads the entire program and converts it into *object code*, which is a translation of the program source code into a form that can be directly executed by the computer. Object code is also referred to as *binary code* or *machine code*. Once the program is compiled, a line of source code is no longer meaningful to the execution of your program.

For example, BASIC is generally interpreted and C is almost always compiled. An interpreter must be present each time you run your program. In BASIC, you have to execute the BASIC interpreter first, load your program, and then type **RUN** each time you want to use the program. A compiler, on the other hand, converts your program into object code that can be directly executed by the computer. Because the compiler translates the program one time, all you need do in C is execute your program directly, generally by typing its name.

Compiled programs run much faster than interpreted ones. However, the compiling process itself does take more time. But this is easily offset by the time you save while using the program. The only time this is not true is if your program is very short—say, less than 50 lines—and does not use any loops.

In addition to the advantages of speed, compilers protect your source code from theft and unauthorized tampering. Compiled code bears no resemblance to source code, and this is the reason compilers are used almost exclusively by commercial software houses.

Two terms you will see often in this book and in your C compiler manual are *compile time* and *run time.* Compile time refers to the events that occur during the compilation process. Run time refers to the events that occur while the program is actually executing. Unfortunately, you will often see them used in connection with the word *error,* as in *compile-time errors* and *run-time errors.*

General Overview of C

CHAPTER 2

Before learning any specific information about C, you should see what a C program looks like compared to its BASIC equivalent. This chapter will go over C fundamentals; the later chapters will thoroughly explain all aspects of the C programming language.

Figure 2-1 gives the first program beginners usually write in C or BASIC. The program simply prints the word **HELLO** on the computer's screen followed by a carriage return/line feed combination.

Functions in C

The C language is based on the concept of building blocks. The building blocks are called *functions*. A C program is a collection of one or more functions. To write a program, you first create functions and then put them together.

```
main()
{
        printf("HELLO\n");
}
```

```
10 PRINT "HELLO"
20 END
```

Figure 2-1. C version and BASIC version of a program that prints **HELLO**

A function is a subroutine that contains one or more C statements and performs one or more tasks. In well-written C code, each function performs only one task. Each function has a name and a list of arguments that the function will receive. In general, you can give a function whatever name you please, with the exception of **main**, which is reserved for the function that begins program execution.

When denoting functions, this book uses a convention that has become standard when writing about C: a function will have parentheses after the function name. For example, if a function's name is **max**, it would be written as **max()**. This notation will help distinguish variable names from function names.

In the HELLO program of Figure 2-1 both **main()** and **printf()** are functions. As stated earlier, **main()** is the first function executed when your program begins to run. The function **printf()**, not a part of the C language proper, is a subroutine written in C. Subroutines such as this are usually written by the developer of the compiler and are a part of the standard C library. The **printf()** function causes its argument to be printed on the screen of the computer. In the HELLO program, the argument is the string in parentheses, "**HELLO \n**". The **\n** is the symbol C uses to denote a new line; that is, a carriage return/line feed combination.

The General Form of C Functions

The HELLO program introduces the general form of a C function. The program starts with **main()**. Then an opening brace signifies the beginning of the function followed by any statements that make up the function. In this program, the only statement is **printf()**. The closing brace signals the end of a function. Here it also marks the end of the program. The general form of a function is

```
function_name(argument list)
argument_list declaration
      {            opening brace begins the body of the function
                   .
                   .  body of the function
                   .
      }            closing brace ends the function
```

As you can see, the first thing a function needs is the name. Inside the parentheses following the function name is the *list of arguments.* Immediately following on the next line is the *argument list declaration,* which tells the compiler what type of variable to expect. Next, braces surround the body of the function. The *body of the function* is composed of the C statements that define what the function does. C does have an explicit **return** statement that forces a return from a function. Since no explicit **return** is encountered, the function automatically stops execution and returns when it reaches the final brace. This differs from the BASIC **GOSUB-RETURN** combination because BASIC requires the **RETURN** to know when to return from a subroutine.

The main() Function

The **main()** function is special because it is the first function called when your program executes. It signifies the beginning of your program. Unlike a program in BASIC, which begins at the lowest line number or the "top" of the program, a C program begins with a call to the **main()** function. The **main()** function can be anywhere in your program, although it is generally the first function for the sake of clarity. There *must be* a **main()** somewhere in your program so the C compiler can determine where to start execution.

The **main()** function is just like every other C function, except that the closing brace of **main()** signals the end of the program. When this brace is reached, the program exits to the operating system.

There can only be one **main()** in a program. If there were more than one, your program would not know where to begin execution. Most compilers will catch an error like this before you ever reach the execution stage.

Function Arguments

In the HELLO program, the function **printf()** has one argument: the string that will be printed on the computer screen. Functions in C can have from zero to several arguments. (The upper limit is determined by the compiler you are using.) An *argument* is a value that is passed into a function. When a function is defined, variables that will receive argument values must also be declared. These are called the *formal parameters* of the function. For example, the following function will return the product of the two integer arguments. The **return** statement transmits the product back to the calling routine.

```
mul(x,y)   /* mul function */
int x,y;            /* here x and y are declared
                       to be integer variables  */
{
        return(x*y); /* gives the product of the
                          two arguments  */
}
```

Each time **mul()** is called, it will multiply the values of **x** and **y**. Remember, however, that **x** and **y** are simply the function's operational variables that receive the values you assign when calling the function.

Figure 2-2 presents a short program that uses the **mul()** function. This program will print two numbers on the screen: **2** and **2340**. The variables **x**, **y**, **j**, and **k** are not modified by the call to the **mul()** function. In fact, **x** and **y** in **main()** have no relationship to **x** and **y** in **mul()**.

The **mul()** function will multiply the values of both **x** and **y** as well as **j** and **k**. When you call a function, the arguments may be either constants, as in the HELLO program, or variables, as in the **mul()** example. When you created the function **mul()** with two arguments, you declared the argument variables to be **x** and **y**. These are the formal parameters of the function; they hold the information that you pass in when calling the function. C copies the value of the constant or variable used as a function argument into the variable that the function has declared in its list of arguments. Unlike some other computer languages, C does not copy any information back into the function arguments.

In C functions, arguments are always separated by commas. In this book, the term *argument list* will refer to comma-separated arguments. The argument list for **mul()** is **x,y**.

此为试读，需要完整PDF请访问：www.ertongbook.com