Advanced C

Herbert Schildt

Advanced C

Herbert Schildt

Osborne McGraw-Hill
Berkeley, California

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.

Advanced C

1234567890 DODO 898765

ISBN 0-07-881208-9

Jon Erickson, Acquisitions Editor
Lorraine Aochi, Technical Editor
Tim Field, Technical Reviewer
Kevin Shafer, Senior Editor
Lynn Heimbucher, Editorial Assistant
Kay Luthin, Copy Editor
Yashi Okita, Cover Design

INTRODUCTION

I have been fortunate to be able to write the kind of programming book that I have always wanted. Years ago, when I started to program, I tried to find a book that had algorithms for such tasks as sorts, linked lists, simulations, and expression parsers in a straightforward presentation. I wanted a book that would give me insight into programming, but I also wanted a book that I could take off the shelf to find what I needed when I needed it. Unfortunately, I never found the exact book I was looking for—so I decided to write it.

This book explores a wide range of subjects and contains many useful algorithms, functions, and approaches written in the C language. C is the de facto systems programming language, as well as one of the most popular general-purpose professional programming languages available. A wide variety of C compilers is available for virtually all computers, and many are quite inexpensive. I used the Aztec C86 compiler for the IBM PC; however, with only a few exceptions, any version 7, UNIX-compatible C compiler will compile and run all the code in this book.

Chapter 1 begins with a brief history of C and a short review of the language. The sorting of both arrays and disk files is explained in Chapter 2. Chapter 3 deals with stacks, queues, linked lists, and binary trees. (You may

think that's a lot to cover in one chapter; however, the subjects go together nicely and form a solid unit.) Dynamic allocation methods are discussed in Chapter 4. Chapter 5 presents an overview of operating-system interfacing and assembly language linkage. Chapter 6 covers statistics and includes a complete statistics program. Codes, ciphers, and data compression are the topics of Chapter 7, which also includes a short history of cryptography. Chapter 8 details several random number generators and then discusses how to use them in two simulations. The first simulation is a check-out line in a store; the second is a random-walk portfolio management program.

Chapter 9 is my personal favorite because it contains the complete code for a recursive descent parser. Years ago, I would have given just about anything to have had that code! If you need to evaluate expressions, Chapter 9 is for you. Chapters 10 and 11 discuss conversions from other languages, efficiency, porting, and debugging.

H.S.

If you would like to obtain an IBM PC-compatible diskette that contains all of the programs and algorithms in this book, please complete the order form and mail it with payment enclosed. If you are in a hurry, you can call (217) 586-4021 and place your order by telephone.

Please send me ____ copies, at $29.95 each, of the programs in this book. Foreign orders: Please add $5.00 for shipping and handling.

Name

Address

City State ZIP

Telephone

Method of payment: check ____ Visa ____ MC ____

Credit card number: _____

Expiration date: _____

Signature: _____

Send to:

Herbert Schildt
RR 1, Box 130
Mahomet, IL 61853

CONTENTS

A Review of C

CHAPTER 1

This book uses a problem-solving approach to illustrate advanced concepts in the C programming language: it examines common programming tasks and develops solutions with an emphasis on style and structure. Through this approach, various advanced C topics and nuances are covered, as well as the general programming theory behind each solution. You should have a working knowledge of C; however, your experience need not be extensive. A review of the C language is presented later in this chapter.

Two notational conventions are used throughout this book. First, all variable names and C keywords are printed in boldface. Second, all C functions are boldface and are followed immediately by a set of parentheses. These conventions will eliminate confusion between variable names and function names. For example, a variable called "test" is printed as **test**, whereas a function by the same name is printed as **test( )**.

All examples and programs in this book were compiled and run using both the Aztec C compiler and the SuperSoft C compiler for the IBM PC. Generally, any version 7 UNIX-compatible compiler, such as the Lattice or Microsoft compilers, will compile and run the code in this book. There are several compilers available for most computers, and you should have little trouble finding one that suits your needs. Remember, however, that all compilers differ slightly—especially in their libraries—so be sure to read the user manual of the compiler that you are using.

The Origins of C

C was invented and first implemented by Dennis Ritchie on a DEC PDP-11 using the UNIX operating system. C is the result of a development process that started with an older language called BCPL, which is still in use primarily in Europe. BCPL, developed by Martin Richards, influenced a language called B, which was invented by Ken Thompson and led to the development of C.

Although C has seven built-in data types, it is not a strongly typed language in comparison to Pascal or Ada. C allows almost all type conversions, and character and integer types can be intermixed freely in most expressions. No run-time error checking—such as array boundary checking or argument-type compatibility checking—is done. This is the responsibility of the programmer.

C is special in that it allows the direct manipulation of bits, bytes, words, and pointers. This makes it well suited for system-level programming, where these operations are common. Another advantage of C is that it has only 28 *keywords*, which are the commands that make up the C language. For comparison, consider IBM PC BASIC: it has 159 keywords.

Although initially developed to run under the UNIX operating system, C has become so popular that compilers are available for virtually all computers and operating systems. This means that C code is very portable between computers and operating systems, making it possible to write code once and use it anywhere.

C as a Structured Language

C is commonly considered to be a structured language with some similarities to Algol and Pascal. Although the term *block-structured language* does not strictly apply to C in an academic sense, C is informally part of that language group. The distinguishing feature of a block-structured language is the *compartmentalization of code and data.* This means the language can section off and hide from the rest of the program all information and instructions necessary to perform a specific task. Generally, compartmentalization is achieved by subroutines with *local variables,* which are temporary. In this way, it is possible to write subroutines so that the events occurring within them cause no side effects in other parts of the program. Excessive use of *global variables* (variables known throughout the entire program) may allow bugs to creep into a program by allowing unwanted side effects. In C, all subroutines are discrete functions.

Functions are the building blocks of C in which all program activity occurs. They allow specific tasks in a program to be defined and coded separately. After debugging a function that uses only local variables, you can rely on the function to work properly in various situations without creating side effects in other parts of the program. All variables declared in a particular function will be known only to that function.

In C, using blocks of code also creates program structure. A *block of code* is a logically connected group of program statements that can be treated as a unit. It is created by placing lines of code between opening and closing curly braces, as shown here:

```
if(x<10) {
        printf("too low, try again");
        reset_counter(-1);
}
```

In this example the two statements after the if between curly braces are both executed if x is less than 10. These two statements together with the braces represent a block of code. They are linked together: one of the statements cannot execute without the other also executing. In C, every statement can be either a single statement or a block of statements. The use of code blocks creates readable programs with logic that is easy to follow.

C is a programmer's language. Unlike most high-level computer languages, C imposes few restrictions on what you can do with it. By using C a programmer can avoid using assembly code in all but the most demanding situations. In fact, one motive for the invention of C was to provide an alternative to assembly language programming.

Assembly language uses a symbolic representation of the actual binary code that the computer directly executes. Each assembly language operation maps into a single task for the computer to perform. Although assembly language gives programmers the potential for accomplishing tasks with maximum flexibility and efficiency, it is notoriously difficult to work with when developing and debugging a program. Furthermore, since assembly language is unstructured by nature, the final program tends to be "spaghetti code," a tangle of jumps, calls, and indexes. This makes assembly language programs difficult to read, enhance, and maintain.

Initially, C was used for systems programming. A *systems program* is part of a large class of programs that form a portion of the operating system of the computer or its support utilities. For example, the following are commonly called systems programs:

- Operating systems
- Interpreters
- Editors
- Assemblers
- Compilers
- Database managers

As C grew in popularity, many programmers began to use C to program all tasks because of its portability and efficiency. Since there are C compilers for virtually all computers, it is easy to take code written for one machine and then compile and run it with few or no changes on another machine. This portability saves both time and money. C compilers also tend to produce tight, fast object code—smaller and faster than most BASIC compilers, for example.

Perhaps the real reason that C is used in all types of programming tasks is because programmers like it. C has the speed of assembler and the extensibility of FORTH, while having few of the restrictions of Pascal. A C programmer can create and maintain a unique library of functions that have been tailored to his or her own personality. Because C allows—and indeed

encourages—separate compilation, large projects are easily managed.

Many programs in this book use a function called **getnum( )**. C has no built-in method to enter decimal numbers from the keyboard and, contrary to popular belief, the standard library function **scanf( )** is generally unsuitable for human use. Therefore, the special function **getnum( )** is used whenever a decimal number needs to be read from the keyboard. The source code for **getnum( )** is shown here:

```
getnum()    /* read a decimal number from the
               keyboard */
{
        char s[80];

        gets(s);
        return(atoi(s));

}
```

The **atoi( )** function is the standard library function used to convert a string of digits into an integer. If your compiler is supplied with a function similar to **getnum( )**, feel free to substitute it.

A Brief Review

Before you begin to explore various programming problems and solutions, read the rest of this chapter to review the C language. If you are an experienced C programmer, skip to Chapter 2.

Refer to Appendix A for a statement summary of most of the keywords in C, a review of the preprocessor directives, and a description of some of the standard library functions used in this book.

The following 28 keywords, combined with the formal C syntax, form the C programming language:

auto	**double**	**if**	**static**
break	**else**	**int**	**struct**
case	**entry**	**long**	**switch**
char	**extern**	**register**	**typedef**
continue	**float**	**return**	**union**
default	**for**	**short**	**unsigned**
do	**goto**	**sizeof**	**while**

C keywords are always in lowercase letters. In C, uppercase or lowercase makes a difference; that is, else is a keyword, but ELSE is not.

Variables—Types and Declarations

C has seven built-in data types, as shown here:

Data Type	C Keyword Equivalent
character	char
short integer	short int
integer	int
unsigned integer	unsigned int
long integer	long int
floating point	float
double floating point	double

Some implementations of C also support **unsigned long int** and **unsigned short int.**

Variable names are strings of letters from one to several characters long; the maximum length depends on your compiler. For clarity, the underscore may also be used as part of the variable name (for example, **first_time**). Don't forget that in C, uppercase and lowercase are different—**test** and **TEST** will be two different variables.

All variables must be declared prior to use. The general form of the declaration is

type **variable_name;**

For example, to declare **x** to be a floating point, **y** to be an integer, and **ch** to be a character, you would type

```
float x;
int y;
char ch;
```

In addition to the built-in types, you can create combinations of built-in types by using **struct** and **union.** You can also create new names for variable types by using **typedef.**

A *structure* is a collection of variables grouped and referenced under one name. The general form of a structure declaration is

```
struct struct_name {
    element 1;
    element 2;
    .
    .
    .
} struct_variable;
```

As an example, the following structure has two elements: **name**, a character array, and **balance**, a floating-point number.

```
struct client {
        char name[80];
        float balance;
};
```

To reference individual structure elements, the dot operator is used if the structure is global or declared in the function referencing it. The arrow operator is used in all other cases.

When two or more variables share the same memory, a **union** is defined. The general form for a **union** is

```
union union_name {
    element 1;
    element 2;
    .
    .
    .
} union_variable;
```

The elements of a **union** overlay each other. For example, the following declares a **union** t, which in memory looks like Figure 1-1.

```
union tom {
        char ch;
        int x;
} t;
```

The individual variables that comprise the **union** are referenced using the

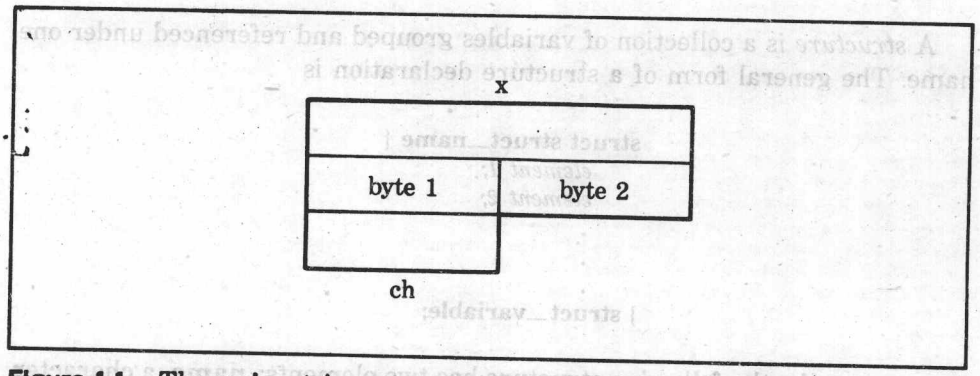

Figure 1-1. The **union** t in memory

dot operator if the **union** is global or is declared in the same function as the reference. The arrow operator is used in all other cases.

The three type modifiers in C—**extern**, **register**, and **static**—can be used to alter the way C treats the variables that follow them. If the **extern** modifier is placed before a variable name, the compiler will know that the variable has been declared elsewhere. The **extern** modifier is commonly used when there are two or more files sharing the same global variables.

The **register** modifier can be used only on local integer or character variables. It causes the compiler to try to keep the value in a register of the CPU instead of placing it in memory. This can make all references to that variable extremely fast. Throughout this book, **register** variables are used for loop control. For example, the following function uses a **register** variable for loop control:

```
f1( )
{
    register int t;
    for(t=0;t<10000;++t) {
        .
        .
        .
    }
}
```

The **static** modifier instructs the C compiler to keep a local variable in existence during the lifetime of the program, instead of creating and destroying it. Remember that the values of local variables are discarded when a function finishes and returns. Using **static** maintains the value of a variable between function calls.

Arrays You may declare arrays on any of the data types discussed earlier. For example, to declare an integer array x of 100 elements, you would write

```
int x[100];
```

This creates an array that is 100 elements long; the first element is 0 and the last is 99. For example, this loop loads the numbers 0 through 99 into array x:

```
for(t=0;t<100; t++) x[t]=t;
```

Multidimensional arrays are declared by placing the additional dimensions inside another pair of brackets. For example, to declare a 10-by-20-integer array, you would write

```
int x[10][20];
```

Operators

C has a rich set of operators that can be divided into classes: *arithmetic, relational and logical, bitwise, pointer, assignment,* and *miscellaneous.*

Arithmetic Operators C has seven arithmetic operators:

Arithmetic Operator	Action
−	subtraction, unary minus
+	addition
*	multiplication
/	division
%	modulo division
−−	decrement
++	increment

The precedence of these operators is

highest ++ −− −(unary minus)
* / %
lowest + −

Operators on the same precedence level are evaluated from left to right.

Relational and Logical Operators Relational and logical operators are used to produce TRUE/FALSE results and are often used together. In C, any nonzero number evaluates TRUE. However, a C relational or logical operator produces the number 1 for TRUE and 0 for FALSE. Here are the relational and logical operators:

Relational Operator	Meaning
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
==	equal
!=	not equal

Logical Operator	Meaning
&&	AND
||	OR
!	NOT

The precedence of these operators is

highest !
> >= < <=
== !=
&&
lowest ||

For example, this expression evaluates as TRUE:

```
(100<200) && 10
```

Bitwise Operators Unlike most other programming languages, C provides bitwise operators that manipulate the actual bits inside a variable. The bitwise operators, listed here, can only be used on integers or characters.

Bitwise Operator	Meaning
&	AND
|	OR
^	XOR
~	one's complement
>>	right shift
<<	left shift