The C Library

Kris Jamsa

The C Library

Kris Jamsa

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.

THE C LIBRARY

1234567890 DODO 898765

ISBN 0-07-881110-4

Jon Erickson, Acquisitions Editor
Raymond Lauzzana, Technical Editor
Fran Haselsteiner, Copy Editor
Deborah Wilson, Composition
Jan Benes, Text Design
Yashi Okita, Cover Design

Introduction

Every day each of us uses a device of some type to make life easier. Man has always been a toolmaker, and the computer may very likely become man's most powerful tool. Whether we are fixing a car or appending one file to another, a tool is something that makes our job easier. The purpose of this text is to provide several tools that you can use in the development of C programs.

Over the past few years, the C programming language has grown a great deal in popularity. Although C was originally developed as a systems programming language, the attributes that make C desirable for systems programs are causing many programmers to realize C's potential in many general-purpose applications. These attributes include portability, modifiability, and access to operations that are normally confined to assembly language programming.

In the past, when we wanted to take a program that was written and running on one type of computer and then move it to a different type of computer, a great deal of the program had to be rewritten. A

major design goal for the implementation of C was to break through this machine dependence. Because of this, C became one of the most portable languages in existence today. A program written in C for one type of computer, such as the IBM PC, will normally run on a second type of computer, such as an APPLE, with little or no modification.

One goal of any programmer is to break a large task into several smaller and more easily implemented tasks. Many programs that at first appear to be large unsolvable tasks can often be broken down into several smaller subtasks that are easier to design and code. Another major advantage of breaking a program into several functions is that the functions created for one program can often be used again in an unrelated application with little or no modification. If we implement most of our program with functions, we increase the readability of the code and decrease the development and testing time of the program. In addition, we create a series of routines that can be placed into a library and shared by other programs. Our goal is to develop functions that perform only one task. In so doing, we will increase the reuseability of our routines.

A Word on UNIX

In the early seventies, Bell Laboratories introduced the UNIX operating system. Because of its tremendous flexibility and the development tools it provides to the user, UNIX has increased in popularity over the past decade. UNIX is well on its way to becoming the industry-wide standard operating system, and because most code used in the UNIX operating system is written in C, the number of applications developed in C and the demand for C programmers will increase for some time to come.

How to Use This Book

This text assumes that you are already familiar with or in the process of learning C. While Chapter 1 provides a brief language overview, it is not intended to be a tutorial on the C programming language. If you are just learning C, many of the routines provided in

this text can be used just as they are written to help you develop powerful programs in minimal time. In addition, by examining the routines along with the documentation provided, you will learn a lot more. If you are an advanced C programmer, many of the routines in this text will introduce you to the concepts employed in creating good programming tools, along with an appreciation of the considerations required to develop utility programs similar to those supported by the UNIX operating system. If you are not currently running under UNIX, you can create an environment similar to UNIX by implementing the routines provided at the end of this text.

The most powerful tool at your disposal is the debug write statement. While the routines provided in this text include detailed explanations, the only way to thoroughly understand a routine is to use it. I strongly recommend that you use debug write statements within each routine to increase your understanding of the processing.

Each chapter of this text will introduce a new topic and build upon concepts previously introduced in the text. Chapter 1 provides a brief overview of the C programming language. It is not intended as a tutorial on C but as a quick reference guide. Chapter 2 introduces constants and macros. The constants and macros provided in this chapter are used by the routines in the later chapters and have been placed in the files **defn.h**, **math.h**, and **strings.h**, which should be included in all programs that access the constants or macros. The content of each of these files is provided in Chapter 2. Chapter 3 provides several string manipulation routines. Chapter 4 examines pointers and their use in string manipulation. Chapter 5 centers upon the user interface and the development of good I/O routines. Chapter 6 presents several array manipulation routines that are developed for the generic **array_type**, which allows each routine to be used for applications requiring arrays of **int**, **float**, or **double**. Chapter 7 examines recursion and how it can be used to simplify difficult programming tasks. Each recursive routine is explained in great detail. Chapter 8 introduces sorting—in particular, the bubble, Shell, and quick sort algorithms. Again, the routines have been developed in a generic manner that allows arrays of type **int**, **float**, or **double**; in addition, a change in the sorting order (ascending or descending) does not require duplicate routines. Chapter 9 provides a series of routines that perform the trigonometric functions and character conversion. In Chapter 10 we will demonstrate the tools developed in Chapters 2 through 9 as we introduce a series of file manipulation routines that are similar to the utilities provided in a

UNIX environment. Chapter 11 introduces the UNIX pipe and how to develop routines to support it.

The additional effort you spend now developing routines that can be shared by other programs will save you many times the time and effort in the future.

Routines included in this book, as well as other useful routines, are available from the author for $29.95, plus $2.50 shipping and handling. The routines are provided on a 5 1/4-inch floppy disk in MS-DOS format. Write to:

Kris Jamsa Software, Inc.
Box 26031
Las Vegas, NV 89126

Contents

	Introduction	vii
Chapter 1	*Language Overview*	1
Chapter 2	*Constants and Macros*	43
Chapter 3	*String Manipulation*	59
Chapter 4	*Pointers*	81
Chapter 5	*Input/Output Routines*	91
Chapter 6	*Array Manipulation Routines*	113
Chapter 7	*Recursion*	127
Chapter 8	*Sorting Routines*	145
Chapter 9	*Trigonometric Functions and Character Conversion*	167
Chapter 10	*File Manipulation Programs*	191
Chapter 11	*Programming the Pipe*	263
Appendix A	*ASCII Codes*	285
	Index	289

CHAPTER

1

Language Overview

This chapter is intended to serve as a quick reference as you examine the routines presented in the remainder of the book. In addition to providing a language overview, this chapter introduces syntax charts, their interpretation, and their use in developing programs in C. Don't worry if you are not familiar with syntax charts: by the end of this chapter you should appreciate the level of information provided in a syntax chart, along with the simplicity of its interpretation.

Pointers and Addresses

Most of the routines provided in this text will utilize *pointers* to memory locations. Many programmers have a difficult time with pointers for several reasons. Firs , some programming languages do

not allow the use of pointers. Second, many programmers avoid using pointers since they do not feel comfortable with their manipulation, which only serves to compound the problem. Third, most texts do not explain pointers adequately. Therefore, before addressing the use of pointers in C, let's take a few steps back and review several concepts that are crucial to understanding pointers.

The basic purpose of pointers is to identify memory locations. Since memory is divided into many locations, each of which is capable of storing information, you need a method of placing a value into a specific location in memory and later retrieving the value. The way this is done is by assigning each memory location a unique address. If, for example, you place a value into the memory location whose address is 1000, you can later retrieve the data since you know where it is located. However, if you had to keep track of memory locations by their actual addresses, your programs would be difficult if not impossible to understand. Instead, programming languages allow you to store data in *variables*. A variable can be viewed as nothing more than a meaningful name you assign to a location or series of locations in memory. A variable, therefore, has two values associated with it. The first is the value you assign to the variable. The second is the address, or memory location, the value is contained in.

A pointer is a variable that contains a memory address. In C you use the symbols & and * when utilizing pointers. The ampersand (&) is used to specify *the address of a variable*, not the value it contains. For example, if you have previously declared the variable **my_data** as type int and the pointer **int_pointer**, you can assign **int_pointer** the address of the variable **my_data** in memory as follows:

```
int my_data;                /* declare the variable */
int *int_pointer;           /* declare the pointer  */
int_pointer = &my_data;     /* assign the address   */
```

Once **int_pointer** is assigned the address of **my_data**, both variables reference the same location memory. The asterisk (*) is used to retrieve *the value contained at the location referenced by a pointer*. The expression ***int_pointer** references the value contained at the address contained in **int_pointer**.

If you have declared the pointer **string** as a pointer to the string "**Computer**", the actual value contained in the variable, **string**, is the address of the first character in the string (C). If you print the value contained in **string**, the address of the letter C in memory will be displayed. If you want to print the actual letter, you must use the

asterisk as follows:

```
putchar(*string);
```

If you want to print the entire string, you can print the character contained at the location contained in the variable **string** and then increment the value in **string** so that it points to the next character as follows:

```
while (*string != '\0')
  putchar(*string++);
```

The postfix expression ***string++** writes the character contained in the memory location referenced by **string**, and then increments **string** to point to the next location in memory.

The best way to understand pointers is to use them. Chapter 4 presents several routines that utilize pointers to character strings. Experiment with these routines and see if you get the results you anticipated. The use of debug write statements within each of these routines is strongly recommended. If you print the address contained in the pointer, along with the value it references, pointer manipulation should become much more understandable.

Syntax Charts

Before you examine the syntax charts, you should become familiar with the following words:

An *expression* is a symbol or series of symbols that expresses a mathematical operation. The following are valid expressions in C:

```
a = b + 1
a                    is the same as the expression a = a + 0
                     or a = a - 0
a++ <= 17
```

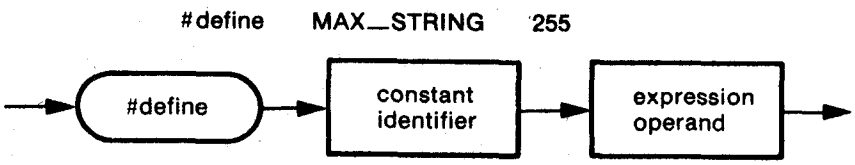
An *identifier* is a unique name that identifies an object. A variable name is an example of an identifier.

A *literal* is a constant value. In the syntax charts later in this chapter, literals are normally the reserved words and punctuation that appear in C.

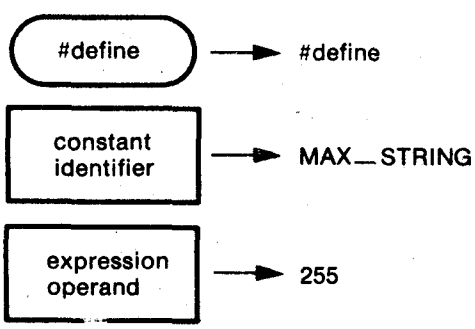
Syntax charts are read from left to right, in the direction of the arrows that connect the symbols contained in the chart. Since syntax charts are composed of symbols, you can normally understand the syntax charts of a language although you may have never programmed in it. Many experienced computer scientists use syntax charts rather than trying to memorize the syntax for the constructs they don't use on a day-to-day basis. Unfortunately, most of us are never properly introduced to syntax charts. Once you understand the flow of a syntax chart, it provides a very useful tool.

The symbols in Table 1-1 are used in C syntax charts.

Consider the following example:



If you start with the first item in the syntax chart and follow the direction of the arrows, the syntax chart provides the following information:

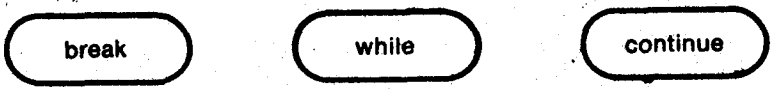


The first two symbols in the chart are fairly straightforward. The third symbol, **expression operand**, requires additional explanation. The rectangle surrounding the words **expression operand** informs you that **expression operand** is defined in terms of another

Table 1-1. Symbols Used in Syntax Charts

The oval symbol contains a literal value that is placed within a program just as it is written.

Examples:

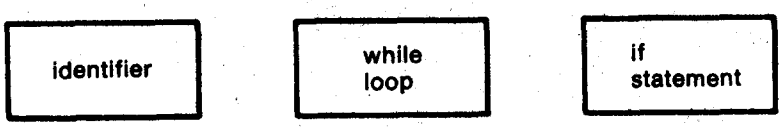


The circle symbol contains a literal value normally used for special symbols and punctuation.

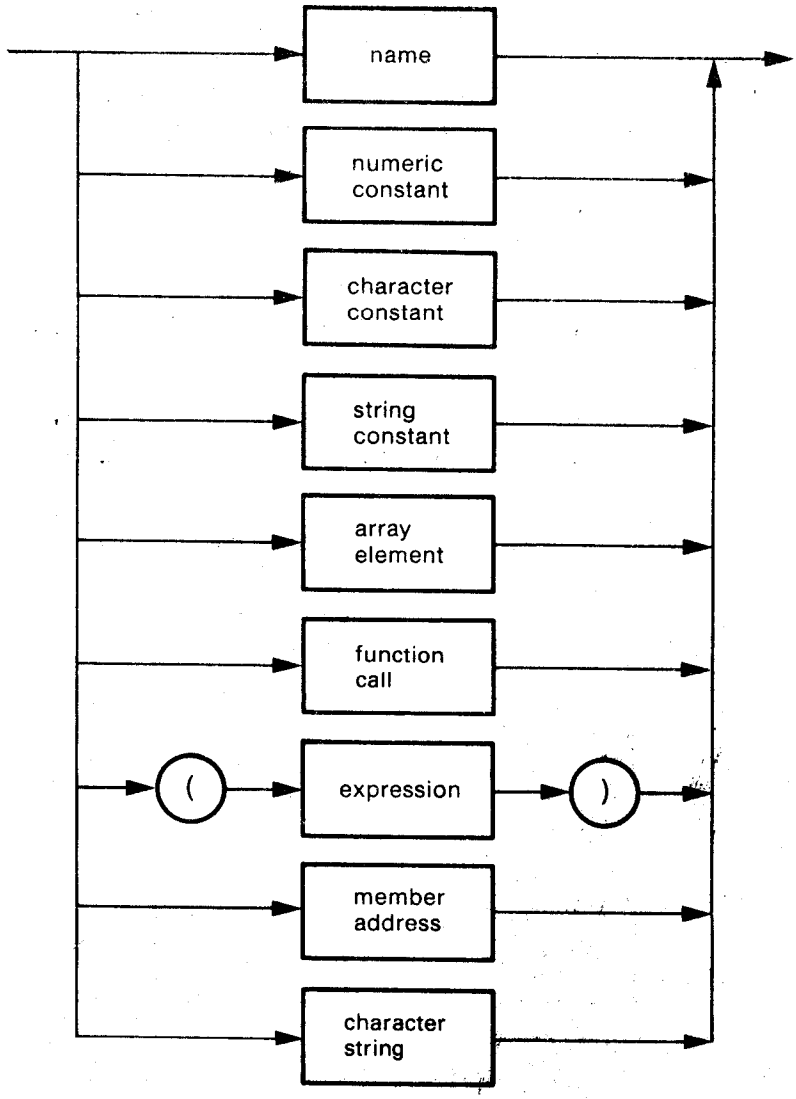
Examples:

The rectangle symbol contains a construct defined by another syntax chart.

Examples:



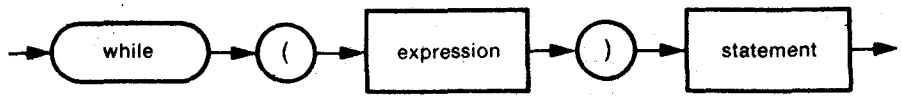
syntax chart. If you examine the syntax chart for an expression operand, you will find that it can be any one of the following:



In this case the **expression operand is the numeric constant** 255. Consider the following while loop:

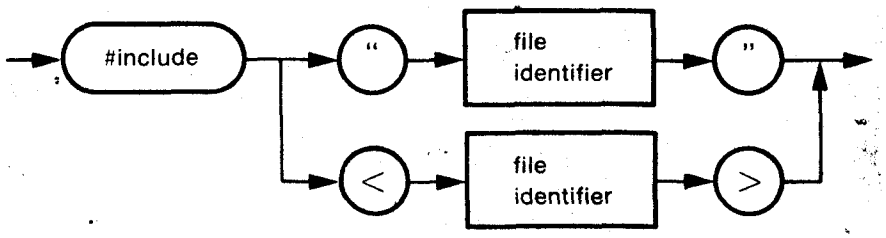
```
while (i <= 10)
  i++;
```

If you examine the syntax chart for the while loop,



the literals **while**, (, and) are again straightforward. The expression in this case is **i** <= 10 and the statement is **i++;**.

If the syntax chart contains more than one possible path, you must select the appropriate path and continue to follow the direction of the arrows. For example, if you want to include the file **stdio.h**, the syntax chart for the **#include** statement provides the following information:



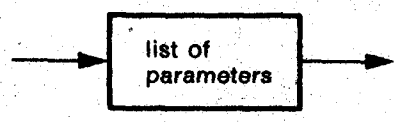
If you already know how the operating system treats files contained with quotes and brackets <>, the syntax chart will provide the correct syntax; otherwise you must examine the reference guide that accompanied your compiler.

The syntax chart is only meant as a guide to the correct syntax of instructions. If you have never worked with syntax charts before, they may at first be intimidating, but keep in mind that the charts are meant to be a tool. Compare the various constructs provided in C (while, do-while, for, if, and so on) to their representations in syntax charts, and you should begin to understand the flow of the charts. If you don't understand a particular chart (a do-while loop, for example), try implementing one in C and then comparing your implementation to the chart. It is important that you become familiar with syntax charts because their popularity is continuing to grow each day.

The following syntax charts illustrate the syntax associated with the C programming language*. Programming examples follow most of the diagrams. Several of the charts are intended for advanced C programmers.

*Brian W. Kernigan and Dennis Ritchie, *The C Programming Language*, Prentice Hall, Inc., Englewood Cliffs, N.J., 1978, Appendix A.

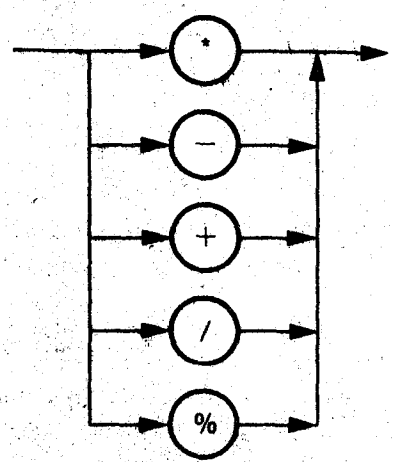
Actual Parameters



Examples:

a, b, c
c

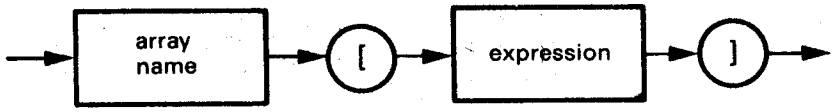
Arithmetic Operator



Examples:

```
x = 3 * 5 - 2;
y = 7/2;
remainder = 3% 2; /* assigns remainder */
```

Array Element



Examples:

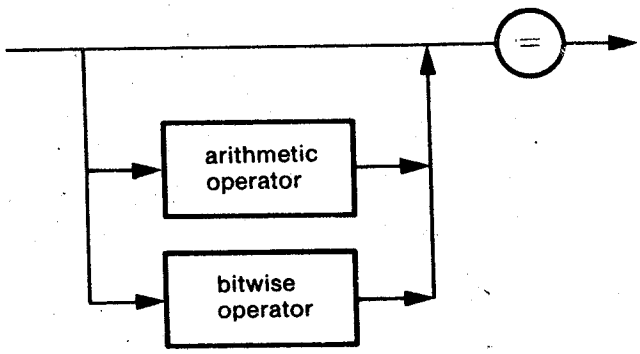
```
string [1];
argv [2];
argv [argc−1];
```

Array Name

Example:

```
string
```

Assignment Operator



Examples:

```
x = 5;
x += 5;    /* x = x + 5 */
x >>=2;    /* x = x >>2 */
```