

# **TopSpeed® C**

**For IBM® Personal Computers and Compatibles**

## **Language Tutorial**

**TopSpeed Corporation**

Copyright© 1990-1991, by TopSpeed Corporation. All rights reserved.

TopSpeed® is a registered trademark of TopSpeed Corporation.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Printed in the United Kingdom.

10 9 8 7 6 5 4 3 2 1

# Contents

<b>CHAPTER 1 .....</b>	<b>8</b>
Introducing C .....	8
History of C .....	8
Advantages of C .....	8
Disadvantages of C .....	9
Tips for Effective Use of C .....	10
<b>CHAPTER 2 .....</b>	<b>12</b>
C Fundamentals .....	12
The main Function .....	12
Comments .....	13
Variables and Assignment .....	14
Identifiers .....	15
Defining Constants .....	17
The printf Function .....	19
Conversion Specifications .....	20
The scanf Function .....	22
Compound Statements .....	24
Layout of a C Program .....	25
<b>CHAPTER 3 .....</b>	<b>28</b>
Operators and Expressions .....	28
Arithmetic Operators .....	28
Operator Precedence and Associativity .....	29
Relational Operators .....	30
Equality Operators .....	30
Logical Operators .....	31
Increment and Decrement Operators .....	32
Assignment Operators .....	33
The Conditional Operator .....	35
Summary of C Operators .....	35

Expression Evaluation .....	38
Expression Statements .....	39
<b>CHAPTER 4 .....</b>	<b>41</b>
Basic Types .....	41
Integer Types .....	41
Floating Types .....	43
Character Types .....	45
Type Conversion .....	47
<b>CHAPTER 5 .....</b>	<b>52</b>
Statements .....	52
The if Statement .....	52
The else Clause .....	54
The switch Statement .....	55
The while Statement .....	57
The do Statement .....	58
The for Statement .....	60
The break Statement .....	63
The continue Statement .....	63
The goto Statement .....	64
The Null Statement .....	65
Unintentional Null Statements .....	65
<b>CHAPTER 6 .....</b>	<b>67</b>
Arrays .....	67
One-Dimensional Arrays .....	67
Multidimensional Arrays .....	70
Array Initialization .....	71
<b>CHAPTER 7 .....</b>	<b>73</b>
Functions .....	73
Function Definitions .....	73
The switch Statement .....	73
The while Statement .....	75
The do Statement .....	76

The for Statement .....	78
The break Statement .....	81
The continue Statement .....	82
The goto Statement .....	82
The Null Statement .....	83
Unintentional Null Statements .....	83
<b>CHAPTER 8 .....</b>	<b>85</b>
Pointers .....	85
Pointer Variables .....	85
The Address and Indirection Operators .....	86
Pointer Assignment .....	87
Null Pointers .....	88
Using Pointers as Function Parameters .....	88
Pointer Arithmetic .....	91
Using an Array Name as a Pointer .....	93
Using an Array Name as a Parameter .....	94
<b>CHAPTER 9 .....</b>	<b>96</b>
Strings .....	96
String Literals .....	96
String Variables .....	98
Reading and Writing Strings .....	100
Accessing the Characters in a String .....	101
Using the C String Library .....	104
Writing String Functions .....	108
<b>CHAPTER 10 .....</b>	<b>111</b>
The Preprocessor .....	111
Preprocessing Directives .....	111
Macro Definition .....	112
File Inclusion .....	119
Conditional Compilation .....	119
The Overall Structure of a C Program .....	122
<b>CHAPTER 11 .....</b>	<b>125</b>

Declarations .....	125
Declarations versus Definitions .....	125
Storage Classes .....	126
Type Qualifiers .....	130
Declarators .....	131
Initializers .....	132
Blocks .....	132
Scope Rules .....	133
<b>CHAPTER 12 .....</b>	<b>135</b>
Structures, Unions, and Enumerations .....	135
Declaring Structures .....	135
Operations on Structures .....	137
Arrays of Structures .....	138
Type Definitions .....	143
Unions .....	145
Enumerations .....	146
Adding a “Tag Field” to a Union .....	147
<b>CHAPTER 13 .....</b>	<b>149</b>
Advanced Uses of Pointers .....	149
Dynamic Storage Allocation .....	149
Linked Lists .....	151
Pointers to Functions .....	159
<b>CHAPTER 14 .....</b>	<b>163</b>
File I/O .....	163
Streams .....	163
Opening and Closing Files .....	164
Text I/O .....	166
Binary I/O .....	170
Random Access .....	171
<b>CHAPTER 15 .....</b>	<b>173</b>
Low-Level Programming .....	173
Bit-Fields in Structures .....	177

<b>APPENDIX A .....</b>	<b>178</b>
Differences between ANSI C and K&R C .....	178
<b>APPENDIX B .....</b>	<b>181</b>
The Standard Library .....	181
Bibliography .....	185
<b>Index .....</b>	<b>186</b>

# CHAPTER 1

## Introducing C

Before we become immersed in the details of C, let's first take a moment to review the history of C and discuss the advantages and disadvantages of the language. We'll also offer a few tips on how to use C most effectively.

### History of C

---

C was designed and implemented by Dennis Ritchie of Bell Laboratories around 1972. C was created to serve as the implementation language for the UNIX operating system.

C's ancestry dates back to Algol 60 (1960), one of the earliest (and most influential) programming languages. The language CPL (1963), a derivative of Algol 60, gave rise to BCPL (1967). BCPL was the basis for B (1970), which, like C, was developed at Bell Labs. B was the immediate ancestor of C.

*The C Programming Language*, a book by Brian Kernighan and Dennis Ritchie, has been the bible of C programmers since its publication in 1978. In the absence of an official standard, this book—known as K&R to aficionados—was the definitive reference for C.

C continued to change after 1978, however, with new features being added and a few older features deleted. With C's growing popularity in the 1980s came the need for a more thorough and precise description of the language. Without such a standard, numerous dialects would arise, threatening the portability of C programs, a major strength of the language.

The development of a U.S. standard for C began in 1983 under the auspices of the American National Standards Institute (ANSI). After many revisions, the standard was completed in 1990. The second edition of K&R, published in 1988, reflects the changes made in the ANSI standard.

The description of C in this tutorial is based on the ANSI standard. Appendix A lists the major differences between ANSI C and original K&R C.

### Advantages of C

---

During the 1980s, C experienced a tremendous growth in popularity. Some of this growth was a result of UNIX systems becoming more common, but a



large part was due to the recognition of C's advantages as a programming language. These advantages include:

- *Efficiency.* Efficiency has been one of C's goals from the beginning, since it was intended for applications where assembly language had traditionally been used. C supports efficiency by avoiding features that can't be translated into efficient machine code.
- *Portability.* One of the reasons for not writing UNIX in assembly language (the traditional choice for operating systems) was that it wouldn't be portable to different computers. C's features support portability, although the language doesn't prevent programmers from writing nonportable programs. C's wide availability also aids portability. C compilers have always been distributed with UNIX; in addition, many C compilers are now available for non-UNIX environments. The ANSI standard for C also helps ensure that programs are portable.
- *Power.* C's large collection of statements, data types, and operators makes it a powerful language.
- *Flexibility.* C imposes very few restrictions on the use of its features; operations that would be illegal in other languages are often permitted in C. For example, C allows a character to be added to an integer value (or, for that matter, a floating-point number). This flexibility makes programming easier.
- *Standard library.* One of C's great strengths is its standard library, which contains a rich assortment of functions for input/output, string handling, storage allocation, and other useful operations.
- *Integration with UNIX.* C is particularly powerful in combination with UNIX. In fact, a knowledge of C is required to use some UNIX tools.

## Disadvantages of C

---

Like every programming language, C has disadvantages as well as advantages:

- *C programs can be error-prone.* C's flexibility makes it an error-prone language. Errors that would be caught in many other languages can't be detected by a C compiler. To make matters worse, C contains a number of pitfalls for the unwary.
- *C's features can be difficult to understand.* Although C is a small language by most measures, it has a number of features that aren't found in other common languages (and consequently are often misunderstood).

- *C programs can be difficult to modify.* C was designed at a time when interactive communication with computers was tedious at best. As a result, C was purposefully kept terse to minimize the time required to enter and edit programs. Unfortunately, C's terseness can make programs difficult to read and therefore difficult to modify. C's flexibility can also be a negative factor; careless use of C's powerful features can make programs almost impossible to understand.

## Tips for Effective Use of C

---

Effective use of C requires taking advantage of C's strengths while minimizing its disadvantages. Here are a few suggestions:

- *Learn how to avoid C pitfalls.* Hints for avoiding pitfalls are scattered throughout this tutorial, enclosed in boxes and preceded by the word *Warning*. For a more extensive list, see Koenig's *C Traps and Pitfalls*. (The TopSpeed C compiler can detect some of the most common pitfalls and issue a warning.)
- *Use debugging tools.* UNIX systems traditionally provide a program named *lint* that will check a C program for possible errors. If *lint* (or a similar program) is available, it's a good idea to use it. (Although TopSpeed C doesn't include a *lint* program, the compiler incorporates some of the features of *lint*.) Another useful tool is a debugger. Because of the nature of C, many bugs can't be detected by a C compiler; these show up in the form of run-time errors or incorrect output. As such, a good debugger, such as JPI's Visual Interactive Debugger, is practically mandatory when programming in C.
- *Use existing code libraries.* A good way to reduce errors (and save programming effort) is to purchase existing libraries of C code. TopSpeed C includes libraries for windowing, graphics, and process scheduling. C libraries for other common tasks including communications, database management, networking, and screen design are readily available from a number of vendors.
- *Adopt a sensible set of coding conventions.* A coding convention is a rule not enforced by the language that makes programs more uniform, easier to read, and easier to modify. The programming examples in this book follow one set of conventions, but there are other, equally valid, conventions in use. Which set you use is less important than adopting *some* conventions and sticking to them.
- *Avoid "tricks" and overly complex code.* C encourages programming tricks. For example, there are usually several ways to accomplish a given task; programmers are often tempted to choose the method that uses the fewest lines or symbols.

Unfortunately, the most concise solution is often the hardest for someone else to understand. In this tutorial, I've tried to find solutions that are reasonably concise yet understandable; I recommend that readers do the same.

# CHAPTER 2

## C Fundamentals

This chapter introduces C by means of example programs and some of C's basic concepts and terminology, with emphasis on the `scanf` and `printf` functions, which perform input and output.

### The main Function

---

The simplest C programs have the form

```
#include <stdio.h>

main()
{
    statements
}
```

(In the template above and in the rest of this tutorial, items printed in Courier would appear in a C program exactly as shown; items in *italics* represent text to be supplied by the programmer.)

The purpose of the line

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

is to inform the compiler that the program will need access to the standard I/O library. (`<stdio.h>` is a *header* containing information about this library.) Input/output isn't supported by the C language proper, so programs that perform I/O normally contain this line.

The rest of the program is a definition of `main`, which is a *function*. Between the `{` and `}` symbols go *statements*—commands to be performed (*executed*) when the program is run. In general, a C program may contain many functions. One of these must be named `main` to indicate where the program begins execution. For now, we'll assume that programs consist of only the `main` function (we'll remove this restriction in Chapter 7).

### Example Program: Printing a Welcome Message

The following simple program consists of just the `main` function:

```
#include <stdio.h>

main()
{
    printf("Welcome to TopSpeed from ");
    printf("JPI\n");
}
```

When run, this program displays the following message on the screen:

```
Welcome to TopSpeed from JPI
```

The program uses `printf`, a function in the standard I/O library, to write two *strings* (sequences of characters). The first string is “Welcome to TopSpeed C from” and the second is “JPI\n”. The use of a function (like `printf`) to perform an action is said to be a *call* of the function.

`printf` doesn’t automatically advance to the next output line when it finishes printing. To instruct `printf` to advance one line, we must include the symbol `\n` (the *newline character*) in the string to be printed. Writing a newline character terminates the current output line; subsequent output goes onto the next line. In our program, the first string didn’t contain a newline character, so the second string was printed on the same line as the first.

As this example illustrates, each statement in a C program must end with a semicolon. (The exception is the compound statement, which we’ll discuss later in this chapter.)

## Comments

---

Programmers often need to include explanatory remarks in a program to identify the name of the program, the date written, the author, the purpose of the program, the algorithm used, and so forth. These remarks are called *comments*. In C, the symbol `/*` marks the beginning of a comment and the symbol `*/` marks the end:

```
/* This is a comment */
```

Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text. The compiler ignores comments; as it processes a program, it replaces each comment by a single space character.

Comments may extend over more than one line; once it has seen the `/*` symbol, the compiler reads (and ignores) whatever follows until it encounters the `*/` symbol.

```
/* This comment starts on one line,  
but ends on another. */
```

Comments may not be nested. Thus, the following code is illegal:

```
/*  
/* not legal */  
*/
```

The `*/` symbol on the second line matches the `/*` symbol on the first line, so the compiler will flag the `*/` symbol on the third line as an error.

Being able to nest comments would be useful during debugging, when we may want to temporarily disable a portion of a program (which may include comments, of course). However, we can use the C preprocessor to get the same effect (see Chapter 10).

Warning:

Forgetting to terminate a comment will cause the compiler to ignore part of your program. Consider the following example:

```
printf("First");
/* forgot to close this comment...
printf("Second");
printf("Third"); /* so it ends here */
printf("Fourth");
```

Because we've neglected to terminate the first comment, this example prints FirstFourth; the compiler has ignored the middle two lines.

## Variables and Assignment

---

Most programs contain *variables*: objects whose values can vary during program execution. Variables must be *declared* in the following way before they can be used:

```
int height, length, width, volume;
```

This *declaration* states that height, length, width, and volume are variables of type int (short for *integer*), meaning that each of these four variables is capable of storing a single whole number. Like statements, declarations are always followed by a semicolon.

Our first template for main didn't include declarations. When main contains declarations, these must precede the statements in main:

```
#include <stdio.h>

main()
{
    declarations
    statements
}
```

A variable can be given a value by means of *assignment*. For example, the statements

```
height = 5;
length = 20;
width = 10;
```

assign the values 5, 20, and 10 to height, length, and width, respectively.

Once a variable has been given a value, it may appear on the right side of an assignment:

This statement multiplies height, length, and width, then assigns the resulting value to the variable volume. Notice that the item to the right of the =

symbol need not be a number; in general, it may be an *expression*. An expression is a sequence of symbols that can be evaluated to produce a single value. An expression can be simple (a single variable or number, for example) or it may contain a number of *operators* (like \*, the multiplication operator) and *operands* (like height, length, and width).

### **Example Program: Computing the Volume of a Box**

The following program computes the volume of a box from its height, length, and width:

```
/* Illustrates the use of variables,
   expressions, and assignments */

#include <stdio.h>

main()
{
    int height, length, width, volume;

    height = 5;
    length = 20;
    width = 10;
    volume = height * length * width;

    printf("The height is %d\n", height);
    printf("The length is %d\n", length);
    printf("The width is %d\n", width);
    printf("The volume is %d\n", volume);
}
```

This program consists of one declaration followed by eight statements (four assignments and four calls of printf).

The %d in each call of printf indicates that an integer value is to be filled in during printing. For example, the line

```
printf("The height is %d\n", height);
```

is a command to print the message The height is *n*, where *n* is the current value of the variable height.

The output of this program is

```
The height is 5
The length is 20
The width is 10
The volume is 1000
```

## **Identifiers**

---

The names of variables, functions, and other entities in a program are called *identifiers*. In C, identifiers may contain letters, digits, and underscores; an identifier must begin with a letter or underscore. Here are some examples of legal identifiers:

```
times10  get_next_char  _done
```

The following are not legal identifiers:

```
10times  get-next-char
```

The symbol `10times` begins with a digit, not a letter or underscore. `get-next-char` contains minus signs, not underscores.

C is *case-sensitive*: it distinguishes between upper-case and lower-case letters in identifiers. For example, the following identifiers are all different:

```
eof  eoF  eOf  e0F  Eof  EoF  EOf  EOF
```

Many C programmers follow the convention of using only lower-case letters in identifiers, with underscores inserted when necessary for legibility:

```
symbol_table  current_page  name_and_address
```

Other programmers avoid underscores, instead using an upper-case letter to begin each word in an identifier:

```
SymbolTable  CurrentPage  NameAndAddress
```

Other reasonable conventions exist. Just be sure to capitalize an identifier the same way each time it appears in a program.

ANSI C places no limit on the maximum length of an identifier.

## **Keywords**

In ANSI C, the following *keywords* have special significance to the compiler and therefore can't be used as identifiers:

<code>auto</code>	<code>double</code>	<code>int</code>	<code>struct</code>
<code>break</code>	<code>else</code>	<code>long</code>	<code>switch</code>
<code>case</code>	<code>enum</code>	<code>register</code>	<code>typedef</code>
<code>char</code>	<code>extern</code>	<code>return</code>	<code>union</code>
<code>const</code>	<code>float</code>	<code>short</code>	<code>unsigned</code>
<code>continue</code>	<code>for</code>	<code>signed</code>	<code>void</code>
<code>default</code>	<code>goto</code>	<code>sizeof</code>	<code>volatile</code>
<code>do</code>	<code>if</code>	<code>static</code>	<code>while</code>

Because of C's case sensitivity, keywords must appear in programs exactly as shown above, with all letters in lower case. Names of functions in the standard library (such as `printf`) contain only lower-case letters also. Avoid the plight of the unfortunate programmer who enters an entire program in upper case, only to find that the compiler can't recognize keywords and calls of library functions.



## Defining Constants

---

When a program contains constants—values that don’t change during program execution—it’s usually a good idea to give them names. For example, a program that performs calculations on Fahrenheit temperatures might use 32 and 212 frequently, since these numbers represent the freezing and boiling points of water. C provides a convenient mechanism, *macro definition*, for naming a constant. The following lines illustrate the use of macro definition to create names for 32 and 212:

```
#define FREEZING_PT 32
#define BOILING_PT 212
```

The first line defines the name `FREEZING_PT` to represent the number 32. Wherever `FREEZING_PT` appears later in the program, it’s understood to be the number 32. Similarly, `BOILING_PT` represents 212.

These lines are actually commands (*directives*) to the C *preprocessor*. When a program is compiled, the preprocessor edits the program according to directives such as the ones shown above. As a result of our two macro definitions, the preprocessor will replace each occurrence of `FREEZING_PT` by 32 and each occurrence of `BOILING_PT` by 212. (Chapter 10 discusses the behavior of the preprocessor in more detail.)

A macro may be defined in terms of an expression:

```
#define SCALE_FACTOR (5.0 / 9.0)
```

When it contains operators, the expression should be enclosed in parentheses. (Chapter 10 explains why.)

Notice that we’ve used only upper-case letters in names of constants. This is a convention that most C programmers follow, not a requirement of the language.

### **Example Program: Converting from Fahrenheit to Celsius**

The following program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature.

```

/* Illustrates macro definition,
   floating-point numbers, and scanf */

#include <stdio.h>

#define FREEZING_PT 32
#define SCALE_FACTOR (5.0 / 9.0)
main()
{
    float fahrenheit, celsius;

    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);

    celsius =
        (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

    printf(
        "Celsius equivalent is: %.1f\n", celsius);
}

```

The output of the program has the following appearance:

```

Enter Fahrenheit temperature: 212
Celsius equivalent is: 100.0

```

The line

```
float fahrenheit, celsius;
```

declares `fahrenheit` and `celsius` to be variables of type `float` (short for “floating point”). Unlike `int` values, `float` values may have digits after the decimal point.

The following line calls the `scanf` function:

```
scanf("%f", &fahrenheit);
```

`scanf` is used to read input. The `f` in `scanf`, like the `f` in `printf`, stands for “formatted”; both `scanf` and `printf` require the use of a *format string* to specify the appearance of the input or output data. In this case, the format string is “%f”. The `%f` indicates that the call of `scanf` should look for an input value in floating-point format (in other words, the number may contain a decimal point). Once it has read the number, `scanf` stores it in the variable `fahrenheit`. (The `&` symbol ensures that `scanf` is supplied with the *address* of `fahrenheit` instead of its *value*. Chapter 8 explains this distinction.)

The following call of `printf` writes the Celsius temperature:

```
printf(
    "Celsius equivalent is: %.1f\n", celsius);
```

The `%.1f` in the format string specifies that the value of `celsius` is to be printed with one digit after the decimal point. (Incidentally, `%d` can’t be used here since `celsius` is of type `float`, not `int`.) During printing, `printf` will substitute the value of `celsius` for `%.1f`.

## The printf Function

---

printf and scanf are two of the most frequently used functions in C. Before continuing further into C, let's take a closer look at these functions, starting with printf.

When it is called, printf must be supplied with a format string, followed by a list of values to print. The format string may contain both ordinary characters and *conversion specifications*, which begin with the % symbol. A conversion specification indicates how a value is to be converted from its internal form (binary) to character form.

Ordinary characters in a format string are printed exactly as they appear in the string; conversion specifications are replaced by the values to be printed. Consider the following example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892;
y = 5527.0;

printf(
    "i = %d, j = %d, x = %f, y = %f\n",
    i, j, x, y);
```

This call of printf produces the following output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

The ordinary characters in the format string ('i', 'j', 'x', 'y', '=', ',', and the space character) were simply copied to the output line. The four conversion specifications were replaced by the values of the variables i, j, x, and y, in that order.

### **Warning:**

**C compilers don't check that the number of conversion specifications in a format string matches the number of output items. Furthermore, there's no check that each conversion specification is appropriate for the type of item being printed. If the programmer uses an incorrect specification, the program will simply produce meaningless output.**

Consider the following (incorrect) use of printf:

```
printf("%d %d %f %f\n", i, x, j, y);
```

The value of x will be printed in integer form. Since x is a float variable, however, the integer that is printed won't reflect the true value of x. Similarly, the value of j, an integer variable, will be printed as a meaningless floating-point number.

## Conversion Specifications

---

A conversion specification has the form

`%m.pX`

For example, in the conversion specification `%10.2f`, the value of *m* is 10, the value of *p* is 2, and *X* is *f*. Either *m* or *p* (or both) may be omitted; if *p* is missing, the period is also omitted. In the specification `%10f`, the value of *m* is 10 and *p* is missing, but in the specification `%.2f`, the value of *p* is 2 and *m* is missing.

The *minimum field width*, *m*, specifies the minimum number of characters to print. If the value to be printed requires more than *m* characters, the field width automatically expands to the necessary size. If the value to be printed requires fewer than *m* characters, the value is right-justified within the field. (In other words, extra spaces precede the value.) To cause left justification, we can put a minus sign in front of *m*.

The meaning of the *precision*, *p*, depends on the choice of *X*.

The *conversion specifier*, *X*, indicates which conversion should be applied to the value before it is printed. The most common conversion specifiers for numbers are:

- *d*ÆDisplays an integer in decimal (base 10) form. The value of *p* indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary).
- *f*ÆDisplays a floating-point number in fixed decimal format. The value of *p* indicates how many digits should appear after the decimal point (the default is 6).
- *e*ÆDisplays a floating-point number in exponential format (scientific notation). *p* has the same meaning as for the *f* specifier.
- *g*ÆDisplays a floating-point number in either fixed decimal format or exponential format, whichever is more compact. The value of *p* indicates the maximum number of significant digits to be displayed. Unlike the *f* conversion, the *g* conversion won't show trailing zeros. Furthermore, if the value to be printed has no digits after the decimal point, then *g* doesn't display the decimal point.

Chapter 4 lists additional conversion specifiers for numbers and characters; Chapter 9 covers conversion specifiers for strings. The *Library Reference* discusses conversion specifications in complete detail.

### **Example Program: Using printf to Print Numbers**

The following program illustrates the use of printf to print integers and floating-point numbers in various formats:

```
#include <stdio.h>

main()
{
    int i;
    float f;

    i = 40;
    f = 839.21;

    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
    printf("|%10.3f|%10.3e|%-10g|\n", f, f, f);
}
```

The output of this program is:

```
|40|    40|40    |   040|
|  839.210| 8.392e+02|839.21    |
```

Let's look at what each conversion specification does:

- `%d`—Displays `i` in decimal form, using a minimum amount of space.
- `%5d`—Displays `i` in decimal form, using a minimum of five characters. Since `i` requires only two characters, three spaces were added.
- `%-5d`—Displays `i` in decimal form, using a minimum of five characters; if the value of `i` doesn't require five characters, the spaces are added afterward (that is, `i` is left justified in a field of length five).
- `%5.3d`—Displays `i` in decimal form, using a minimum of five characters overall and a minimum of three digits. Since `i` is only two digits long, an extra zero was added to guarantee three digits. The resulting number is only three characters long, so two spaces were then added, for a total of five characters.
- `%10.3f`—Displays `f` in fixed decimal form, using ten characters overall, with three digits after the decimal point. Since `f` requires only seven characters (three before the decimal point, three after the decimal point, and one for the decimal point itself), three spaces precede `f`.
- `%10.3e`—Displays `f` in exponential form, using ten characters overall, with three digits after the decimal point. `f` requires nine characters altogether (including the exponent), so one space precedes `f`.
- `%-10g`—Displays `f` in either fixed decimal form or exponential form, whichever is more concise, using ten characters overall.

In this case, fixed decimal form is shorter (six characters as opposed to nine). The presence of the minus sign forces left justification, so `f` is followed by four spaces. Notice the suppression of the trailing zero.

## The scanf Function

---

Just as `printf` prints output in a specified format, `scanf` reads input data according to a particular format.

`printf` and `scanf` are closely related. In particular, both require a format string (although the two functions interpret the format string in slightly different ways). In a call of `scanf`, the format string is followed by the names of variables into which input is to be stored; each variable is normally preceded by the symbol `&`.

A `scanf` format string, like a `printf` format string, may contain both ordinary characters and conversion specifications. Let's first consider the simple case that the format string contains only conversion specifications, as in the following example:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

Suppose that the user enters the following input line:

```
10 20 43.2892 5527.0
```

`scanf` will store the values 10, 20, 43.2892, and 5527.0 into the variables `i`, `j`, `x`, and `y`, respectively. The numbers need not be on the same line. As a rule, `scanf` ignores *white-space* characters (the space, horizontal and vertical tab, form-feed, and newline characters) when searching for a number, allowing numbers to be spread out over a single line or several lines.

The conversions allowed with `scanf` are essentially the same as those used with `printf`. When used with `scanf`, however, the `e`, `f`, and `g` specifiers are identical. Each will read a floating-point number, with or without an exponent.

Let's take a closer look at how `scanf` processes a format string. If `scanf` encounters a conversion specification, it tries to locate an item of the appropriate type in the input, skipping white-space characters if necessary. `scanf` then reads the item, stopping when it encounters a character that can't belong to the item. (For example, when asked to read an integer in decimal format, `scanf` first searches for a digit, plus sign, or minus sign; it then reads digits until it reaches a nondigit.) This character is "put back" to be read again.

A format string may contain ordinary characters in addition to conversion specifications. The action that `scanf` takes when it reaches an ordinary character depends on whether or not it is a white-space character. (1) When it encounters one or more consecutive white-space characters in a format string, `scanf` repeatedly reads white-space characters from the input until reaching a non-white-space character (which is “put back”). The number of white-space characters in the format string is irrelevant; one white-space character in the format string will match any number of white-space characters in the input. (2) When it encounters a non-white-space character in a format string, `scanf` compares it with the next input character. If the two characters match, `scanf` discards the input character and continues processing the format string. If the characters don’t match, `scanf` puts the input character back, then terminates without reading further.

For example, suppose that the format string is “%d/%d”. If the input is “5/89”, `scanf` skips the first space, matches %d with 5, matches / with /, skips a space, and matches %d with 89. On the other hand, if the input is “5 /89”, `scanf` skips one space, matches %d with 5, then attempts to match the / in the format string with a space in the input. There is no match, so `scanf` puts the space back; the characters “ /89” remain to be read by the next call of `scanf`. To allow spaces after the first number, we should use the format string “%d /%d” instead.

Since `scanf` normally skips white-space characters when looking for data items, there’s often no need for a format string to include characters other than conversion specifications. Incorrectly assuming that `scanf` format strings should resemble `printf` format strings is a common error of beginning C programmers. It may cause `scanf` to behave in unexpected ways. Let’s see what happens when the following call of `scanf` is executed:

```
scanf("%d, %d, %f, %f\n", &i, &j, &x, &y);
```

`scanf` will first look for an integer value in the input, which it stores in the variable `i`. `scanf` will then try to match a comma with the next input character. If the next input character is a space, not a comma, `scanf` will terminate prematurely without reading values for `j`, `x`, and `y`.

**Warning:** Putting the newline character at the end of a format string won’t necessarily cause `scanf` to advance to the beginning of the next input line. To `scanf`, a newline character in a format string is equivalent to a space; both cause `scanf` to advance to the next non-white-space character. For example, if the format string is “%d\n”, `scanf` will skip white space, read an integer, then skip to the next non-white-space character (which may or may not be at the beginning of the next input line).

`scanf`, like `printf`, contains several traps for the unwary. When using `scanf`, the programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable—as with `printf`, the compiler doesn’t check for a possible mismatch. Furthermore, the programmer is responsible for ensuring that each variable is preceded by the `&` symbol.

**Warning:** Forgetting to put the & symbol in front of a variable in a call of scanf will have unpredictable and possibly disastrous results. A system crash is a common outcome. At the very least, the value that is read from the input won't be stored in the variable; instead, the variable will retain its old value (which may be meaningless if the variable was not given an initial value). Omitting the & is an extremely common error. Be careful! TopSpeed C can detect this error in some cases. If the variable that is missing the & (i, say) hasn't yet been assigned a value, TopSpeed C will issue the warning *"Possible use of 'i' before definition."* If you get this warning, check for a missing &.

## Compound Statements

---

A compound statement has the following form:

```
{
    statements
}
```

Compound statements are used whenever the syntax of C requires a single statement, but we want to include more than one statement.

*Note:* Compound statements, unlike all other statements, aren't followed by a semicolon.

### Example Program: Printing a Table of Squares

The following program prints a table of squares.

```
/* Illustrates while statements and compound statements */

#include <stdio.h>

main()
{
    int i, n;

    printf("Print a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    i = 1;
    while (i <= n) {
        printf("%10d%10d\n", i, i*i);
        i = i + 1;
    }
}
```

The program prompts the user to enter a number  $n$ ; it then prints  $n$  lines of output, with each line containing a number between 1 and  $n$  together with its square:



Print a table of squares.

Enter number of entries in table: 5

1	1
2	4
3	9
4	16
5	25

This program introduces the while statement, which has the general form

`while ( expression ) statement`

When a while statement is executed, the expression in parentheses is evaluated; if the expression is true, then the statement after the parentheses (the *loop body*) is executed and the expression is checked again. The process continues until the expression is false. C requires that the body of a while statement be a *single* statement. If we need to put two or more statements in the body, they must be enclosed in braces to form a compound statement. In our example program, the loop body is the compound statement

```
{
    printf("%10d%10d\n", i, i*i);
    i = i + 1;
}
```

This statement, in turn, consists of two other statements (a call of printf and an assignment).

## Layout of a C Program

---

C places few restrictions on the layout of a program. For the most part, the programmer can insert or delete white-space characters, including spaces and newline characters. For example, we could delete most of the space in the “table of squares” program:

```
/*Illustrates while statements and compound statements*/

#include<stdio.h>
main(){int i,n;
printf("Print a table of squares.\n");
printf("Enter number of entries: ");
scanf("%d",&n);i=1;
while(i<=n){
printf("%10d%10d\n",i,i*i);
i=i+1;}}
```

White-space characters in a C program aren’t significant (and can therefore be deleted) with the following exceptions:

- Space can’t be deleted when this would cause two symbols to merge into one. In the “table of squares” program, we must leave space between `int` and `i`; without the space, the compiler

would treat `inti` as a single symbol.

- Spaces inside strings are significant; they can't be deleted without changing the meaning of the string.
- Preprocessor directives/commands that begin with the `#` symbol must be on separate lines.

Extra white-space characters can be inserted as well as deleted. The following rules apply:

- Space can be added only between symbols, not inside a symbol. For example, the line

```
scanf("%d", &n);
couldn't be written
scanf("%d", &n);
although it could be written
scanf ( "%d"
                                , & n ) ;
```

One consequence of this rule is that a symbol can't be divided over two lines by inserting a newline character in the middle of the symbol.

- Extra space can't be inserted in a string without changing the meaning of the string. It is possible to split a string over two lines, however, as we'll see in Chapter 9.

Judicious use of extra spaces and blank lines can make a program easier to read and understand. Extra spaces make it easier for the eye to recognize the symbols in a line of program text. For example, in this tutorial we'll often leave an extra space before and after each operator; we'll write

```
i = i + 1;
```

instead of

```
i=i+1;
```

Extra blank lines can divide a program into logical units, making it easier for the reader to discern the program's structure.

Another important issue in program layout is the use of indentation to indicate nesting. For example, in the "table of squares" program, the while statement contains two other statements (a call of `printf` and an assignment) nested inside it. To make this nesting easily visible to someone reading the program, we've indented the inner statements two spaces:

```
while (i <= n) {
    printf("%10d%10d\n", i, i*i);
    i = i + 1;
}
```

Furthermore, we've put the { symbol on the same line as the word while and put the closing } on a separate line, aligned with the word while. Putting the } symbol on a separate line enables easy insertion or deletion of inner statements; aligning it with while makes it easy to spot the end of the while statement.

Some programmers prefer to format while statements (and similar nested constructs) in other ways. For example, some prefer to place the { symbol on a line by itself:

```
while (i <= n)
{
    printf("%10d%10d\n", i, i*i);
    i = i + 1;
}
```

Also, some prefer to indent the inner statements three or more spaces instead of two. Which style to use is mainly a matter of taste. What is most important is to use some consistent style of indentation to highlight the structure of the program.

# CHAPTER 3

## Operators and Expressions

One of C's distinguishing characteristics is its rich collection of operators. In this chapter, we discuss many (though not all) of C's operators and show how they're used to build expressions. Some operators (the arithmetic, relational, equality, and logical operators, and the simple assignment operator) are similar to operators in other programming languages. Others (the increment and decrement operators, the compound assignment operators, and the conditional operator) are rarely seen outside C.

In addition to describing individual operators, this chapter also covers the issues of operator precedence and associativity, which come into play when an expression contains more than one operator. Also included is a discussion of how C expressions are evaluated; in some cases, which we'll show how to avoid, the value of an expression may depend on which C compiler you use.

The chapter concludes with an introduction to the *expression statement*, an unusual feature that allows any expression to serve as a statement.

### Arithmetic Operators

---

C's arithmetic operators can be divided into three classes:

#### *Unary operators:*

- + unary plus
- unary minus

#### *Additive operators:*

- + addition
- subtraction

#### *Multiplicative operators:*

- \* multiplication
- / division
- % remainder

The unary operators require one operand; the other operators require two. (Operators that require two operands are called *binary* operators.)

The % operator requires integer operands; all other arithmetic operators allow either integer or floating-point operands (with mixing allowed).

/ operator

The / and % operators deserve special mention. When both of its operands are integers, / truncates the result by dropping the fractional part. For example, the value of 5 / 2 is 2. The value of i % j is the remainder when i is divided by j. For example, the value of 5 % 2 is 1, and the value of 4 % 5 is 4.

When / and % are used with negative operands, the result depends on the implementation. For example, the value of -5 / 2 or 5 / -2 could be either  $\mathbb{E}3$  or  $\mathbb{E}2$  (in TopSpeed C, the result is  $\mathbb{E}2$ ). When i or j is negative, the sign of i % j depends on the implementation. (In TopSpeed C, the sign of i % j is the same as the sign of the left operand; for example, the value of -5 % 2 is  $\mathbb{E}1$ , while the value of 5 % -2 is 1.)

Notice that C has no exponentiation operator. Exponentiation can be done by calling the pow function in the standard library (see Appendix B).

## Operator Precedence and Associativity

---

When an expression contains more than one operator, the meaning of the expression may not be immediately clear. (Does  $i + j * k$  mean  $(i + j) * k$  or  $i + (j * k)$ ?) In C, this potential ambiguity is resolved by *operator precedence*. The arithmetic operators have the following relative precedence:

```
highest: + - (unary)
* / %
lowest: + - (binary)
```

When two or more operators appear in the same expression, we can determine the meaning of the expression by repeatedly putting parentheses around subexpressions, starting with high-precedence operators and working down to low-precedence operators. The following examples illustrate the result:

```
i + j * k    means  i + (j * k)
-i * -j      means  (-i) * (-j)
+i + j / k   means  (+i) + (j / k)
```

Operator precedence rules alone aren't enough when an expression contains two or more operators at the same level of precedence. In this situation, the *associativity* of the operators helps determine the meaning of the expression. The binary arithmetic operators are all *left associative* (that is, they group from left to right); the unary operators are *right associative*. The following examples illustrate associativity:

```
i * j / k    means  (i * j) / k
i - j * i + k means  (i - (j * i)) + k
- - i        means  -(-i)
```

## Relational Operators

---

C provides four *relational operators*:

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to

The precedence of the relational operators is lower than that of the arithmetic operators; for example,  $i + j < k - 1$  means  $(i + j) < (k - 1)$ . The relational operators are left associative.

The relational operators can be used to compare integers and floating-point numbers, with operands of mixed types allowed.

The value of a relational expression is either 1 (if the condition is true) or 0 (if it's false). For example,  $10 < 11$  has the value 1, while  $10 > 11$  has the value 0. C doesn't have names for the Boolean values *true* and *false*; instead, the number 1 (or any nonzero value, for that matter) represents *true*, and the number 0 represents *false*.

Warning: The expression

$i < j < k$

is legal in C, but doesn't have the meaning that you might expect. The < operator is left associative, so this expression is equivalent to;

$(i < j) < k$

In other words, the expression first tests whether *i* is less than *j*; the 1 or 0 produced by this comparison is then compared to *k*. The expression does *not* test whether *j* lies between *i* and *k*.

## Equality Operators

---

The *equality operators* are

- == equal to
- != not equal to

Like the relational operators, the equality operators are left associative and produce either 0 (*false*) or 1 (*true*) as their result.

The fact that the relational and equality operators produce integer values can sometimes be used to advantage. For example, the expression  $(i >= j) + (i == j)$  performs a three-way test on the relative values of the variables *i* and *j*; the value of the expression is either 0, 1, or 2, depending on whether *i* is less than, greater than, or equal to *j*.

In most programming languages, the equality operators are grouped with the relational operators. In C, however, the equality operators have somewhat different properties than the relational operators. In particular, the precedence of the equality operators is lower than that of the relational operators. For example, the expression

```
i < j == j < k
```

is equivalent to

```
(i < j) == (j < k)
```

which is true if both  $i < j$  and  $j < k$  are true or if both are false.

## Logical Operators

---

The *logical operators* are

```
! logical negation
&& logical and
|| logical or
```

The `!` operator is unary, while `&&` and `||` are binary.

The logical operators produce either 0 or 1 as their result. Often, the operands will also be 0 or 1, but this isn't a requirement; the logical operators treat any nonzero operand as a true value and any zero operand as a false value.

The logical operators behave as follows:

- `!expr` has the value 1 if `expr` has the value 0.
- `expr1 && expr2` has the value 1 if `expr1` and `expr2` are both nonzero.
- `expr1 || expr2` has the value 1 if either `expr1` or `expr2` (or both) is nonzero.

In all other cases, these operators produce the value 0.

Both `&&` and `||` perform “short-circuit” evaluation of their operands. That is, these operators evaluate first the left operand, then the right operand. If the value of the expression can be deduced from the value of the left operand alone, then the right operand isn't evaluated. Consider the following expression:

```
(i != 0) && (j / i > 0)
```

To find the value of this expression, we must first evaluate `(i != 0)`. If `i` isn't equal to 0, we then evaluate `(j / i > 0)` to determine whether the entire expression is true or false. However, if `i` is equal to 0, then the entire expression must be false, so there's no need to evaluate `(j / i > 0)`. The

advantage of short-circuit evaluation is apparent. Without it, evaluation of the expression would have caused a division by zero.

The `!` operator has the same precedence as the unary plus and minus operators. The precedence of `&&` and `||` is lower than that of the relational and equality operators; for example, `i < j && k == m` means `(i < j) && (k == m)`. The `!` operator is right associative; `&&` and `||` are left associative.

## Increment and Decrement Operators

---

The *increment* and *decrement operators*, written `++` and `--`, although used frequently in C programs, are rarely provided by other languages. The `++` operator increments (adds 1 to) a variable; the `--` operator decrements (subtracts 1 from) a variable.

The `++` operator can be used in either *prefix* or *postfix* form; that is, it may either precede its operand (`++i`, for example) or follow it (`i++`). The same is true of the `--` operator. The postfix versions of `++` and `--` have higher precedence than unary plus and minus and are left associative. The prefix versions have the same precedence as unary plus and minus and are right associative.

The `++` and `--` operators are unusual in that both modify the values of their operands. Operators that modify their operands are said to have *side effects*. Evaluating the expression `++i` produces the result `i + 1` and, as a side effect, increments `i`. Evaluating the expression `i++` produces the result `i`, but causes `i` to be incremented afterwards. In either case, `i` is permanently increased by 1 as a result of evaluating the expression.

In other words, when used as a *prefix* operator, `++` increments its operand *before* its value is fetched:

```
i = 1;
printf("i is %d\n", ++i); /* prints "i is 2" */
```

When used as a *postfix* operator, `++` increments its operand *after* its value is fetched:

```
i = 1;
printf("i is %d\n", i++); /* prints "i is 1" */
printf("i is %d\n", i);   /* prints "i is 2" */
```

The `--` operator has similar properties:

```
i = 1;
printf("i is %d\n", --i); /* prints "i is 0" */

i = 1;
printf("i is %d\n", i--); /* prints "i is 1" */
printf("i is %d\n", i);   /* prints "i is 0" */
```



When `++` or `--` is used more than once in the same expression, the result can be sometimes be tricky to understand. For example, consider the following statements:

```
i = 1;
j = 2;
k = ++i + j++;
```

What are the values of `i`, `j`, and `k` after these statements are executed? Since `i` is incremented *before* its value is used, but `j` is incremented *after* it is used, the last statement is equivalent to

```
i = i + 1;
k = i + j;
j = j + 1;
```

so the final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.

## Lvalues

The increment and decrement operators require an *lvalue* (pronounced “Lvalue”) as their operand. An lvalue is an expression that represents a storage location in computer memory. Variables are lvalues, since a variable is just a name for a storage location. Expressions other than variables are usually not lvalues, however. For example, the expression `i + j` isn’t an lvalue, so it’s not legal to write `++(i + j)`, `(i + j)++`, `--(i + j)`, or `(i + j)--`.

## Assignment Operators

---

We’ve encountered the *simple assignment operator* `=` in previous examples. Simple assignment is only one of many assignment operators in C; the other assignment operators perform *compound assignment*.

### Simple Assignment

The effect of a simple assignment `v = e` is to evaluate the expression `e` and copy its value into `v`, which must be an lvalue. `e` can be a constant, a variable, or a more complicated expression:

```
i = 5;           /* i is now 5 */
j = i;          /* j is now 5 */
k = 10 * i + j;  /* k is now 55 */
```

If `v` and `e` don’t have the same type, then the value of `e` is converted to the type of `v` before assignment:

```
int i;
float f;

i = 72.99;      /* i is now 72 */
f = 136;        /* f is now 136.0 */
```

Chapter 4 discusses type conversion in more detail.

In C, assignment is a true operator. It produces a value that can be used by other operators (the value of  $v = e$  is the value of  $v$  after the assignment). As a result, assignments can be chained together when several variables are to be assigned the same value:

```
i = j = k = 0;
```

The `=` operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```

The effect is to assign 0 first to `k`, then to `j`, and finally to `i`.

Any expression may contain the assignment operator:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k);
/* prints "1 1 2" */
```

Using the assignment operator in this fashion usually isn't a good idea, however, as we'll see later in the chapter.

## **Compound Assignment**

C provides ten compound assignment operators:

```
*= /= %= += -= <<= >>= &= ^= |=
```

The operator `*=` is a combination of the `*` and `=` operators; the expression  $v *= e$  is equivalent to  $v = v * e$ . (Actually, there's a minor difference between the two expressions:  $v$  is evaluated only once in  $v *= e$ , but twice in  $v = v * e$ . Chapter 6 contains an example that illustrates this difference.) The other operators have similar meanings.

Here are several examples of compound assignment:

```
i = 5;
j = 2;
i *= j; /* i is now 10 */
```

```
i = 5;
j = 2;
i /= j; /* i is now 2 */
```

```
i = 5;
j = 2;
i %= j; /* i is now 1 */
```

```
i = 5;
j = 2;
i += j; /* i is now 7 */
```

```
i = 5;
j = 2;
i -= j; /* i is now 3 */
```

We'll discuss the operators `<<=`, `>>=`, `&=`, `^=`, and `|=` in Chapter 15.

The compound assignment operators have the same properties as the `=` operator; in particular, they're right associative and can be used in arbitrary expressions.

## The Conditional Operator

---

A *conditional expression* produces one of two possible values, depending on the outcome of a test. Conditional expressions have the form

```
expr1 ? expr2 : expr3
```

where *expr1*, *expr2*, and *expr3* are expressions. (The symbols `?` and `:` are considered to be a single operator—the *conditional operator*—not a combination of two different operators. The conditional operator is unique among C operators in that it requires *three* operands instead of one or two.)

```
operands: use of
```

A conditional expression is evaluated in stages: *expr1* is evaluated first; if its value isn't zero, then *expr2* is evaluated, and its value is the value of the entire expression. If the value of *expr1* is zero, then the value of *expr3* is the value of the expression.

The precedence of the conditional operator is less than that of the other operators we've discussed so far, with the exception of the assignment operators.

The following example illustrates the use of the conditional operator:

```
int i, j, k;

i = 1;
j = 2;
k = i > j ? i : j;      /* k is now 2 */
k = (i >= 0 ? i : 0) + j; /* k is now 3 */
```

Conditional expressions tend to make programs shorter but harder to understand. It's probably best to limit the use of conditional expressions to certain situations in which they're particularly useful. (See the discussion of macros in Chapter 10 for an example.)

## Summary of C Operators

---

On the following page is a complete table of C operators, showing the precedence and associativity of each operator. (The highest precedence is 1; the lowest is 15.) We've seen many of these operators already; the remaining ones are covered in later chapters.

Operators of precedence 1 are postfix operators. Operators of precedence 2 (the *unary* operators) are prefix operators, as is the cast operator (precedence 3). All other operators (except the conditional operator) are binary operators.

Using this table, we can fill in the missing parentheses in any C expression. For example, consider the following expression:

```
a = b = c < d++ && e != f ? g + h / -i : j >> k
```

To understand this expression, we need to add parentheses to show how the expression is constructed from subexpressions. Adding parentheses to an expression is easy: After examining the expression to find the operator with highest precedence, we put parentheses around the operator and its operands. We then repeat the process (ignoring the subexpression that we've just parenthesized) until the expression is fully parenthesized. In our example, the operator with highest precedence is ++ (used here as a postfix operator), so we put parentheses around ++ and its operand:

```
a = b = c < (d++) &&
e != f ? g + h / -i : j >> k
```

Repeating the process yields the following sequence of partially parenthesized expressions:

Unary minus (precedence 2):

```
a = b = c < (d++) &&
e != f ? g + h / (-i) : j >> k
```

Division (precedence 4):

```
a = b = c < (d++) &&
e != f ? g + (h / (-i)) : j >> k
```

Addition (precedence 5):

```
a = b = c < (d++) &&
e != f ? (g + (h / (-i))) : j >> k
```

Bitwise shift (precedence 6):

```
a = b = c < (d++) &&
e != f ? (g + (h / (-i))) : (j >> k)
```

Relational (precedence 7):

```
a = b = (c < (d++)) &&
e != f ? (g + (h / (-i))) : (j >> k)
```

Equality (precedence 8):

```
a = b = (c < (d++)) &&
(e != f) ? (g + (h / (-i))) : (j >> k)
```

Logical *and* (precedence 12):

```
a = b = ((c < (d++)) && (e != f)) ? (g + (h / (-i))) : (j >> k)
```

Conditional (precedence 14):

```
a = b = (((c < (d++)) &&
(e != f)) ? (g + (h / (-i))) : (j >> k))
```

Assignment (precedence 15):

```
(a = (b = (((c < (d++)) &&
(e != f)) ? (g + (h / (-i))) : (j >> k))))
```

The expression is now fully parenthesized.

Precedence	Name	Symbol(s)	Associativity
1	array subscripting	[]	left
1	function call	()	left
1	structure and union member	. ->	left
1	increment (postfix)	++	left
1	decrement (postfix)	—	left
2	increment (prefix)	++	right
2	decrement (prefix)	—	right
2	address of	&	right
2	indirection	*	right
2	unary plus	+	right
2	unary minus	-	right
2	bitwise complement	~	right
2	logical negation	!	right
2	size	sizeof	right
3	cast	()	right
4	multiplicative	* / %	left
5	additive	+ -	left
6	bitwise shift	<< >>	left
7	relational	< > <= >=	left
8	equality	==	left
9	bitwise <i>and</i>	&	left
10	bitwise exclusive <i>or</i>	^	left
11	bitwise inclusive <i>or</i>		left
12	logical <i>and</i>	&&	left
13	logical <i>or</i>		left
14	conditional	?:	right
15	assignment	= *= /= %= += -= <<= >>= &= ^=  =	right
16	comma	,	left

When an expression contains two or more operators with the same precedence, associativity comes into play. Our example expression contains two uses of the = operator. Since assignment operators are right associative,

we first put parentheses around the second = operator and its operands, then around the first = operator and its operands.

## Expression Evaluation

---

The rules of operator precedence and associativity allow us to break any C expression into subexpressions to determine uniquely where the parentheses would go if the expression were fully parenthesized. However, these rules don't always allow us to determine the value of the expression, which may depend on the order in which its subexpressions are evaluated.

C doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical *and*, logical *or*, conditional, and comma operators). Thus, in the expression  $(a + b) * (c - d)$  we don't know whether  $(a + b)$  will be evaluated before  $(c - d)$ .

Most expressions have the same value regardless of the order in which their subexpressions are evaluated. However, this may not be true when a subexpression modifies one of its operands. For example, consider the following statements:

```
a = 5;  
c = (b = a + 2) - (a = 1);
```

After these statements have been executed, the value of *c* will be either 6 or 2. If the subexpression  $(b = a + 2)$  is evaluated first, then *b* is assigned the value 7 and *c* is assigned 6. However, if  $(a = 1)$  is evaluated first, then *b* is assigned the value 3 and *c* is assigned 2.

Because of this problem, it's a good idea to avoid using the assignment operators in subexpressions; instead, use a series of separate assignments. For example, the statements above could be rewritten as

```
a = 5;  
b = a + 2;  
a = 1;  
c = b - a;
```

The value of *c* will always be 6 after these statements are executed.

Besides the assignment operators, the only operators that modify their operands are increment and decrement. When using the increment and decrement operators, be careful that your expressions don't depend on a particular order of evaluation. In the following example, *j* may be assigned one of two values:

```
i = 2;  
j = i * i++;
```

It's natural to assume that *j* is assigned the value 4. However, *j* could just as well be assigned 6 instead, because the first use of *i* could refer to its new value (after *i* is incremented). Here's the scenario: (1) The second operand (the original value of *i*) is fetched, then *i* is incremented. (2) The first operand (the new value of *i*) is fetched. (3) The new and old values of *i* are multiplied, giving the result 6. Moral: Don't assume that postfix increment and decrement operations are postponed until last.

## Expression Statements

---

C has the unusual property that *any* expression can be used as a statement. That is, any expression can be turned into a statement by putting a semicolon after the expression. For example, we can turn the expression `++i` into a statement as follows:

```
++i;
```

When this statement is executed, *i* is first incremented, then the new value of *i* is fetched (as though it were to be used in an enclosing expression). However, since `++i` isn't part of a larger expression, its value is discarded and the next statement executed. (The change to *i* is permanent, of course.)

In general, the value of an expression is always discarded when the expression is used as a statement. Consequently, the statement is useless unless the expression has a side effect. Let's look at three examples. In the first example, 1 is stored into *i*, then the new value of *i* is fetched but not used:

```
i = 1;
```

In the second example, the value of *i* is fetched but not used; however, *i* is decremented afterwards:

```
i--;
```

In the third example, the value of the expression `i * j - 1` is computed and then discarded:

```
i * j - 1;
```

Since *i* and *j* aren't changed, the statement has no effect.

As the first two examples show, expression statements are useful for performing assignments and for incrementing or decrementing variables, because the assignment operators, like the increment and decrement operators, have side effects.

*Warning:* A slip of the finger can easily create a meaningless expression statement.

For example, instead of entering

```
i = j;
```

we might accidentally type

```
i + j;
```

(Believe it or not, this is a fairly common error, since the = and + characters occupy the same key.) TopSpeed C will detect meaningless expression statements and issue the warning “*Code has no effect.*”



# CHAPTER 4

## Basic Types

C's builtin types are called *basic types*. The four types `int` (integer), `float` (single-precision floating-point), `double` (double-precision floating-point), and `char` (character) may be qualified by the words `long`, `short`, `signed`, and `unsigned` to create a great variety of basic types.

In this chapter, we'll see how to use variables and constants of each basic type and how to perform I/O on values of the basic types. We'll also tackle the issue of converting a value of one basic type to another basic type.

### Integer Types

---

On a 16-bit computer (such as the members of the IBM PC family), a value of type `int` occupies 16 bits. The leftmost or most significant bit (the *sign bit*) indicates whether the number is positive or negative; the other 15 bits indicate the magnitude of the number. Consequently, an `int` value must lie between  $-32,768$  and  $32,767$  ( $2^{15}-1$ ). If a variable will never take on a value outside this range, it can be declared to be of type `int`. However, if the value of a variable will exceed  $32,767$  (or become less than  $-32,768$ ), `int` is no longer satisfactory. In this situation, the programmer should declare the variable to be a *long* integer. At other times, we may need to save space by instructing the compiler to allocate fewer bits; such a number is called a *short* integer. On occasion, we may want to treat the sign bit as part of the number's magnitude; the result is an *unsigned* integer.

By adding the specifiers `long`, `short`, `signed`, and `unsigned` to `int`, we can construct an integer type that exactly meets our needs. We can even combine several specifiers (e.g., `long unsigned int`). However, only the following six combinations actually produce different types:

```
short int
unsigned short int
int
unsigned int
long int
unsigned long int
```

Other combinations are synonyms for one of these six types. (For example, `long signed int` is the same as `long int`, since integers are always signed unless otherwise specified.) Incidentally, the order of the specifiers doesn't matter; `unsigned short int` is the same as `short unsigned int`.

C allows the names of integer types to be abbreviated by dropping the word `int`. For example, `unsigned short int` may be abbreviated to `unsigned short`, and `long int` may be abbreviated to `long`.

The following table shows the range of values for each of the six integer types in TopSpeed C:

<i>Type</i>	<i>Smallest Value</i>	<i>Largest Value</i>
short int	−32,768	32,767
unsigned short int	0	65,535
int	−32,768	32,767
unsigned int	0	65,535
long int	−2,147,483,648	2,147,483,647
unsigned long int	0	4,294,967,295

**Note:** short int may represent the same range of values as int; also, int may have the same range as long int. In TopSpeed C, short int and int have identical ranges.

Macros that define the smallest and largest values of each integer type can be found in the header <limits.h>, which is part of the C standard library (see Appendix B).

## Integer Constants

---

Integer constants may be written in decimal (base 10), octal (base 8), or hexadecimal (base 16) notation.

- *Decimal* integer constants contain digits between 0 and 9, but must not begin with a zero:  
**15 255 32767**
- *Octal* integer constants contain only digits between 0 and 7, and *must* begin with a zero:  
**017 0377 077777**
- *Hexadecimal* integer constants contain digits between 0 and 9 and letters between a and f, and always begin with 0x:  
**0xf 0xff 0x7fff**

The letters in a hexadecimal constant may be either upper or lower case:

**0xff 0xfF 0xFf 0xFF  
0Xff 0XfF 0XFf 0XFF**

When an integer constant appears in a program, the compiler treats it as a normal integer if it falls in the

range of normal integers and as a long integer otherwise. The programmer can explicitly indicate that an integer constant should be long by following it with the letter L (or l):

```
15L 0377L 0x7fffL
```

To indicate that a number is unsigned, put the letter U (or u) after it:

```
15U 0377U 0x7fffU
```

L and U may be used in combination.

## Reading and Writing Integers

---

Chapter 2 discussed the conversion specification %d, which is used for reading and writing ordinary integers in decimal form. Reading and writing unsigned, short, and long integers requires several new conversion specifiers:

- When reading or writing an *unsigned* integer, use the letter u, o, or x instead of d in the conversion specification. If the u specifier is present, the number is read (or written) in decimal notation; use o for octal notation and x for hexadecimal notation.

```
unsigned int u;
```

```
printf("%u", u); /* writes u in base 10 */  
printf("%o", u); /* writes u in base 8 */  
printf("%x", u); /* writes u in base 16 */
```

- When reading or writing a *short* integer, put the letter h in front of d, o, u, or x:

```
short int s;  
printf("%hd", s);
```

- When reading or writing a *long* integer, put the letter l in front of d, o, u, or x:

```
long int l;  
printf("%ld", l);
```

## Floating Types

---

C supports three floating types:

float	single-precision floating-point
double	double-precision floating-point

long double                      extended-precision floating-point

In TopSpeed C, values of type float range in magnitude from  $1.17 \times 10^{E38}$  to  $3.40 \times 10^{38}$ , with six digits of precision, and values of type double range between  $2.22 \times 10^{E308}$  and  $1.79 \times 10^{308}$ , with 15 digits of precision.

**Note:** float may have the same set of values as double; similarly, double may have the same values as long double. In TopSpeed C, double and long double have the same values; long double provides no additional range or precision.

Macros that define the characteristics of the floating types can be found in the header <float.h>, which is part of the C standard library (see Appendix B).

## Floating Constants

---

A floating constant is a numerical constant containing a decimal point and/or an exponent. If an exponent is present, it must be preceded by the letter E (or e). The following constants all represent the same number:

```
57.0  57.  57.0e0  57.0E0  57e0
5.7e1  5.7e+1  .57e2  570.e-1
```

The exponent indicates the power of 10 by which the number is to be scaled.

By default, floating constants are stored as double-precision numbers. To indicate that only single precision is desired, put the letter F (or f) at the end of the constant (for example, 5.7e1F). To indicate that a constant should be stored in long double format, put the letter L (or l) at the end (5.7e1L).

## Reading and Writing Floating-Point Numbers

---

The conversion specifications %e, %f, and %g, introduced in Chapter 2, are used for reading and writing single-precision floating-point numbers. Values of types double and long double require slightly different conversion specifications:

- When *reading* a value of type double, put the letter l in front of e, f, or g:

```
double d;
scanf("%lf", &d);
```

**Note:** Use l only in a scanf format string, not a printf string. In a printf format string, the e, f, and g conversions can be used to write either float or double values.

- When reading or writing a value of type long double, put the letter L in front of e, f, or g:

```
long double ld;
printf("%Lf", ld);
```

## Character Types

---

The values of type char are the characters in the underlying character set (ASCII, in the case of TopSpeed C). A variable of type char can be assigned any character in this set:

```
char c;

c = 'a'; /* the letter a */
c = 'A'; /* the letter A */
c = '0'; /* the digit 0 */
c = ' '; /* the space character */
```

Notice that character constants are enclosed in single quotes.

C treats characters as integers. The integer value of a particular character depends on the character set in use. In ASCII, characters are numbered from 0 to 127. The character 'a' has the value 97 in ASCII, 'A' has the value 65, '0' has the value 48, and ' ' has the value 32.

Characters and integers may be mixed freely in expressions:

```
char c;
int i;

i = 'a'; /* i is now 97 */
c = 65; /* c is now 'A' */
c = c + 1; /* c is now 'B' */
c++; /* c is now 'C' */
c = c - 'A' + 'a'; /* c is now 'c' */
```

It's worth noting that `c = c - 'A' + 'a'` converts the character `c` to lower case (assuming that `c` is an upper-case letter to start with) and that `c = c - 'a' + 'A'` converts `c` to upper case (if `c` is a lower-case letter).

The char type may be made signed or unsigned, if desired:

```
signed char c1;
unsigned char c2;
```

In TopSpeed C, signed characters have values between -128 and 127, while unsigned characters have values between 0 and 255.

**Note:** ANSI C doesn't specify whether char itself is a signed or an unsigned type. In TopSpeed C, char is normally a signed type; the pragma option `(uns_char=>on)` makes it an unsigned type.

## Character Constants

A character constant is usually one character enclosed in single quotes, as we've seen in previous examples. However, certain special characters—including the newline character—can't be written in this way, because they're invisible (nonprinting). So that programs can deal with *all* characters in the underlying character set, C provides a special notation, the *escape sequence*.

There are two kinds of escape sequences: *character escapes* and *numeric escapes*. The following is a complete list of character escapes:

alert (bell)	\a	backslash	\\
backspace	\b	question mark	\?
form feed	\f	single quote	\'
new line	\n	double quote	\"
carriage return	\r	horizontal tab	\t
vertical tab	\v		

Character escapes exist only for the characters listed above. However, numeric escapes are capable of representing any character. Numeric escapes use either octal or hexadecimal notation to specify the integer value of the character.

- An *octal escape sequence* consists of the \ character followed by an octal number with at most three digits. (This number must be representable as an unsigned character, so its maximum value in TopSpeed C is 377 octal.) For example, the backspace character, the newline character, and the letter a can be written

**\10 \12 \141**

respectively.

- A *hexadecimal escape sequence* consists of \x followed by a hexadecimal number. Although ANSI C places no limit on the number of digits in the hexadecimal number, it must be representable as an unsigned character (hence it can't exceed FF in TopSpeed C). Using this notation, the backspace character, the newline character, and the letter a would be written

**\x8 \xa \x61**

When used as a character constant, an escape sequence must be enclosed in single quotes. For example, a constant representing the newline character would be written '\n' (or '\12' or '\xa').

ANSI C provides another special notation, *trigraph sequences*, for representing certain ASCII characters that are unavailable on some computers. ANSI C also provides *multibyte characters*, which are used for

representing large non-English alphabets (the Chinese alphabet, for example). Consult the *Language Reference* for more information.

## **Reading and Writing Characters**

The %c conversion specification allows scanf and printf to read and write single characters:

```
char c;

scanf("%c", &c); /* reads a single character */
printf("%c", c); /* writes a single character */
```

scanf doesn't skip white space before reading a character. If the next unread character is a space, then the variable c in the previous example will contain a space after scanf returns. This property of scanf provides a way to detect the end of an input line: check to see if the character just read is the newline character. For example, the following loop will read and ignore all remaining characters in the current input line:

```
scanf("%c", &c);
while (c != '\n') scanf("%c", &c);
```

When the while loop terminates, c will contain the newline character. When scanf is called the next time, it will read the first character on the next input line.

C provides another way to read and write single characters: call getchar and putchar instead of scanf and printf. Each time getchar is called, it returns a single character. In order to save the character that getchar has read, we must explicitly store it in a variable:

```
/* read a character and store it in c */

c = getchar();
```

putchar writes a single character:

```
putchar(c);
```

Using getchar and putchar (rather than scanf and printf) saves time when the program is executed. getchar and putchar are fast for two reasons. First, they're much simpler than scanf and printf, which must be prepared to read and write many kinds of data in a variety of formats. Second, getchar and putchar are usually implemented as macros (see Chapter 10) for additional speed.

---

## **Type Conversion**

Most binary operators in C require operands of the same type. When operands have different types, one of the operands (or in some cases, both operands) must be converted before the operation can be performed.

There are two ways to change the type of an operand: *implicit conversion* (performed automatically) and *explicit conversion* (performed by the *cast* operation). Let's look first at implicit conversion; we'll discuss casting later in the chapter.

There are four occasions when implicit conversions are performed:

- When the operands in an expression don't have appropriate types.
- When the type of the expression on the right side of an assignment doesn't match the type of the variable on the left side.
- When the type of an actual parameter in a function call doesn't match the type of the corresponding formal parameter.
- When the type of the expression in a return statement doesn't match the function's return type.

We'll discuss the first two cases now; the other two are covered in Chapter 7.

### **The Usual Arithmetic Conversions**

The *usual arithmetic conversions* are applied to the operands of many binary operators, including the arithmetic, relational, and equality operators. The strategy behind these conversions is to convert the operands to the "smallest" type that will safely accommodate both values. The types of the operands can often be made to match by converting the operand of the "smaller" type to the type of the other operand (this act is known as *promotion*). Among the most common promotions are the *integral promotions*, which convert a character or short integer to type int (or unsigned int if necessary).

We can divide the rules for performing the usual arithmetic conversions into two cases:

- *The type of either operand is a floating type.* Use the following chart to promote the operand whose type is "smaller":

```

long double
-
double
-
float
```

That is, if one operand has type long double, then convert the other operand to type long double. Otherwise, if one operand has type double, convert the other operand to type double. Otherwise, if one operand has type float, convert the other operand to type float.

- *Neither operand type is a floating type.* First perform integral promotion on both operands (guaranteeing that neither operand



will be a character or short integer). Then use the following chart to promote the operand whose type is “smaller”:

```

        unsigned long int
        -
        long int
        -
        unsigned int
        -
        int

```

**Warning:** When a signed operand is combined with an unsigned operand, the signed operand is converted to an unsigned value; this can cause obscure programming errors. For example, suppose that the int variable *i* has the value -10 and the unsigned int variable *u* has the value 10. If we compare *i* and *u* using the < operator, we might expect to get the result 1 (true). Before the comparison, however, *i* is converted to type unsigned int. Since a negative number can't be represented as a value of type unsigned int, the converted value won't be -10, but a large positive number (the result of interpreting the bits in *i* as an unsigned number). The comparison *i* < *u* will therefore produce 0.

Here are some examples showing the usual arithmetic conversions in action:

```

char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c; /* c is converted to int */
i = i + s; /* s is converted to int */

u = u + i;
/* i is converted to unsigned int */
l = l + u;
/* u is converted to long int */
ul = ul + l;
/* l converted to unsigned long int */
f = f + ul;
/* ul is converted to float */
d = d + f;
/* f is converted to double */
ld = ld + d;
/* d is converted to long double */

```

## Conversion During Assignment

---

The usual arithmetic conversions don't apply to assignment. Instead, C follows the simple rule that the value on the right side of the assignment is converted to the type of the variable on the left side. This may or may not involve promotion.

*Warning:*

Converting from a “large” type to a “small” type may not always produce a meaningful result. Consider the following example:

```
int i;  
float f;  
  
i = f;
```

This assignment is meaningful only if the value of `f` when converted to an integer lies between the smallest and largest values allowed for an `int` variable (between  $-2^{31}$  and  $2^{31}-1$  in TopSpeed C). If `f` is too large or too small, `i` will be assigned an apparently meaningless number.

Incidentally, the conversion of a floating-point number to an integer is done by dropping the fractional part of the number (*not* by rounding to the nearest integer):

```
int i;  
  
i = 842.97;    /* i is now 842 */  
i = -842.97;   /* i is now -842 */
```

## Casting

---

Although C's implicit conversions are convenient, we sometimes need a greater degree of control. For this reason, C provides *casts*. A cast expression has the form

```
( type-name ) expression
```

The type name in parentheses indicates the type to which the expression should be converted.

The following example shows how to use a cast expression to compute the fractional part of a float value:

```
float f, frac_part;  
  
frac_part = f - (int) f;
```

The cast expression `(int) f` represents the result of converting the value of `f` to type `int`. C's usual arithmetic conversions then require that `(int) f` be converted back to type `float` before the subtraction can be performed. The difference between `f` and `(int) f` is the fractional part of `f`, which was dropped during the conversion of `f` to type `int`.

# CHAPTER 5

## Statements

Although C has many operators, it has relatively few statements. We've encountered two already: the compound statement (Chapter 2) and the expression statement (Chapter 3).

C's other statements can be divided into three categories:

- *Selection statements.* The if and switch statements allow a program to select an execution path from among several alternatives.
- *Iteration statements.* The while, do, and for statements support iteration (looping).
- *Jump statements.* The break, continue, goto, and return statements cause an unconditional transfer of control.

C also provides the null statement, which performs no action.

This chapter discusses all of the statements listed above, with the exception of the return statement, which is covered in Chapter 7.

### The if Statement

---

The if statement provides a way to choose between alternative execution paths by testing the value of an expression. if statements have the form

```
if ( expression ) statement
```

If the value of the expression in parentheses is nonzero, the statement after the parentheses is executed.

Let's look at an example of the if statement:

```
if (num_items == MAX_ITEMS)
    num_items = 0;
```

The statement `num_items = 0;` is executed if the condition `num_items == MAX_ITEMS` is true (has a nonzero value). What if we want to execute two statements if `num_items == MAX_ITEMS` is true? We simply replace `num_items = 0;` by a compound statement:

```
if (num_items == MAX_ITEMS) {
    printf("\n");
    num_items = 0;
}
```

**Warning:** Don't confuse `==` (equality) with `=` (assignment).

The statement

```
if (i == 0) ...
```

tests whether *i* is equal to 0. However, the statement

```
if (i = 0) ...
```

assigns 0 to *i*, then tests whether the *result* is nonzero. In this case, the test always fails.

Confusing `==` with `=` is perhaps the most common C programming error. TopSpeed C issues the warning “*Assignment in test expression*” if it detects an occurrence of `=` where `==` would normally be used.

## **Boolean Values**

The values 0 and 1 are used throughout C to represent the Boolean values *false* and *true*. (More precisely, 0 represents *false* and any nonzero value represents *true*.) One way to store a Boolean value for later use is to declare an integer variable, then assign it either a nonzero value (representing a true condition) or a zero value (to represent a false condition):

```
int flag; /* represents a Boolean condition */

flag = 1; /* sets the condition to be true */
...
flag = 0; /* sets the condition to be false */
```

Although this scheme works, it doesn’t contribute much to program readability. Someone reading the program may overlook the fact that *flag* is to be assigned only Boolean values and that 1 and 0 represent *true* and *false*.

To make programs more understandable, it’s a good idea to define macros named `boolean`, `TRUE`, and `FALSE`:

```
#define boolean int
#define TRUE 1
#define FALSE 0
```

`boolean` can be used in place of `int` to declare Boolean variables. The previous example now has the following appearance:

```
boolean flag;

flag = TRUE;
...
flag = FALSE;
```

It’s now clear that *flag* isn’t an ordinary integer variable, but instead represents a Boolean condition.

To test whether *flag* is *true*, we can write

```
if (flag == TRUE) ...
or just
```

```
if (flag) ...  
To test whether flag is false, we can write  
if (flag == FALSE) ...  
or  
if (!flag) ...
```

## The else Clause

---

An if statement may have an else clause:

*if ( expression ) statement else statement*

The statement that follows the word else is executed if the expression in parentheses has the value 0.

Here's an example of an if statement with an else clause:

```
if (i > j) max = i;  
else max = j;
```

Notice that the statement `max = i;` must still end with a semicolon, even though it's nested inside the if statement.

Here's an example of an if statement that contains two other if statements:

```
if (i > j)  
    if (i > k) max = i;  
    else max = k;  
else  
    if (j > k) max = j;  
    else max = k;
```

if statements can be nested to any depth. Notice how each else is aligned with the matching if; this alignment makes the nesting easier to see.

### The “Dangling else” Problem

Despite their simplicity, if statements contain a few pitfalls. One of the most notorious is the “dangling else” problem. Consider the following if statement, which contains another if statement nested inside it:

```
if (y != 0)  
    if (x != 0)  
        result = x / y;  
else  
    printf("Error: y is equal to 0\n");
```

To which if statement does the else clause belong? The indentation suggests that it belongs to the outer if statement. However, C follows that rule that an else clause belongs to the nearest if statement that hasn't already been paired with an else. In this example, the else clause actually belongs to the inner if statement; a correctly indented version would look like this:

```
if (y != 0)
    if (x != 0)
        result = x / y;
    else
        printf("Error: y is equal to 0\n");
```

To make the else clause part of the outer if statement, we can enclose the inner if statement in braces:

```
if (y != 0) {
    if (x != 0)
        result = x / y;
} else
    printf("Error: y is equal to 0\n");
```

This example illustrates the danger of careless indentation. Indentation should make programs easier to read, but if it doesn't reflect the actual nesting of statements, it actually makes programs much harder to read (and debug).

### **Side Effects in if Statements**

Another common pitfall in the if statement arises when the condition to be tested has side effects. Because the operators `&&` and `||` perform shortcircuit evaluation of their operands, the side effects may not always occur. Consider the following example:

```
if (i > 0 && ++j > 0) ...
```

Although `j` is apparently incremented as a side effect of executing the if statement, this isn't always the case. If the condition `i > 0` is false, then `++j > 0` is not evaluated, so `j` isn't incremented. The problem can be fixed by changing the condition to `++j > 0 && i > 0` or, even better, by incrementing `j` separately:

```
++j;
if (i > 0 && j > 0) ...
```

## **The switch Statement**

---

One way to test a series of conditions is to “cascade” a number of if statements. For example, the following cascaded if statement tests the value of the variable `grade`:

```
if (grade == 'A')
    printf("Excellent");
else if (grade == 'B')
    printf("Good");
else if (grade == 'C')
    printf("Average");
else if (grade == 'D')
    printf("Poor");
else if (grade == 'F')
    printf("Failing");
else
    printf("Illegal grade");
```

This particular kind of cascaded if statement in which an expression is compared with a set of possible values is so common that C provides a simpler alternative: the switch statement. The following switch statement is equivalent to our lengthy cascaded if statement:

```
switch (grade) {
    case 'A': printf("Excellent");
               break;
    case 'B': printf("Good");
               break;
    case 'C': printf("Average");
               break;
    case 'D': printf("Poor");
               break;
    case 'F': printf("Failing");
               break;
    default:  printf("Illegal grade");
}
```

When this statement is executed, the value of the variable grade is tested against 'A', 'B', 'C', 'D', and 'F'. If it matches 'A', for example, the message "Excellent" is printed, then control passes to the statement after the } symbol. (The break statement which we'll discuss later in this chapter transfers control out of the switch statement.) If the value of grade doesn't match any of the choices listed, the default case applies, and the message "Illegal grade" is printed.

In general, switch statements have the form

```
switch ( expression ) {
    case constant-expression : statements
    0
    case constant-expression : statements
    default : statements
}
```

The word switch must be followed by a character or integer expression in parentheses. Each case label must contain a character or integer expression whose value is constant (the expression may contain constants and operators, but not variables or function calls).

Only one constant expression may follow the word case; however, several case labels may precede the same group of statements:

```
switch (grade) {
    case 'A': case 'B': case 'C': case 'D':
               printf("Passing");
               break;
    case 'F': printf("Failing");
               break;
    default:  printf("Illegal grade");
}
```

The last statement in each group is usually break, but this isn't required. Without break (or some other jump statement), control will flow from one case into the next. Consider the following switch statement:



```

switch (grade) {
    case 'A': printf("Excellent");
    case 'B': printf("Good");
    case 'C': printf("Average");
    case 'D': printf("Poor");
    case 'F': printf("Failing");
    default: printf("Illegal grade");
}

```

If the value of grade is 'A', the message printed is

ExcellentGoodAveragePoorFailingIllegal grade

since control flows through all five cases plus the default case.

*Warning:* Forgetting to end each case with `break` is a common error—one that the compiler can't detect. Although omitting `break` is sometimes done intentionally to allow several cases to share code, it's usually just an oversight.

A switch statement isn't required to have a default case. If it's missing and the value of the controlling expression doesn't match any of the case labels, control passes to the next statement after the switch.

## The while Statement

---

The while statement is one of C's three iteration statements. The while statement has the form

```
while ( expression ) statement
```

The expression in parentheses is the *controlling expression*; the statement after the parentheses is the *loop body*.

When a while statement is executed, the controlling expression is evaluated first. If its value is nonzero, the loop body is executed and the expression is tested again. The process continues in this fashion—first testing the controlling expression, then executing the loop body—until the expression has a zero value. Since the controlling expression is tested *before* the loop body is executed, it's possible that the body isn't executed even once.

Here's an example of a while statement:

```

i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    --i;
}

```

When the while statement is executed, the variable `i` has the value 10. Since 10 is greater than 0, the loop body is executed, causing the message T minus 10 and counting to be printed and `i` to be decremented. The condition `i > 0` is now tested again. Since 9 is greater than 0, the loop body is executed once

more. This process continues until the message T minus 1 and counting is printed and i becomes 0. The test  $i > 0$  now fails, causing the loop to terminate.

The previous while statement could have been written in a number of different ways. For example, we can eliminate the braces in the loop body by decrementing i inside the call of printf:

```
i = 10;
while (i) printf("T minus %d and counting\n", i--);
```

We've also shortened the controlling expression; the loop will now execute as long as i isn't equal to 0. Provided that i is never negative, the new test is equivalent to the old one.

A while statement won't terminate if the controlling expression never becomes zero. Sometimes, programmers deliberately create an infinite loop by using a nonzero constant as the controlling expression:

```
while (1) {
    ...
}
```

A while statement of this form will execute forever unless its body contains a statement that transfers control out of the loop or causes the program to terminate.

## The do Statement

---

The do statement is closely related to the while statement; in fact, the do statement is just a while statement whose controlling expression is tested *after* each execution of the loop body. The do statement has the form

```
do statement while ( expression );
```

When a do statement is executed, the loop body (the statement between do and while) is executed first, then the controlling expression is evaluated. If the value of the expression is nonzero, the loop body is executed again and then the expression is evaluated once more. Execution of the do statement terminates when the controlling expression has the value 0.

Let's rewrite our while statement example, using a do statement this time:

```
i = 10;
do {
    printf("T minus %d and counting\n", i);
    --i;
} while (i > 0);
```

When the do statement is executed, the loop body is first executed, causing the message T minus 10 and counting to be printed and i to be decremented. The condition  $i > 0$  is now tested. Since 9 is greater than 0, the loop body is executed a second time. This process continues until the message T minus 1

and counting is printed and `i` becomes 0. The test `i > 0` now fails, causing the loop to terminate.

As this example shows, the `do` statement frequently produces the same results as the `while` statement. The difference between the two is that the body of a `do` statement is always executed at least once; the body of a `while` statement may be skipped entirely if the controlling expression is 0 initially.

### **Example Program: Finding the Largest in a List of Numbers**

Let's write a program that finds the largest number in a list entered by the user. The program will repeatedly prompt the user to enter a number. When the user enters 0 or a negative number, the program will print the largest nonnegative number entered. For example, the program might produce the following output:

```
Enter a number: 60
Enter a number: 38.3
Enter a number: 4.89
Enter a number: 100
Enter a number: 75.2295
Enter a number: 0
```

The largest number entered was 100

Clearly we'll need some kind of loop, but should we use a `while` statement or a `do` statement? The `do` statement turns out to be the better choice. Here's a version of the program built around a `do` loop:

```
#include <stdio.h>

main()
{
    float max, x;

    max = 0.0;
    do {
        printf("Enter a number: ");
        scanf("%f", &x);
        if (x > max) max = x;
    } while (x > 0.0);
    printf("\nThe largest number entered ");
    printf("was %g\n", max);
}
```

To see why the `do` statement is the right choice, let's look at an equivalent `while` loop:

```
printf("Enter a number: ");
scanf("%f", &x);
while (x > 0.0) {
    if (x > max) max = x;
    printf("Enter a number: ");
    scanf("%f", &x);
}
```

The while statement tests the condition  $x > 0.0$  before the loop body has been executed even once, so we're forced to put extra calls of `printf` and `scanf` before the while statement to guarantee that  $x$  has a value to be tested.

## The for Statement

---

The for statement has the form

```
for ( expr1 ; expr2 ; expr3 ) statement
```

where *expr1*, *expr2*, and *expr3* are expressions.

The for statement is closely related to the while statement. In fact, any for statement can be replaced by the following statements:

```
expr1;
while (expr2) {
    statement
    expr3;
}
```

In other words, *expr1* is an initialization step to be performed before the loop is executed, *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero), and *expr3* is an operation to be performed at the end of each loop iteration.

Here's a simple example of a for statement:

```
for (i = 10; i > 0; -i)
    printf("T minus %d and counting\n", i);
```

To see what this statement does, let's convert it to an equivalent while statement:

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    -i;
}
```

When the for statement is executed, the variable *i* is initialized to 10, then *i* is tested to see if it's greater than 0. Since it is, the message T minus 10 and counting is printed, then *i* is decremented. The condition  $i > 0$  is then tested again. The loop body will be executed 10 times in all, with *i* varying from 10 down to 1.

Incidentally, the third expression in this for statement, `—i`, could be replaced by `i—` if desired. Since the first and third expressions in a for statement are executed as entire statements, their values are irrelevant—they're useful only for their side effects. As a result, these two expressions are usually assignments or increment/decrement expressions.

## Omitting Expressions in a for Statement

Any or all of the three expressions in a for statement may be omitted. If the first expression is omitted, no initialization is performed before the loop is executed:

```
i = 10;
for (; i > 0; -i)
    printf("T minus %d and counting\n", i);
```

In this example, *i* has been initialized by a separate assignment, so we've omitted the first expression in the for statement. (Notice that the semicolon between the first and second expressions remains. The two semicolons must always be present, even when we've omitted some of the expressions.)

If the second expression is missing, the for statement doesn't terminate (unless stopped in some other fashion). For example, programmers often use the following for statement to establish an infinite loop:

```
for (;;) {
    ...
}
```

If we omit the third expression in a for statement, the loop body is responsible for making sure that the statement's second expression eventually becomes 0. Our example for statement could be written like this:

```
for (i = 10; i > 0;)
    printf("T minus %d and counting\n", i--);
```

To compensate for omitting the third expression, we've arranged for *i* to be decremented inside the loop body. As a result, this for statement is equivalent to the original one.

## The Comma Operator

On occasion, we might like to write a for statement with *two* initialization expressions or one that increments *two* variables each time through the loop. We can do this by using a *comma expression* as the first or third expression in the for statement.

A comma expression has the form

```
expr1 , expr2
```

where *expr1* and *expr2* are any two expressions. The comma operator allows us to "glue" two expressions together to form a single expression.

The comma operator is useful in situations where C requires a single expression, but we'd like to include two or more expressions. For example, suppose that we want to initialize two variables when entering a for statement. Instead of writing

```
sum = 0;
for (i = 1; i <= N; i++) sum += i;
```

we can write

```
for (sum = 0, i = 1; i <= N; i++) sum += i;
```

The expression `sum = 0, i = 1` first assigns 0 to `sum`, then assigns 1 to `i`. With additional commas, the `for` statement could initialize more than two variables.

A comma expression is evaluated in two steps: First, *expr1* is evaluated; its value, if any, is discarded. Second, *expr2* is evaluated; its value is the value of the entire expression. For example, suppose that the variables `i` and `j` have the values 1 and 5, respectively. When the comma expression `i++, i+j` is evaluated, `i` is first incremented, then `i+j` is evaluated, so the value of the expression is 7. (And, of course, `i` now has the value 2.) The precedence of the comma operator is less than that of all other operators, so there's no need to put parentheses around `i++` and `i+j`.

Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose, since its value is discarded.

Comma expressions are also useful in macros (see Chapter 10). Other than in `for` statements and macros, the comma operator is best used sparingly.

### **Example Program: Printing a Table of Squares (Revisited)**

The `for` statement in C is more powerful than the `for` statement in other common programming languages because C places no restrictions on the three expressions that control its behavior. In particular, the three expressions need not involve the same variable. As an example, consider the following program, a modification of the program in Chapter 2 that prints a table of squares:

```
#include <stdio.h>

main()
{
    int i, n, odd, square;

    printf("This program prints ");
    printf("a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    i = 1;
    odd = 3;
    for (square = 1; i <= n; odd += 2) {
        printf("%10d%10d\n", i, square);
        ++i;
        square += odd;
    }
}
```

The `for` statement in this program initializes one variable (`square`), tests another (`i`), and increments a third (`odd`), thereby illustrating the great flexibility of the `for` statement. `i` is the number to be squared, `square` is the square of `i`, and `odd` is the odd number that must be added to the current square to get the next square (allowing the program to compute consecutive squares without performing any multiplications).

## The break Statement

---

The `break` statement transfers control out of an enclosing `while`, `do`, `for`, or `switch` statement. In other words, `break` is a restricted form of `goto`.

We've already seen examples of how `break` is used in `switch` statements. In a `while`, `do`, or `for` loop, `break` is useful for terminating the loop prematurely. For example, suppose that we're writing a `for` statement that checks whether a number `n` is prime by successively dividing `n` by the numbers between 2 and  $n - 1$ . We can break out of the loop as soon as any divisor is found; there's no need to try the remaining possibilities. After the loop has terminated, we can use an `if` statement to determine whether termination was premature (hence `n` isn't prime) or normal (`n` is prime):

```
for (divisor = 2; divisor < n; divisor++)
    if (n % divisor == 0) break;
if (divisor < n)
    printf("%d is not prime\n", n);
else
    printf("%d is prime\n", n);
```

The `break` statement is particularly useful for writing loops in which the exit test is in the middle rather than at the beginning or end. In the following `for` statement, for example, the exit test is in the middle:

```
for (;;) {
    printf("Enter a number (enter 0 to stop): ");
    scanf("%d", &n);
    if (n == 0) break;
    printf("%d cubed is %d\n", n, n*n*n);
}
```

When statements are nested, executing a `break` statement transfers control out of the innermost `while`, `do`, `for`, or `switch` that encloses the `break`. Consider the case of a `switch` statement nested inside a `while` statement:

```
while (...) {
    switch (...) {
        ...
        break;
        ...
    }
}
```

The `break` statement transfers control out of the `switch` statement, but not out of the `while` loop.

## The continue Statement

---

The `continue` statement, like the `break` statement, is a restricted form of the `goto` statement. However, `break` transfers control just *past* the end of a loop, while `continue` transfers control to a point just *before* the end of the loop

body. With `break`, control leaves the loop; with `continue`, control remains inside the loop.

The following example illustrates a typical use of `continue`. This example reads a series of numbers and computes their sum. The loop terminates when ten nonzero numbers have been read. Whenever the number 0 is read, the `continue` statement is executed, skipping the rest of the loop body (the statements `++n`; and `sum += i`;) but remaining inside the loop.

```
n = 0;
sum = 0;
do {
    scanf("%d", &i);
    if (i == 0) continue;
    ++n;
    sum += i;
} while (n < 10);
```

The `continue` statement can appear only inside `while`, `do`, and `for` statements.

## The goto Statement

---

The `goto` statement has the form

```
goto identifier ;
```

where the identifier is a *label*. The effect of executing a `goto` statement is to transfer control to the statement that follows the label, which must be in the same function as the `goto` statement itself.

Any statement may be preceded by a label:

```
identifier : statement
```

A statement may have more than one label.

If C didn't have a `break` statement, here's how we would use the `goto` statement to exit from a loop:

```
for (divisor = 2; divisor < n; divisor++)
    if (n % divisor == 0) goto done;
done:
if (divisor < n)
    printf("%d is not prime\n", n);
else
    printf("%d is prime\n", n);
```

The `goto` statement is rarely needed in everyday C programming. The `break`, `continue`, and `return` statements—which are essentially restricted `goto` statements—and the `exit` function (Chapter 7) are sufficient to handle most situations that might require a `goto` in other programming languages.



## The Null Statement

---

The null statement is just a semicolon. Here's an example:

```
i = 0; ; j = 1;
```

This line contains three statements: an assignment to `i`, a null statement, and an assignment to `j`.

The null statement is useful when a label must be placed at the end of a compound statement:

```
{  
    ...  
    goto end_of_stmt;  
    ...  
    end_of_stmt: ;  
}
```

A label must always be followed by a statement; however, the statement may be null.

The null statement is also useful when the body of a loop is to be left empty. The classic case of a loop with an empty body is the following statement:

```
while ((ch = getchar()) == ' ');
```

This statement reads a character, stores it into the variable `ch`, then tests whether `ch` contains a blank. If so, the loop body (which is empty) is executed, then the loop test is performed once more, causing a new character to be read. In other words, the while statement repeatedly reads characters until it finds a nonblank character. The value of `ch` when the loop terminates.

**Note:** Some programmers prefer to put the semicolon on a separate line to draw attention to the fact that the loop body is empty:

```
while ((ch = getchar()) == ' ')  
;
```

**Warning:**

Accidentally putting a semicolon after the parentheses in an if, while, or for statement ends the statement prematurely; the compiler can't detect an error of this kind. See next section

## Unintentional Null Statements

---

An accidental semicolon often causes unexpected problems in one of the following situations, which you should look out for because they are common mistakes

- In an if statement, putting a semicolon after the parentheses creates an if statement that apparently performs the same action regardless of the value of its controlling expression:

```

        if (divisor == 0); /* error */
            printf("Error: Division by
zero\n");

```

The call of printf isn't inside the if statement, so it's performed regardless of whether divisor is equal to 0.

- In a while statement, putting a semicolon after the parentheses may create an infinite loop:

```

i = 10;
while (i > 0); /* error */
{
    printf("T minus %d and counting\n", i);
    —i;
}

```

Another possibility is that the loop terminates, but the statement that should be the loop body is executed only once, after the loop has terminated:

```

i = 11;
while (—i > 0); /* error */
    printf("T minus %d and counting\n", i);

```

This example prints the message

T minus 0 and counting

- In a for statement, putting a semicolon after the parentheses causes the statement that should be the loop body to be executed only once:

```

for (i = 10; i > 0; i—); /* error */
    printf("T minus %d and counting\n", i);

```

This example also prints the message

T minus 0 and counting

# CHAPTER 6

## Arrays

Creating arrays is the primary method of structuring data in C. (We'll cover the other methods in Chapter 12.) In this chapter, we'll see how to define one-dimensional and multidimensional arrays and how to initialize arrays. Chapter 8 contains further discussion of arrays. In particular, the close relationship between arrays and pointers.

### One-Dimensional Arrays

---

An *array* is a data structure containing a number of data values, all of which belong to the same type. These values, known as *elements*, can be individually selected by their position within the array.

The simplest kind of array is the one-dimensional array. The elements of a one-dimensional array are arranged one after another in a single row (or column, if you prefer). Thus, a one-dimensional array named *a* might have the following appearance:

To declare an array, we must specify the type of the array's elements and the number of elements. For example, to declare that the array *a* has 10 elements of type *int*, we would write

```
int a[10];
```

The elements of an array may be of any type; the length of the array can be specified by any (integer) constant expression.

#### Array Subscripting

An individual element of an array can be accessed by writing the array name followed by an integer value in square brackets (this is referred to as *subscripting* or *indexing* the array). Array elements are always numbered starting from 0, so the elements of an array of length *n* are indexed from 0 to *n* - 1. For example, the elements of *a* are *a*[0], *a*[1], ..., *a*[9], as the following figure shows:

The elements of an array are lvalues and can therefore be used in the same way as ordinary variables. For example, the element *a*[0] can be used as though it were a variable of type *int*:

```
a[0] = 1;
printf("%d\n", a[0]);
++a[0];
a[0]--;
```

**Warning:**

**C doesn't require that subscript bounds be checked; when a subscript goes out of bounds, the program may behave unpredictably. (TopSpeed C will check subscript bounds during program execution if the program was compiled with the /ri option.)**

One common cause of a subscript out of bounds is forgetting that an array with  $n$  elements is indexed from 0 to  $n - 1$ , not 1 to  $n$ . The following example illustrates this situation:

```
int i, a[10];

for (i = 1; i <= 10; i++)
    a[i] = 0;
```

With some implementations (not TopSpeed C), this innocent-looking for statement causes an infinite loop! When  $i$  reaches 10, the program stores 0 into  $a[10]$ . But  $a[10]$  doesn't exist, so 0 is stored into memory immediately after  $a[9]$ . If the variable  $i$  follows  $a[9]$  in memory—as might be the case—then  $i$  will be reset to 0, causing the loop to start over again.

An array subscript may be any integer expression:

```
a[i] = 0;
a[i+j*10] = 0;
a[i++] = 0;
```

**Warning:**

Be careful when an array subscript has a side effect. For example, the following loop—which is supposed to copy the array  $b$  into the array  $a$ —may not work properly:

```
i = 0;
while (i < n) a[i] = b[i++];
```

If all goes well,  $a[i]$  is evaluated before  $b[i++]$ , so that  $b[i]$  is copied to  $a[i]$ . However, C implementations are free to evaluate  $b[i++]$  before  $a[i]$ , causing the loop to copy  $b[i]$  to  $a[i+1]$ . We can avoid the problem altogether by writing

```
for (i = 0; i < n; i++) a[i] = b[i];
```

Note that the compound assignment

```
a[i++] += 2;
```

is safe, since  $a[i++]$  is evaluated only once. The corresponding simple assignment

```
a[i++] = a[i++] + 2;
```

suffers from the problem described in the warning.

### **Example Program: Reversing a List of Numbers**

Our first example program prompts the user to enter ten numbers, then writes the numbers in reverse order:

```
Enter ten numbers: 34 82 49 12 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 12 49 82 34
```

The program must store the complete list of ten numbers before reversing it; using an array to store the numbers is a natural choice.

```
#include <stdio.h>

#define N 10

main()
{
    int a[N], i;

    printf("Enter ten numbers: ");
    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);

    printf("In reverse order:");
    for (i = N - 1; i >= 0; i--)
        printf(" %d", a[i]);
    putchar('\n');
}
```

Notice that we must use the & symbol when calling scanf to read an array element, just as we would with an ordinary variable. Note also the use of the macro N to define the length of the array; changing the program to reverse lists of other lengths requires only that we alter the definition of N.

### **Example Program: Checking a Number for Repeated Digits**

Our next example program checks numbers for repeated digits. The program will interact with the user, repeatedly requesting a number, then printing a message indicating whether or not the number contains a repeated digit. Entering a number less than or equal to 0 causes the program to terminate. A session with the program might look like this:

```
Enter a number: 9357
No repeated digit

Enter a number: 28212
Repeated digit

Enter a number: 0
```

The program uses an array of Boolean values to keep track of which digits appear in a number. The array, named `digit_seen`, is indexed from 0 to 9 to correspond to the ten possible digits. When given a number `n`, the program initializes all elements of `digit_seen` to `FALSE`, then examines `n`'s digits one at a time, using each digit `d` as an index into `digit_seen`. If the value of `digit_seen[d]` is `TRUE`, then `d` appears at least twice in `n`. On the other hand, if `digit_seen[d]` is `FALSE`, then `d` hasn't been seen before, so we haven't yet

found a repeated digit. (In the latter case, the program sets `digit_seen[d]` to `TRUE`.)

```
#include <stdio.h>

#define boolean int
#define TRUE 1
#define FALSE 0

main()
{
    boolean digit_seen[10], repeated_digit;
    int d;
    long int n;

    printf("Enter a number: ");
    scanf("%ld", &n);

    while (n > 0) {
        repeated_digit = FALSE;
        for (d = 0; d < 10; d++)
            digit_seen[d] = FALSE;

        while (n > 0) {
            d = n % 10;
            if (digit_seen[d]) {
                repeated_digit = TRUE;
                break;
            }
            digit_seen[d] = TRUE;
            n /= 10;
        }

        if (repeated_digit)
            printf("Repeated digit\n\n");
        else
            printf("No repeated digit\n\n");

        printf("Enter a number: ");
        scanf("%ld", &n);
    }
}
```

Notice that the program uses a long int variable to store the input; this allows the user to enter numbers up to 2,147,483,647.

## Multidimensional Arrays

---

In C, an array may have any number of dimensions. For example, the following declaration creates a two-dimensional array (or *matrix*, in mathematical terminology):

```
int m[5][9];
```

The array `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0, as the following figure shows:

To access the element of  $m$  in row  $i$ , column  $j$ , we must write  $m[i][j]$  (*not*  $m[i,j]$ ), as you might expect from other programming languages).

The following example uses nested for statements to initialize a two-dimensional array. The for statements initialize the  $N \times N$  matrix `ident` by storing 1 into elements on the main diagonal (where the row and column index are the same) and 0 into all other elements:

```
#define N 10

int ident[N][N], row, col;

for (row = 0; row < N; row++)
    for (col = 0; col < N; col++)
        if (row == col)
            ident[row][col] = 1;
        else
            ident[row][col] = 0;
```

## Array Initialization

---

An array may be given an initial value by including an *array initializer*—a list of constant expressions enclosed in braces—in the declaration of the array:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

If the initializer is *shorter* than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is
   {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

It's illegal for the initializer to be *longer* than the array.

If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

In this case, the compiler uses the length of the initializer to determine the length of the array. *Note:* The array still has a fixed length, just as if we had specified the length explicitly.

Incidentally, ordinary variables—not just arrays—can be initialized in a declaration:

```
int count = 0;
```

Chapter 11 contains more information about initializers.

## Initializing a Multidimensional Array

---

An initializer for a two-dimensional array can be created by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

Each inner initializer provides values for one row of the matrix. Initializers for higher-dimensional arrays are constructed in a similar fashion.

The following rules allow us to simplify initializers for multidimensional arrays:

- If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0. For example, the following initializer fills only the first three rows of `m`; the last two rows will contain zeros:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1},
               {0, 1, 0, 1, 1, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- The inner braces can be omitted:

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

Once the compiler has seen enough values to fill one row, it begins filling the next.



# CHAPTER 7

## Functions

Functions in C are comparable to *subroutines* or *procedures* in other programming languages. Functions allow us to divide a program into parts that are easier to understand and modify, to clarify the structure of a program, and to avoid code duplication.

The example programs in previous chapters have contained only one function: `main`. In general, however, programs may have other functions in addition to `main`. Like `main`, each of these functions may contain both declarations and statements.

In this chapter, we'll first show how to define functions; we'll then discuss function declarations and how they differ from function definitions. We'll examine two ways in which functions can communicate: by passing parameters and by sharing external variables. Other topics include the `return` statement and recursive functions. The chapter concludes with a look at the overall organization of a C program.

### Function Definitions

---

Before giving the formal rules for defining a function, let's look at the definitions of three simple functions: `print_message`, `print_count`, and `average`.

#### The `print_message` Function

The `print_message` function prints a brief message each time it's called:

### The `switch` Statement

---

One way to test a series of conditions is to “cascade” a number of `if` statements. For example, the following cascaded `if` statement tests the value of the variable `grade`:

```

if (grade == 'A')
    printf("Excellent");
else if (grade == 'B')
    printf("Good");
else if (grade == 'C')
    printf("Average");
else if (grade == 'D')
    printf("Poor");
else if (grade == 'F')
    printf("Failing");
else
    printf("Illegal grade");

```

This particular kind of cascaded if statement is so common that C provides a simpler alternative: the switch statement. The following switch statement is equivalent to our lengthy cascaded if statement:

```

switch (grade) {
    case 'A': printf("Excellent");
               break;
    case 'B': printf("Good");
               break;
    case 'C': printf("Average");
               break;
    case 'D': printf("Poor");
               break;
    case 'F': printf("Failing");
               break;
    default:  printf("Illegal grade");
}

```

When this statement is executed, the value of the variable `grade` is tested against 'A', 'B', 'C', 'D', and 'F'. If it matches 'A', for example, the message "Excellent" is printed, then control passes to the statement after the `}` symbol. (The `break` statement—which we'll discuss later in this chapter—transfers control out of the switch statement.) If the value of `grade` doesn't match any of the choices listed, the default case applies, and the message "Illegal grade" is printed.

In general, switch statements have the form

```

switch ( expression ) {
    case constant-expression : statements
    case constant-expression : statements
    default : statements
}

```

The word `switch` must be followed by a character or integer expression in parentheses. Each case label must contain a character or integer expression whose value is constant (the expression may contain constants and operators, but not variables or function calls).

Only one constant expression may follow the word `case`; however, several case labels may precede the same group of statements:

```

switch (grade) {
    case 'A': case 'B': case 'C': case 'D':
        printf("Passing");
        break;
    case 'F': printf("Failing");
        break;
    default: printf("Illegal grade");
}

```

The last statement in each group is usually `break`, but this isn't required. Without `break` (or some other jump statement), control will flow from one case into the next. Consider the following switch statement:

```

switch (grade) {
    case 'A': printf("Excellent");
    case 'B': printf("Good");
    case 'C': printf("Average");
    case 'D': printf("Poor");
    case 'F': printf("Failing");
    default: printf("Illegal grade");
}

```

If the value of `grade` is 'A', the message printed is

ExcellentGoodAveragePoorFailingIllegal grade

since control flows through all five cases plus the default case.

*Warning:* Forgetting to end each case with `break` is a common error—one that the compiler can't detect. Although omitting `break` is sometimes done intentionally to allow several cases to share code, it's usually just an oversight.

A switch statement isn't required to have a default case. If it's missing and the value of the controlling expression doesn't match any of the case labels, control passes to the next statement after the switch.

## The while Statement

---

The while statement is one of C's three iteration statements. The while statement has the form

```
while ( expression ) statement
```

The expression in parentheses is the *controlling expression*; the statement after the parentheses is the *loop body*.

When a while statement is executed, the controlling expression is evaluated first. If its value is nonzero, the loop body is executed and the expression is tested again. The process continues in this fashion—first testing the controlling expression, then executing the loop body—until the expression has a zero value. Since the controlling expression is tested *before* the loop body is executed, it's possible that the body isn't executed even once.

Here's an example of a while statement:

```
i = 10;
while (i > 0) {
    printf("T minus %d and counting\n", i);
    --i;
}
```

When the while statement is executed, the variable *i* has the value 10. Since 10 is greater than 0, the loop body is executed, causing the message T minus 10 and counting to be printed and *i* to be decremented. The condition *i* > 0 is now tested again. Since 9 is greater than 0, the loop body is executed once more. This process continues until the message T minus 1 and counting is printed and *i* becomes 0. The test *i* > 0 now fails, causing the loop to terminate.

The previous while statement could have been written in a number of different ways. For example, we can eliminate the braces in the loop body by decrementing *i* inside the call of `printf`:

```
i = 10;
while (i) printf("T minus %d and counting\n", i--);
```

We've also shortened the controlling expression; the loop will now execute as long as *i* isn't equal to 0. Provided that *i* is never negative, the new test is equivalent to the old one.

A while statement won't terminate if the controlling expression never becomes zero. Sometimes, programmers deliberately create an infinite loop by using a nonzero constant as the controlling expression:

```
while (1) {
    ...
}
```

A while statement of this form will execute forever unless its body contains a statement that transfers control out of the loop or causes the program to terminate.

## The do Statement

---

The do statement is closely related to the while statement; in fact, the do statement is just a while statement whose controlling expression is tested *after* each execution of the loop body. The do statement has the form

```
do statement while ( expression );
```

When a do statement is executed, the loop body (the statement between do and while) is executed first, then the controlling expression is evaluated. If the value of the expression is nonzero, the loop body is executed again and then the expression is evaluated once more. Execution of the do statement terminates when the controlling expression has the value 0.

Let's rewrite our while statement example, using a do statement this time:

```
i = 10;
do {
    printf("T minus %d and counting\n", i);
    --i;
} while (i > 0);
```

When the `do` statement is executed, the loop body is first executed, causing the message `T minus 10 and counting` to be printed and `i` to be decremented. The condition `i > 0` is now tested. Since 9 is greater than 0, the loop body is executed a second time. This process continues until the message `T minus 1 and counting` is printed and `i` becomes 0. The test `i > 0` now fails, causing the loop to terminate.

As this example shows, the `do` statement frequently produces the same results as the `while` statement. The difference between the two is that the body of a `do` statement is always executed at least once; the body of a `while` statement may be skipped entirely if the controlling expression is 0 initially.

### **Example Program: Finding the Largest in a List of Numbers**

Let's write a program that finds the largest number in a list entered by the user. The program will repeatedly prompt the user to enter a number. When the user enters 0 or a negative number, the program will print the largest nonnegative number entered. For example, the program might produce the following output:

```
Enter a number: 60
Enter a number: 38.3
Enter a number: 4.89
Enter a number: 100
Enter a number: 75.2295
Enter a number: 0
```

The largest number entered was 100

Clearly we'll need some kind of loop, but should we use a `while` statement or a `do` statement? The `do` statement turns out to be the better choice. Here's a version of the program built around a `do` loop:

```
#include <stdio.h>

main()
{
    float max, x;

    max = 0.0;
    do {
        printf("Enter a number: ");
        scanf("%f", &x);
        if (x > max) max = x;
    } while (x > 0.0);
    printf("\nThe largest number entered ");
    printf("was %g\n", max);
}
```

To see why the `do` statement is the right choice, let's look at an equivalent `while` loop:

```

printf("Enter a number: ");
scanf("%f", &x);
while (x > 0.0) {
    if (x > max) max = x;
    printf("Enter a number: ");
    scanf("%f", &x);
}

```

The while statement tests the condition  $x > 0.0$  before the loop body has been executed even once, so we're forced to put extra calls of `printf` and `scanf` before the while statement to guarantee that `x` has a value to be tested.

## The for Statement

---

The for statement has the form

```

for ( expr1 ; expr2 ; expr3 ) statement

```

where `expr1`, `expr2`, and `expr3` are expressions.

The for statement is closely related to the while statement. In fact, any for statement can be replaced by the following statements:

```

expr1;
while (expr2) {
    statement
    expr3;
}

```

In other words, *expr1* is an initialization step to be performed before the loop is executed, *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero), and *expr3* is an operation to be performed at the end of each loop iteration.

Here's a simple example of a for statement:

```

for ( i = 10; i > 0; -i )
    printf("T minus %d and counting\n", i);

```

To see what this statement does, let's convert it to an equivalent while statement:

```

i = 10;
while ( i > 0 ) {
    printf("T minus %d and counting\n", i);
    -i;
}

```

When the for statement is executed, the variable `i` is initialized to 10, then `i` is tested to see if it's greater than 0. Since it is, the message T minus 10 and counting is printed, then `i` is decremented. The condition  $i > 0$  is then tested again. The loop body will be executed 10 times in all, with `i` varying from 10 down to 1.

Incidentally, the third expression in this for statement, `—i`, could be replaced by `i—` if desired. Since the first and third expressions in a for statement are executed as entire statements, their values are irrelevant—they're useful only

for their side effects. As a result, these two expressions are usually assignments or increment/decrement expressions.

### Omitting Expressions in a for Statement

Any or all of the three expressions in a for statement may be omitted. If the first expression is omitted, no initialization is performed before the loop is executed:

```
i = 10;
for (; i > 0; -i)
    printf("T minus %d and counting\n", i);
```

In this example, *i* has been initialized by a separate assignment, so we've omitted the first expression in the for statement. (Notice that the semicolon between the first and second expressions remains. The two semicolons must always be present, even when we've omitted some of the expressions.)

If the second expression is missing, the for statement doesn't terminate (unless stopped in some other fashion). For example, programmers often use the following for statement to establish an infinite loop:

```
for (;;) {
    ...
}
```

If we omit the third expression in a for statement, the loop body is responsible for making sure that the statement's second expression eventually becomes 0. Our example for statement could be written like this:

```
for (i = 10; i > 0;)
    printf("T minus %d and counting\n", i--);
```

To compensate for omitting the third expression, we've arranged for *i* to be decremented inside the loop body. As a result, this for statement is equivalent to the original one.

### The Comma Operator

On occasion, we might like to write a for statement with *two* initialization expressions or one that increments *two* variables each time through the loop. We can do this by using a *comma expression* as the first or third expression in the for statement.

A comma expression has the form

```
expr1 , expr2
```

where *expr1* and *expr2* are any two expressions. The comma operator allows us to "glue" two expressions together to form a single expression.

The comma operator is useful in situations where C requires a single expression, but we'd like to include two or more expressions. For example, suppose that we want to initialize two variables when entering a for statement. Instead of writing

```
sum = 0;
for (i = 1; i <= N; i++) sum += i;
```

we can write

```
for (sum = 0, i = 1; i <= N; i++) sum += i;
```

The expression `sum = 0, i = 1` first assigns 0 to `sum`, then assigns 1 to `i`. With additional commas, the `for` statement could initialize more than two variables.

A comma expression is evaluated in two steps: First, *expr1* is evaluated; its value, if any, is discarded. Second, *expr2* is evaluated; its value is the value of the entire expression. For example, suppose that the variables `i` and `j` have the values 1 and 5, respectively. When the comma expression `i++, i+j` is evaluated, `i` is first incremented, then `i+j` is evaluated, so the value of the expression is 7. (And, of course, `i` now has the value 2.) The precedence of the comma operator is less than that of all other operators, so there's no need to put parentheses around `i++` and `i+j`.

Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose, since its value is discarded.

Comma expressions are also useful in macros (see Chapter 10). Other than in `for` statements and macros, the comma operator is best used sparingly.

### **Example Program: Printing a Table of Squares (Revisited)**

The `for` statement in C is more powerful than the `for` statement in other common programming languages because C places no restrictions on the three expressions that control its behavior. In particular, the three expressions need not involve the same variable. As an example, consider the following program, a modification of the program in Chapter 2 that prints a table of squares:

```
#include <stdio.h>

main()
{
    int i, n, odd, square;

    printf("This program prints ");
    printf("a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    i = 1;
    odd = 3;
    for (square = 1; i <= n; odd += 2) {
        printf("%10d%10d\n", i, square);
        ++i;
        square += odd;
    }
}
```



The for statement in this program initializes one variable (square), tests another (i), and increments a third (odd), thereby illustrating the great flexibility of the for statement. i is the number to be squared, square is the square of i, and odd is the odd number that must be added to the current square to get the next square (allowing the program to compute consecutive squares without performing any multiplications).

## The break Statement

---

The break statement transfers control out of an enclosing while, do, for, or switch statement. In other words, break is a restricted form of goto.

We've already seen examples of how break is used in switch statements. In a while, do, or for loop, break is useful for terminating the loop prematurely. For example, suppose that we're writing a for statement that checks whether a number n is prime by successively dividing n by the numbers between 2 and  $n \div 1$ . We can break out of the loop as soon as any divisor is found; there's no need to try the remaining possibilities. After the loop has terminated, we can use an if statement to determine whether termination was premature (hence n isn't prime) or normal (n is prime):

```
for (divisor = 2; divisor < n; divisor++)
    if (n % divisor == 0) break;
if (divisor < n)
    printf("%d is not prime\n", n);
else
    printf("%d is prime\n", n);
```

The break statement is particularly useful for writing loops in which the exit test is in the middle rather than at the beginning or end. In the following for statement, for example, the exit test is in the middle:

```
for (;;) {
    printf("Enter a number (enter 0 to stop): ");
    scanf("%d", &n);
    if (n == 0) break;
    printf("%d cubed is %d\n", n, n*n*n);
}
```

When statements are nested, executing a break statement transfers control out of the innermost while, do, for, or switch that encloses the break. Consider the case of a switch statement nested inside a while statement:

```
while (...) {
    switch (...) {
        ...
        break;
        ...
    }
}
```

The break statement transfers control out of the switch statement, but not out of the while loop.

## The continue Statement

---

The continue statement, like the break statement, is a restricted form of the goto statement. However, break transfers control just *past* the end of a loop, while continue transfers control to a point just *before* the end of the loop body. With break, control leaves the loop; with continue, control remains inside the loop.

The following example illustrates a typical use of continue. This example reads a series of numbers and computes their sum. The loop terminates when ten nonzero numbers have been read. Whenever the number 0 is read, the continue statement is executed, skipping the rest of the loop body (the statements ++n; and sum += i;) but remaining inside the loop.

```
n = 0;
sum = 0;
do {
    scanf("%d", &i);
    if (i == 0) continue;
    ++n;
    sum += i;
} while (n < 10);
```

The continue statement can appear only inside while, do, and for statements.

## The goto Statement

---

The goto statement has the form

```
goto identifier ;
```

where the identifier is a *label*. The effect of executing a goto statement is to transfer control to the statement that follows the label, which must be in the same function as the goto statement itself.

Any statement may be preceded by a label:

```
identifier : statement
```

A statement may have more than one label.

If C didn't have a break statement, here's how we would use the goto statement to exit from a loop:

```
for (divisor = 2; divisor < n; divisor++)
    if (n % divisor == 0) goto done;
done:
if (divisor < n)
    printf("%d is not prime\n", n);
else
    printf("%d is prime\n", n);
```

The goto statement is rarely needed in everyday C programming. The break, continue, and return statements—which are essentially restricted goto

statements and the exit function (Chapter 7) are sufficient to handle most situations that might require a goto in other programming languages.

## The Null Statement

---

The null statement is just a semicolon. Here's an example:

```
i = 0; ; j = 1;
```

This line contains three statements: an assignment to i, a null statement, and an assignment to j.

The null statement is useful when a label must be placed at the end of a compound statement:

```
{
    ...
    goto end_of_stmt;
    ...
    end_of_stmt: ;
}
```

A label must always be followed by a statement; however, the statement may be null.

The null statement is also useful when the body of a loop is to be left empty. The classic case of a loop with an empty body is the following statement:

```
while ((ch = getchar()) == ' ');
```

This statement reads a character, stores it into the variable ch, then tests whether ch contains a blank. If so, the loop body (which is empty) is executed, then the loop test is performed once more, causing a new character to be read. In other words, the while statement repeatedly reads characters until it finds a nonblank character—the value of ch when the loop terminates.

Note: Some programmers prefer to put the semicolon on a separate line to draw attention to the fact that the loop body is empty:

```
while ((ch = getchar()) == ' ' )
;
```

### Warning:

**Accidentally putting a semicolon after the parentheses in an if, while, or for statement ends the statement prematurely; the compiler can't detect an error of this kind. See next section**

## Unintentional Null Statements

---

An accidental semicolon often causes unexpected problems in one of the following situations, which you should look out for because they are common mistakes

- In an if statement, putting a semicolon after the parentheses creates an if statement that apparently performs the same action regardless of the value of its controlling expression:

```
if (divisor == 0); /* error */
    printf("Error: Division by zero\n");
```

The call of printf isn't inside the if statement, so it's performed regardless of whether divisor is equal to 0.

- In a while statement, putting a semicolon after the parentheses may create an infinite loop:

```
i = 10;
while (i > 0); /* error */
{
    printf("T minus %d and counting\n", i);
    -i;
}
```

Another possibility is that the loop terminates, but the statement that should be the loop body is executed only once, after the loop has terminated:

```
i = 11;
while (-i > 0); /* error */
    printf("T minus %d and counting\n", i);
```

This example prints the message

T minus 0 and counting

- In a for statement, putting a semicolon after the parentheses causes the statement that should be the loop body to be executed only once:

```
for (i = 10; i > 0; i-); /* error */
    printf("T minus %d and counting\n", i);
```

This example also prints the message

T minus 0 and counting

# CHAPTER 8

## Pointers

Pointers are one of C's most important and most often misunderstood features. This chapter begins with an introduction to the basic properties of pointers, including a discussion of the address and indirection operators. The rest of the chapter explains how to use pointers to (1) write functions that modify their actual parameters and (2) access array elements. Chapter 13 covers several additional uses of pointers.

### Pointer Variables

---

A *pointer variable* is a variable whose value isn't an ordinary item of data, like an integer or character, but a *pointer* to some other entity (a variable, say).

For example, suppose that *i* is a variable of type `int` and *p* is a pointer variable. Assume that *i* has the value 1 and that *p* points to *i*. The following diagram depicts the situation:

To indicate that *p* points to *i*, we show the contents of *p* as an arrow directed toward *i*. (What *p* *really* contains is *i*'s address in memory. Most people find pointers easier to understand when they're shown as arrows, so we'll stick with that notation.)

Although the value of a pointer variable may change during program execution, the variable must always point to objects of the same type (the *referenced type*). The declaration of a pointer variable must specify the referenced type. In the following declaration, `int` is the referenced type; the pointer variable *p* can point only to objects of type `int`:

```
int *p;
```

The asterisk in front of *p* tells the compiler that *p* is a pointer, not an `int` variable itself.

Pointer variables can appear in declarations along with ordinary variables and arrays:

```
int i, j, a[10], b[20], *p, *q;
```

## The Address and Indirection Operators

Declaring a pointer variable creates space for the pointer, but doesn't make it point to an object. One way to initialize a pointer variable is to assign it the address of a variable, which we can do by using the `&` operator:

```
p = &i;
```

This statement makes `p` point to `i` by assigning the address of `i` to the variable `p`. The following diagram shows the relationship between `p` and `i`:

The assignment of `&i` to `p` doesn't affect the value of `i`.

`&` is called the *address operator*. The value of `&x`, where `x` is any variable, is the address of `x` in memory. In other words, applying `&` to `x` creates a pointer to `x`.

To gain access to the variable that a pointer points to, we can use the `*` operator (the *indirection operator*). If `p` is a pointer, then `*p` represents the variable to which `p` currently points. If `p` points to a variable `i`, then `*p` has the same value as `i`; furthermore, changing the value of `*p` also changes the value of `i`. (`*p` is an lvalue, so assignment to it is legal.) The following example illustrates the equivalence of `*p` and `i`; diagrams show the values of `p` and `i` at various points in the computation.

```
p = &i;
i = 1;
printf("%d\n", i);    /* prints 1 */
printf("%d\n", *p);   /* prints 1 */
*p = 2;
printf("%d\n", i);    /* prints 2 */
printf("%d\n", *p);   /* prints 2 */
```

The `*` operator is the inverse of the `&` operator. For example, the following statements are equivalent:

```
j = *&i;
j = i;
```

**Warning:** Never apply the indirection operator to an uninitialized pointer variable. If a pointer variable `p` hasn't been initialized, the value of `*p` is undefined:

```
int *p;

printf("%d", *p); /* prints garbage */
```

Assigning a value to `*p` is even worse; `p` might point anywhere in memory, so the assignment modifies some unknown memory location:

```
int *p;

*p = 1;
```

```
/* modifies an unknown */  
/* memory location */
```

The location modified by this assignment might belong to the program (perhaps causing it to behave erratically) or to the operating system (possibly causing a system crash).

If the TopSpeed C compiler detects a potential use of an uninitialized pointer variable `p`, it issues the warning “*Possible use of ‘p’ before definition.*”

## Pointer Assignment

---

Suppose that `i`, `j`, `p`, and `q` have been declared as follows:

```
int i, j, *p, *q;
```

The statement

```
p = &i;
```

is an example of *pointer assignment*; no “genuine” data is copied, just a pointer. Here’s another example of pointer assignment:

```
q = p;
```

This statement copies the contents of `p` into `q`, making `q` point to the same place as `p`:

Both `p` and `q` point to `i`, so `i` can be changed by assigning to either `*p` or `*q`:

```
*p = 1;  
*q = 2;
```

Any number of pointer variables may point to the same object.

Notice the difference between

```
q = p;
```

and

```
*q = *p;
```

The first statement is a pointer assignment; the second isn’t, as the following example shows:

```
p = &i;  
q = &j;  
i = 1;  
*q = *p;
```

The assignment `*q = *p` copies the value that `p` points to (the value of `i`) into the location that `q` points to (the variable `j`).

## Null Pointers

---

To indicate that a pointer variable doesn't currently point to an object, we can assign it the value `NULL`, which represents a *null pointer*. `NULL` is a macro defined in `<stddef.h>`, so programs that use `NULL` should contain the following line:

```
#include <stddef.h>
```

The headers `<locale.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>` also define `NULL`, so there's no need to include `<stddef.h>` if the program uses one of these.

It's often a good idea to initialize a pointer variable when declaring it:

```
int *p = NULL;
```

It is illegal to apply the indirection operator to a null pointer:

```
p = NULL;
i = *p;    /* illegal */
```

TopSpeed C will detect this error during program execution if the program was compiled with the `/rn` option.

## Using Pointers as Function Parameters

---

We saw in Chapter 7 that a variable supplied as an actual parameter in a function call is always protected against change. This property of C can be a nuisance if we want the function to be able to modify the actual parameter. In particular, we examined the `decompose` function, which needs to modify two of its parameters.

Pointers offer a solution to this problem: instead of providing the variable `x` as an actual parameter to a function, we supply `&x`, a pointer to `x`. We declare the corresponding formal parameter `y` to be a pointer. When the function is called, `y` will be given the value `&x`, hence `*y` (the object that `y` points to) will be `x`. Each appearance of `*y` in the body of the function is an indirect reference to `x`, allowing the function both to read `x` and to modify it.

To see this method in action, let's modify the `decompose` function by declaring the parameters `int_part` and `frac_part` to be pointers. The definition of `decompose` would look like this:

```
void decompose(
    float x,
    int *int_part,
    float *frac_part)
{
    *int_part = x; /* drops the fractional */
                  /* part of x */
    *frac_part = x - *int_part;
}
```



The declaration of `decompose` could be either

```
void decompose(  
    float x,  
    int *int_part,  
    float *frac_part);
```

or

```
void decompose(float, int *, float *);
```

We would call `decompose` in the following way:

```
decompose(3.14159, &i, &f);
```

Because of the `&` symbol in front of `i` and `f`, `decompose` is supplied with *pointers* to `i` and `f`, not the *values* of `i` and `f`.

When `decompose` is called, the value 3.14159 is copied into `x`, a pointer to `i` is stored in `int_part`, and a pointer to `f` is stored in `frac_part`:

The first assignment in the body of `decompose` converts the value of `x` to type `int` and stores it into the location pointed to by `int_part`. Since `int_part` points to `i`, the assignment puts the value 3 in `i`:

The second assignment fetches the value that `int_part` points to (the value of `i`), which is 3. This value is converted to type `float` and subtracted from `x`, giving .14159, which is then stored in the location that `frac_part` points to:

When `decompose` returns, `i` and `f` will have the values 3 and .14159, just we had originally wanted.

`scanf` is an example of a function whose parameters must be pointers. When we use `scanf`, we put the address operator `&` in front of each variable `x` in the parameter list so that `scanf` is given a *pointer* to `x`. Without the `&`, `scanf` would be supplied with the *value* of `x`, which would be of no use to it. The pointer tells `scanf` where to put the value that it reads.

Warning:

Failing to pass a pointer to a function when one is expected can have disastrous results. For example, suppose that we call `decompose` without the `&` symbol in front of `i` and `f`:

```
decompose(3.14159, i, f);
```

`decompose` is expecting pointers as its second and third parameters, but it's been given the *values* of `i` and `f` instead. `decompose` has no way to tell the difference, so it will use the values of `i` and `f` as though they were pointers. When `decompose` stores values into `*int_part` and `*frac_part`, it will be writing to (apparently) random memory locations instead of modifying `i` and `f`. If a prototype for `decompose` precedes the call, TopSpeed C issues the “*Pointer conversion*” warning.

## Example Program: Finding Largest and Smallest Numbers in a List

To illustrate the technique of passing pointers to a function, we'll write a function named `max_min` that finds the largest and smallest elements in an array, then returns these values using two pointers supplied as parameters. `max_min` has the following declaration:

```
void max_min(int a[], int n, int *max, int *min);
```

A call of `max_min` might have the following form:

```
max_min(b, N, &big, &small);
```

`N` is the length of the array `b`; `big` and `small` are ordinary integer variables. When `max_min` finds the largest element of `b`, it stores the value in `big` by assigning it to `*max`. (Since `max` points to `big`, an assignment to `*max` will modify the value of `big`.) `max_min` stores the smallest element of `b` in `small` by assigning it to `*min`.

To test `max_min`, we'll write a program that reads ten numbers into an array, passes the array to `max_min`, and prints the results. The output of the program will look like this:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

Here's the complete program:

```
#include <stdio.h>
#define N 10
void max_min(
    int a[],
    int n,
    int *max,
    int *min);
main()
{
    int b[N], i, big, small;

    printf("Enter %d numbers: ", N);
    for (i = 0; i < N; i++)
        scanf("%d", &b[i]);
    max_min(b, N, &big, &small);
    printf("Largest: %d\n", big);
    printf("Smallest: %d\n", small);
}

void max_min(
    int a[],
    int n,
    int *max,
    int *min)
{
    int i;
    *max = *min = a[0];
    for (i = 1; i < n; i++) {
        if (a[i] > *max) *max = a[i];
        if (a[i] < *min) *min = a[i];
    }
}
```

## Using const to Protect Parameters

When a function has a pointer `p` as a formal parameter, the function can normally change either `*p` or `p` itself. For example, the following function can modify the integer that `p` points to; it can also modify `p` by making it point to another variable:

```
void f(int *p)
{
    int j;

    *p = 0;    /* legal */
    p = &j;    /* legal */
}
```

To protect `*p` (allowing `f` to examine `*p` but not change it), we can put the word `const` in the declaration of `p`, just *before* the specification of its type:

```
void f(const int *p)
{
    int j;

    *p = 0;    /* illegal */
    p = &j;    /* legal */
}
```

To protect `p` itself, we would put `const` *after* the type specification:

```
void f(int * const p)
{
    int j;

    *p = 0;    /* legal */
    p = &j;    /* illegal */
}
```

## Pointer Arithmetic

---

Pointers can point to array elements, not just ordinary variables. For example, suppose that `a` and `p` have been declared as follows:

```
int a[10], *p;
```

We can make `p` point to `a[0]` by writing

```
p = &a[0];
```

We can now access `a[0]` through `p`; for example, we can store the value 5 in `a[0]` by writing

```
*p = 5;
```

By itself, making `p` point to an element of `a` isn't particularly exciting. However, by performing *pointer arithmetic* on `p`, we can access any of the other elements of `a`.

C supports three (and only three) forms of pointer arithmetic:

- *Adding an integer to a pointer.* If *p* points to the array element *a[i]*, then *p + j* points to *a[i+j]* (provided, of course, that *a[i+j]* exists). For example:

```
int a[10], *p, *q, i;

p = &a[3]; /* p points to a[3] */
q = p + 1; /* q points to a[4] */
p += 4;    /* p points to a[7] */
```

- *Subtracting an integer from a pointer.* If *p* points to the array element *a[i]*, then *p - j* points to *a[i-j]*. For example:

```
p = &a[7]; /* p points to a[7] */
q = p - 1; /* q points to a[6] */
p -= 4;    /* p points to a[3] */
```

- *Subtracting two pointers.* If *p* points to *a[i]* and *q* points to *a[j]*, then *p - q* is equal to *i - j*. For example:

```
p = &a[5]; /* p points to a[5] */
q = &a[1]; /* q points to a[1] */
i = p - q; /* i is 4 */
i = q - p; /* i is -4 */
```

Arithmetic on a pointer *p* is defined only when *p* points to an array element. Furthermore, two pointers can be subtracted only when both point to elements in the same array.

Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=). For example, after the assignments

```
p = &a[5];
q = &a[1];
```

the value of the expression *p <= q* is 0 and the value of *p >= q* is 1.

## Using a Pointer to Step through an Array

One of the most common uses of pointer arithmetic is stepping through an array by repeatedly incrementing a pointer variable. The following program fragment, which sums the elements of an array *a*, illustrates this technique. In this example, the pointer variable *p* initially points to *a[0]*. Each time through the loop, *p* is incremented; as a result, it points to *a[1]*, then *a[2]*, and so forth. The loop terminates when *p* steps past the last element of *a*.

```
#define N 100

int a[N], sum, *p;

sum = 0;
for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

Notice the condition *p < &a[N]* in the for statement. In ANSI C, it's legal to apply the address operator to *a[N]*, even though this element doesn't exist (*a* is indexed from 0 to *N - 1*). This use of *a[N]* is perfectly safe, since the loop doesn't attempt to examine its value. The body of the loop will be

executed with `p` equal to `&a[0]`, `&a[1]`, ..., `&a[N-1]`, but when `p` is equal to `&a[N]`, the loop will terminate.

Why step through an array using a pointer instead of using a variable to subscript the array? The reason most often cited is that using a pointer can save execution time. However, this depends on the implementation—some C compilers actually produce better code when subscripting is used.

## Using an Array Name as a Pointer

---

Pointer arithmetic is one way in which arrays and pointers are related, but it's not the only way. Here's another key relationship: *the name of an array is actually a pointer to the first element in the array.*

For example, suppose that `a` is declared as follows:

```
int a[10];
```

Since `a` is a pointer to the first element in the array, we can modify `a[0]` by storing a value into `*a`:

```
*a = 0;      /* stores 0 in a[0] */
```

We can modify `a[1]` through the pointer `a + 1`:

```
*(a+1) = 1; /* stores 1 in a[1] */
```

In general, the expression `*(a+i)` is equivalent to `a[i]`; in other words, array subscripting is just a form of pointer arithmetic. Because of this equivalence, the expression `i[a]` is legal and has the same meaning as `a[i]`. (But please don't use `i[a]` in programs; it's terribly confusing.)

Notice that writing `a` is equivalent to writing `&a[0]`. Consequently, we can simplify the for statement that we used to sum the elements of `a` in a previous example:

```
for (p = a; p < &a[N]; p++)
    sum += *p;
```

There's one important difference between an array name and a pointer variable: an array name is a "constant pointer"; it can't be changed. The following is illegal:

```
*a = 0;
++a;      /* illegal */
*a = 1;
```

To get this effect, we must use a pointer variable:

```
p = a;
*p = 0;
++p;
*p = 1;
```

## Using a Pointer as an Array Name

If an array name can be used as a pointer, can we subscript a pointer as though it were an array name? The answer is yes, as the following example shows:

```
#define N 100

int a[N], i, sum, *p;

p = a;
sum = 0;
for (i = 0; i < N; i++)
    sum += p[i];
```

Writing `p[i]` is the same as writing `*(p+i)`, so it's perfectly legal to subscript a pointer. Although this ability may seem to be of little interest, we'll see in Chapter 13 that it's actually quite useful.

## Using an Array Name as a Parameter

---

When an array name is supplied as an actual parameter in a function call, the corresponding formal parameter is assigned a pointer to the first element in the array. For example, consider the following function, which returns the largest element in an array of integers:

```
int find_largest(int a[], int n)
{
    int i, max;

    max = a[0];
    for (i = 1; i < n; i++)
        if (a[i] > max) max = a[i];
    return max;
}
```

Suppose that we call `find_largest` as follows:

```
i = find_largest(b, N);
```

This call causes a pointer to the first element of `b` to be assigned to `a`; the array itself isn't copied.

The fact that an array parameter is treated as a pointer has several consequences:

- An ordinary variable used as an actual parameter in a function call is copied by the function; any changes to the corresponding formal parameter don't affect the variable. In contrast, an array used as an actual parameter isn't protected against change, since no copy is made of the array itself. For example, the following function modifies an array by storing zero in each of its elements:

```
void store_zeroes(int a[], int n)
{
    int i;

    for (i = 0; i < n; i++) a[i] = 0;
}
```

To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:

```
int find_largest(const int a[], int n)
{
    ...
}
```

If `const` is present, the compiler will check that no assignment to an element of `a` appears in the body of `find_largest`.

- The time required to call a function with an array parameter doesn't depend on the size of the array. There's no penalty for passing a large array to a function, since no copy of the array is made.
- An array parameter can be declared as a pointer if desired. For example, the definition of `find_largest` could declare `a` to be a pointer:

```
int find_largest(int *a, int n)
{
    ...
}
```

Declaring `a` to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.

## CHAPTER 9

### Strings

Strings are used extensively in many, if not most, C programs. This chapter explains the rules that govern the use of string literals and string variables. It also describes several of the string-handling functions in the C library and shows how two of these functions might be written.

#### String Literals

---

A *string literal* is a sequence of characters enclosed by double quotes:

```
"this is a string"
```

We first encountered string literals in Chapter 2; they often appear as format strings in calls of `printf` and `scanf`.

String literals may contain escape sequences. For example, the following string literal contains a newline character:

```
"Top line\nBottom line"
```

When printed, this string has the following appearance:

```
Top line
Bottom line
```

The `\` escape sequence is particularly important, since it allows a string literal to contain the double quote character. For example, the string `"\"Hello\""` contains the seven characters `"`, `H`, `e`, `l`, `l`, `o`, and `"`.

**Warning:** Be careful when using octal and hexadecimal escape sequences in string literals. An octal escape ends after three digits or with the first non-octal character. For example, the string literal `"\1234"` contains two characters (`'\123'` and `'4'`), and the literal `"\189"` contains three characters (`'\1'`, `'8'`, and `'9'`). A hexadecimal escape, on the other hand, isn't limited to three digits; it doesn't end until the first non-hex character.

A string literal may be continued over more than one line by writing `\` immediately followed by a newline character (neither the `\` nor the newline character is part of the string):

```
"This is a single long string that happens to \
be split over two lines"
```

Unfortunately, the string must continue at the beginning of the next line, which can destroy the indented structure of a program.



ANSI C provides an alternative way to deal with long string literals. When two or more string literals are adjacent (separated only by white space), ANSI C requires the compiler to join them into a single string. This rule allows us to break up a string literal that is too long to fit comfortably on a single line. For example, instead of using two calls of `printf` to print the message “Welcome to TopSpeed C, a product of Jensen & Partners International”, we can use a single call:

```
printf("Welcome to TopSpeed C, a product of "
      "Jensen & Partners International\n");
```

## Using String Literals

---

A string literal is stored as an array of characters. String literals have one special property, however: the compiler places a *null character* (usually written as the octal escape `\0`) at the end of each literal:

```
"abc" /* contains the characters 'a', 'b',
      /* 'c', and '\0' */
""    /* contains only '\0' */
```

Because of this property, a string literal of length  $n$  actually occupies  $n + 1$  characters of storage. An important fact to remember.

A string literal can be used in the same way as an array, except that no attempt should be made to change any of its elements. Like an array, which is represented by a pointer to its first element, a string literal is represented by a pointer to its first character. The following examples illustrate two uses of string literals:

```
char *p;

p = "abc";
```

`p` is assigned a pointer to the first character of “abc”.

```
char c;

c = "abc"[1];
```

`c` is assigned the letter `b`. (The elements of the array “abc” are indexed from 0 to 2.)

A string literal of length 1 isn’t the same as a character constant. A string literal of length 1 (“a”, for example) is represented by a *pointer* to a memory location that contains the character `a`. A character constant (`'a'`, for example) is represented by an *integer* (in TopSpeed C, the ASCII code for the character).

**Warning:** Don’t ever use a character when a string is required (or vice-versa). The call

```
printf("\n");
```

is legal, because `printf` expects a string as its first parameter, but the following call isn't:

```
printf('\n'); /* illegal */
```

The TopSpeed C compiler issues the warning “*Pointer conversion*” when it detects an error of this kind.

## String Variables

---

A string variable is just a one-dimensional array of characters. For example, the following string variable can store a string of up to 80 characters:

```
#define STR_LEN 80

char str[STR_LEN+1];
```

When declaring an array of characters that will be used to hold a string, always make the array one character longer than the string, because of the C convention that every string is terminated by the null character.

*Warning:* Failing to leave room for the null character may cause unpredictable results when the program is executed, since all functions in the C library assume that strings are null-terminated.

### Initializing a String Variable

The following examples illustrate the initialization of string variables:

```
char date1[7] = "June 14"; /* example 1 */
char date2[8] = "June 14"; /* example 2 */
char date3[9] = "June 14"; /* example 3 */
```

In example 1, `date1` has no room for the null character, so it isn't copied into the array:

In example 2, the array contains room for the null character:

In example 3, the array is longer than the initializer, so the final character of the array is initialized to `\0`:

Example 3 requires further explanation. Technically, “June 14” is an abbreviated array initializer. The declaration in example 3 is equivalent to

```
char date3[9] = {'J', 'u', 'n', 'e', ' ', '1', '4', '\0'};
```

We saw in Chapter 6 that when an array initializer is shorter than the array itself, the remaining elements are initialized to 0, which in the case of a character array is the same as `\0`.

The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char greeting[] = "Welcome to TopSpeed C";
```

This is the same as declaring `greeting` to have length 22, since the string "Welcome to TopSpeed C" has length 21.

## **Declaring a Pointer to a String**

Instead of declaring an array to *contain* a string, we have a second option: declare a variable that *points* to the string. The following example declares a pointer variable named `greeting` and makes it point to the string "Welcome to TopSpeed C":

```
char *greeting = "Welcome to TopSpeed C"; /* pointer version */
```

This kind of declaration is often confused with the similar-looking declaration

```
char greeting[] = "Welcome to TopSpeed C"; /* array version */
```

Both declarations allocate 22 characters of storage, place the message Welcome to TopSpeed C (terminated by the null character) in these 22 characters, and make the variable `greeting` point to the first character. However, there are some subtle differences between the two declarations:

- In the array version, the string that `greeting` points to can be modified, like the elements of any array. In the pointer version, the string can't be modified, because it's a string literal that `greeting` just happens to point to. (More accurately, the string *shouldn't* be modified; the compiler can't always detect attempts at modification.)
- In the array version, `greeting` is an array name, so it can't be made to point elsewhere. In the pointer version, `greeting` is a pointer variable, so it can be made to point to some other string later in the program.

**Warning:** The following declaration doesn't allocate space for a string:

```
char *greeting;
```

Don't confuse this declaration with any of the following:

```
char greet[22];  
char greet[22] = "Welcome to TopSpeed;  
char greet[] = "Welcome to TopSpeed;  
char *greet = "Welcome to TopSpeed;
```

All four of these declarations allocate 22 characters of storage and make `greeting` point to the first character. In contrast, the declaration

```
char *greet;
```

allocates space only for the pointer `greeting`. Don't attempt to use `greeting` to store any characters without first making it point to an array. Otherwise, the characters may be written into

memory at some unknown location, causing unpredictable (and often disastrous) results. If the TopSpeed C compiler detects an attempt to use `greet` before it has been initialized, the compiler issues the warning “*Possible use of ‘greeting’ before definition.*”

## Reading and Writing Strings

---

The `%s` conversion specification allows `scanf` and `printf` to read and write strings:

```
char str[STR_LEN+1];

scanf("%s", str);    /* read a string */
printf("%s", str);   /* write a string */
```

**Note:** There’s no need to put the `&` operator in front of `str` in the call of `scanf`; since `str` is an array name, it’s already a pointer.

When `scanf` is called, it will first skip white space, then read characters and store them into `str` until it encounters a white-space character. `scanf` always stores a null character at the end of the string.

A string read using `scanf` will never contain white space. Consequently, `scanf` won’t usually read a full line of input; a newline character will cause `scanf` to stop reading, but so will a space or tab character.

`printf` writes the characters in a string one by one until it encounters a null character. (If we’ve neglected to put a null character at the end of the string, `printf` continues past the end until it eventually finds a null character somewhere in memory.)

C provides another way to read and write strings: the `gets` and `puts` functions. Like `scanf`, `gets` reads input characters into an array, then puts a null character at the end of the string. In other respects, however, `gets` is somewhat different from `scanf`:

- `gets` doesn’t skip white space before starting to read the string (`scanf` does).
- `gets` reads until it finds a newline character (`scanf` stops at any white-space character).

To see the difference between `scanf` and `gets`, consider the following program fragment:

```
char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");
scanf("%s", sentence);
```

Suppose that after the prompt

Enter a sentence:

the user enters the line

To C, or not to C: that is the question.

scanf will store the string “To” into sentence. Now suppose that we replace scanf by gets:

```
gets(sentence);
```

When the user enters the same input as before, gets will store the string

“ To C, or not to C: that is the question.”

into sentence.

**Warning:** scanf and gets have no way to detect the end of an input array. As a result, they may store characters past the end of the array, causing the program to behave erratically. scanf can be made safer by using the conversion specification `%ns` instead of `%s`, where *n* is a number indicating the maximum number of characters to be read. The function `fgets` (see Chapter 14) can be used instead of gets for greater safety.

puts is similar to printf, but not identical. Suppose that we call puts as follows:

```
puts(str);
```

puts writes the characters in str up to the first null character; it then writes a newline character. (printf, of course, doesn’t automatically add a newline character after it writes a string.)

Since gets and puts are more specialized than scanf and printf, the former are usually faster.

## Accessing the Characters in a String

---

Because of the close relationship between arrays and pointers, strings can be accessed either by array subscripting or by pointer reference. For example, suppose that we’re writing a function that counts the number of spaces in a string *s*. Treating *s* as a array, we could write

```

int count_spaces(const char s[])
{
    int count, i;

    count = 0;
    for (i = 0; s[i] != '\0'; i++)
        if (s[i] == ' ') count++;
    return count;
}

```

Alternatively, we could use a pointer to “walk through” the string:

```

int count_spaces(const char *s)
{
    int count;

    count = 0;
    for (; *s != '\0'; s++)
        if (*s == ' ') count++;
    return count;
}

```

We’ve included `const` in the declaration of `s` to indicate that `count_spaces` doesn’t change the string that `s` points to. Notice that using `const` doesn’t prevent us from modifying `s` in the second version of `count_spaces`. Note also that `s` is a copy of the actual parameter, so incrementing `s` doesn’t affect the actual parameter.

The `count_spaces` function raises several questions about the use of strings:

- *Is it better to use array operations or pointer operations to access the characters in a string?* We’re free to use whichever is more convenient; we can even mix the two. In the second version of `count_spaces`, treating `s` as a pointer simplifies the function slightly by removing the need for the variable `i`.
- *Should a string parameter be declared as an array or as a pointer?* The two versions of `count_spaces` illustrate the options: the first version of `count_spaces` declares `s` to be an array; the second version declares `s` to be a pointer. Actually, there’s no difference between the two declarations—recall from Chapter 8 that the compiler treats a formal parameter of an array type as though it had been declared as a pointer.
- *Does the type of the formal parameter affect what can be supplied as an actual parameter?* No; for example, when `count_spaces` is called, the actual parameter can be an array name, a pointer variable, or a string literal—`count_spaces` can’t tell the difference.

### **Example Program: Reversing a String**

As an example of string processing, we’ll write a program that reads a string and then prints its reversal. The program will read the string one character at a time, stopping when it encounters the newline character. It will then step through the string in reverse order, printing characters one at a time.

Let's look at two versions of the program. The first uses array subscripting to access characters in the string.

```
#include <stdio.h>

#define STR_LEN 80/* maximum length */
                /* of string      */

main()
{
    char str[STR_LEN];/*no null character*/
                    /*needed        */
    int i;

    printf("Enter a string: ");
    for (i = 0; i < STR_LEN; i++) {
        str[i] = getchar();
        if (str[i] == '\n') break;
    }

    printf("Reversal is: ");
    for (-i; i >= 0; i-)
        putchar(str[i]);
    putchar('\n');
}
```

Two points to note: (1) Since the program processes the string one character at a time, never using the string as a whole, there's no need to put a null character at the end. (2) To avoid going outside the bounds of the str array, the program stops reading after STR\_LEN characters have been read.

Here's an equivalent program, this time using a pointer to process the string:

```
#include <stdio.h>

#define STR_LEN 80/* maximum length */
                /* of string      */

main()
{
    char str[STR_LEN];/*no null character*/
                    /*needed        */
    char *p;

    printf("Enter a string: ");
    for (p = str; p < &str[STR_LEN]; p++) {
        *p = getchar();
        if (*p == '\n') break;
    }

    printf("Reversal is: ");
    for (-p; p >= str; p-)
        putchar(*p);
    putchar('\n');
}
```

## Using the C String Library

---

The C language provides little support for strings. Strings are treated as arrays, so they're restricted in the same ways as arrays. In particular, they can't be copied or compared.

*Warning:* Attempts to copy or compare two strings using C's built-in operators will fail. For example, suppose that `str1` and `str2` are declared as follows:

```
char str1[10], str2[10];
```

Assignment of one array to the other is illegal:

```
str1 = str2; /* illegal */
```

C interprets this statement as an (illegal) pointer assignment. Attempting to compare arrays using a relational or equality operator probably won't produce the desired result

```
if (str1 == str2)
    ... /* may produce the */
        /* wrong result */
```

This statement compares `str1` and `str2` as *pointers*; it doesn't compare the contents of the two arrays.

Fortunately, the C library provides a rich set of functions for performing operations on strings. Declarations for these functions reside in the header `<string.h>`; programs that need string operations should contain the following line:

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

The functions in `<string.h>` have formal parameters of type `char *`, allowing each actual parameter to be a character array, a pointer of type `char *`, or a string literal (since all three are pointers to the first character in a string). However, when a formal parameter isn't protected (the word `const` isn't present), the actual parameter shouldn't be a string literal.

There are many functions in `<string.h>`; we'll discuss just four of them. In subsequent examples, assume that `s5` and `s10` have been declared as follows:

```
char s5[6], s10[11];
```

### **The strcpy Function**

The `strcpy` function has the following declaration:

```
char *strcpy(char *s1, const char *s2);
```



strcpy copies the string s2 into the string s1. (To be precise, we should say “strcpy copies the string pointed to by s2 into the array pointed to by s1.”) That is, strcpy copies characters one at a time from s2 to s1 up to (and including) the first null character in s2. strcpy returns s1 (a pointer to the destination string). The string pointed to by s2 isn’t modified, so it has been declared const.

The existence of strcpy compensates for the fact that we can’t use assignment to copy a string directly. For example, suppose that we want to store the string “abcd” in s5. We can’t use the assignment

```
s5 = "abcd"; /* illegal */
```

because s5 is an array name and can’t appear on the left side of an assignment. Instead, we can call strcpy:

```
strcpy(s5, "abcd"); /* s5 now contains "abcd" */
```

Similarly, we can’t assign s5 to s10 directly, but we *can* call strcpy:

```
strcpy(s10, s5); /* s10 now contains "abcd" */
```

strcpy is usually called as a statement, causing its return value to be discarded. On occasion, however, it’s useful to call strcpy as part of a larger expression in order to use its return value. For example, we can chain together a series of strcpy calls to get the same effect as a multiple assignment:

```
strcpy(
    s10,
    strcpy(s5, "abcd")); /* both s10 and s5
                        /* now contain "abcd" */
```

### Warning:

**strcpy has no way to check that the string pointed to by s2 will actually fit in the array pointed to by s1. Suppose that s1 points to an array of length  $n$ . If the string that s2 points to has no more than  $n - 1$  characters, then the copy will succeed. But if s2 points to a longer string, the result is unpredictable. (Since strcpy always copies up to the first null character, it will continue copying past the end of the array that s1 points to. Whatever is stored in memory after that array will be overwritten.)**

### The **strcat** Function

The strcat function has the following declaration:

```
char *strcat(char *s1, const char *s2);
```

strcat appends the contents of the string s2 to the end of the string s1; it returns s1.

Here are some examples of strcat in action:

```
strcpy(s10, "abc");
strcat(s10, "def"); /* s10 now contains */
/*"abcdef" */
strcpy(s10, "abc");
strcpy(s5, "def");
strcat(s10, s5); /* s10 now contains */ /* "abcdef"
*/
```

As with `strcpy`, the value returned by `strcat` is normally discarded. The following example shows how the return value might be used:

```
strcpy(s10, "abc");
strcpy(s5, "def");
strcat(s10, strcat(s5, "ghi"));
/* s10 now contains */
/* "abcdefghi"; */
/* s5 contains */
/* "defghi" */
```

## The strcmp Function

The `strcmp` function has the following declaration:

```
int strcmp(const char *s1, const char *s2);
```

`strcmp` compares the strings `s1` and `s2`, returning a value less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2`. For example, to see if `s5` is less than `s10`, we'd write

```
if (strcmp(s5, s10) < 0) ...
```

`strcmp` considers `s1` to be less than `s2` if either one of the following conditions is satisfied:

- The first  $i$  characters of `s1` and `s2` match, but the  $(i + 1)$ st character of `s1` is less than the  $(i + 1)$ st character of `s2`. For example, "abc" is less than "bcd", and "abd" is less than "abe".
- All characters of `s1` match `s2`, but `s1` is shorter than `s2`. For example, "abc" is less than "abcd".

## The strlen Function

The `strlen` function has the following declaration:

```
int strlen(const char *s);
```

Actually, `strlen` returns a value of type `size_t` instead of `int`, but this technicality need not concern us here.

`strlen` returns the length of a string `s`. More precisely, `strlen` returns the number of characters in `s` up to, but not including, the first null character. For example:

```
i = strlen("abc"); /* i is now 3 */
i = strlen(""); /* i is now 0 */
strcpy(s5, "abc");
i = strlen(s5); /* i is now 3 */
```

The last example illustrates an important point: When given an array as its parameter, `strlen` doesn't measure the length of the array itself; instead, it returns the length of the string stored inside the array.

### **Example Program: Finding Largest and Smallest Words in a List**

To illustrate the use of the C string library, we'll write a program that finds the "smallest" and "largest" words in a list. That is, the program will read a series of words entered by the user and determine which words would come first and last if the words were listed in dictionary order. The program will stop accepting input when the user enters a four-letter word. We assume that no word is more than 20 letters long. An interactive session with the program might look like this:

```
Enter word: dog
Enter word: baboon
Enter word: rabbit
Enter word: walrus
Enter word: cat
Enter word: fish

Smallest word: baboon
Largest word: walrus
```

Here's the program:

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

#define WORD_LEN 20

main()
{
    char smallest_word[WORD_LEN+1],
        largest_word[WORD_LEN+1],
        current_word[WORD_LEN+1];

    printf("Enter word: ");
    scanf("%s", current_word);
    strcpy(
        smallest_word,
        strcpy(largest_word, current_word));
    while (strlen(current_word) != 4) {
        printf("Enter word: ");
        scanf("%s", current_word);
        if (strcmp(current_word, smallest_word) < 0)
            strcpy(smallest_word, current_word);
        if (strcmp(current_word, largest_word) > 0)
            strcpy(largest_word, current_word);
    }
    printf(
        "\nSmallest word: %s\n", smallest_word);
    printf("Largest word: %s\n", largest_word);
}
```

## Writing String Functions

---

A good way to develop proficiency in string manipulation is to write some of the string functions in the C library. In the next two sections, we'll develop versions of the `strlen` and `strcat` functions.

### Writing the `strlen` Function

Here's a straightforward version of the `strlen` function:

```
int strlen(const char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++) n++;
    return n;
}
```

The pointer `s` moves across the string from left to right; the variable `n` keeps track of how many characters have been seen so far. When `s` finally points to a null character, `strlen` returns the value of `n`.

As an exercise, let's see if we can simplify the definition of `strlen`. First, we notice that `n` can be initialized in its declaration:

```
int strlen(const char *s)
{
    int n = 0;

    for (; *s != '\0'; s++) n++;
    return n;
}
```

Next, we notice that the condition `*s != '\0'` is the same as `*s != 0`, because the integer value of the null character is 0. But the condition `*s != 0` is the same as the condition `*s`; both are true if `*s` isn't equal to 0. This observation leads to our next version:

```
int strlen(const char *s)
{
    int n = 0;

    for (; *s; s++) n++;
    return n;
}
```

We now notice that incrementing `s` can be combined with testing that `*s` isn't zero:

```
int strlen(const char *s)
{
    int n = 0;
    for (; *s++;) n++;
    return n;
}
```

Replacing the for statement by a while statement, we arrive at the following version of strlen:

```
int strlen(const char *s)
{
    int n = 0;

    while (*s++) n++;
    return n;
}
```

Since this version is so brief, it might seem to be the best possible. Actually, we can get a faster version of strlen by eliminating the variable n:

```
int strlen(const char *s)
{
    const char *p = s;

    while (*s++);
    return s - p - 1;
}
```

This version computes the length of the string by locating the position of the null character, then subtracting from it the position of the first character in the string. Because of the postfix increment operator in the while loop, s actually ends up pointing just past the null character, hence the subtraction of 1 in the return statement. The improvement in speed comes from not having to increment n inside the while loop.

The concise style used in these examples is popular with many C programmers. Mastering this style of programming is important for understanding programs written by others.

## **Writing the strcat Function**

Here's one version of the strcat function:

```
char *strcat(char *s1, const char *s2)
{
    char *p;

    p = s1;
    while (*p != '\0') p++;
    while (*s2 != '\0') {
        *p = *s2;
        p++;
        s2++;
    }
    *p = '\0';
    return s1;
}
```

This function has two steps: (1) Locate the null character at the end of the string s1 and make p point to it. (2) Copy characters one by one from s2 to the positions just after the end of s1.

The first two statements in the function implement step (1). `p` is set to point to the first character in `s1`. Then `p` is incremented within a while statement as long as it doesn't point to a null character.

The second while statement implements step (2). The loop body copies one character from where `s2` points to where `p` points, then increments both `p` and `s2`. The loop terminates when `s2` points to the null character.

By a process similar to the one we used for `strlen`, we can simplify the definition of `strcat`, arriving at the following version:

```
char *strcat(char *s1, const char *s2)
{
    char *p = s1;

    while (*p++);
    -p;
    while (*p++ = *s2++);
    return s1;
}
```

Notice that `p` is decremented after the first while statement, because the post-increment in the while statement has caused `p` to end up pointing just past the null character at the end of `s1`. Also notice that the second while loop copies the null character at the end of `s2`, so we don't need a separate statement for this purpose.

# CHAPTER 10

## The Preprocessor

One unusual feature of C is the preprocessing phase that precedes the actual compilation of a program. (For efficiency, preprocessing may actually be integrated with compilation, but from the programmer's point of view they appear to be separate.) During preprocessing, the text of the program is modified in accordance with preprocessing directives that are embedded in the program. In this chapter, we'll look at the various directives that can be given to the preprocessor. We'll also see how a C program can be divided into several files and how the `#include` directive can be used to share information among the files.

### Preprocessing Directives

---

There are three major preprocessing operations:

- *Macro definition.* The `#define` directive can be used to define both simple and parameterized macros; the `#undef` directive removes a macro definition.
- *File inclusion.* The `#include` directive causes the contents of a specified file to be included in a program.
- *Conditional compilation.* The `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` directives allow blocks of text to be either included in or excluded from a program, depending on conditions that can be tested by the preprocessor.

Several other preprocessing directives (`#line`, `#error`, and `#pragma`) exist, but these are rather specialized; see the *Language Reference* for information.

Before we take a closer look at these operations, let's review several general rules that apply to all preprocessing directives:

- *Directives always begin with the # symbol.* The `#` symbol need not be at the beginning of a line, as long as only white space precedes it. After the `#` comes the name of the directive, followed by any other information the directive requires.
- *Directives may contain extra spaces and horizontal tab characters between symbols.* For example, the following directive is legal:  

```
#    define    N    100
```
- *Directives always end at the first newline character, unless*

*explicitly continued.* To continue a directive to the next line, we must precede the newline character by a \ symbol. For example, the following directive defines a macro whose value is a string literal containing all 95 printable ASCII characters:

```
#define ASCII \
"!\"#$%&'()*+,-./0123456789:;<=>?" \
"@ABCDEFGHIJKLMNPQRSTUVWXYZ[\]^_` \
"abcdefghijklmnopqrstuvwxyz{|}~"
```

- *Directives can appear anywhere in a program.* Although it's customary to put preprocessing directives at the beginning of a file, they can actually appear anywhere, even in the body of a function.
- *Comments may appear on the same line as a directive.*

## Macro Definition

---

There are two kinds of macros: simple macros and parameterized macros. We'll look at these separately, then consider properties shared by both.

### Simple Macros

The definition of a simple macro has the form

```
#define identifier replacement-list
```

*replacement-list* is any sequence of C tokens; it may include identifiers, reserved words, operators, and all other symbols that can appear in C programs. When it encounters a macro definition, the preprocessor records the fact that *identifier* represents *replacement-list*; wherever *identifier* appears later in the file, the preprocessor substitutes *replacement-list*.

**Warning:** Don't put any extraneous symbols in a macro definition. They'll become part of the replacement list. Here are examples of the two most common macro definition errors:

```
#define N = 100 /* wrong */
```

```
int a[N];      /* becomes int a[= 100]; */
```

In this example, we've (incorrectly) defined N to be a pair of tokens (= and 100).

```
#define N 100; /* wrong */
```

```
int a[N];      /* becomes int a[100;]; */
```

**Here N is defined to be the tokens 100 and ;. In both examples, macro replacement causes a compilation error.**



Uses of simple macros include:

- *Defining “manifest constants.”* For example, here are definitions of constants named N, PI, and WARNING:

```
#define N      100
#define PI     3.14159
#define WARNING "Warning: nonstandard feature"
```

There are several advantages to defining constants: (1) Programs are easier to read; the name of the macro helps the reader understand the meaning of the constant. (2) Programs are easier to modify; the value of a constant can be changed throughout a program by modifying a single macro definition. (3) Errors of inconsistent use, as well as typographical errors, are reduced. If a numerical constant like 3.14159 appears many times in a program, chances are that it will occasionally be written 3.1416 or 3.14195 by accident.

- *Making minor changes to the syntax of C.* For example, the following macros allow the programmer to use the words begin and end instead of C’s braces:

```
#define begin {
#define end   }
```

Changing the syntax of C usually isn’t a good idea, however, since it can make programs harder for others to understand.

- *Renaming types.* For example, in previous chapters, we’ve defined a Boolean type by renaming int:

```
#define boolean int
```

- *Controlling condition compilation.* Macros can be used to control conditional compilation (discussed later in the chapter). For example, the following line in a program might signal that it’s to be compiled in “debugging mode,” with extra statements included to produce debugging output:

```
#define DEBUG
```

If this line is absent (or if DEBUG is later cancelled using #undef), the debugging statements won’t be included. Incidentally, it’s legal for a macro’s replacement list to be empty, as this example shows.

## **Parameterized Macros**

The definition of a parameterized macro has the form

```
#define identifier( x1 , x2 , ..., xn ) replacement-list
```

where  $x_1, x_2, \dots, x_n$  are identifiers (the macro’s *formal parameters*).

**Warning:** There must be *no space* between the identifier and the left parenthesis. If space is left, the preprocessor will assume that the definition is of a simple macro, with  $(x_1, x_2, \dots, x_n)$  part of *replacement-list*.

When the preprocessor encounters the definition of a parameterized macro, it stores the definition away for later use. Wherever a macro *invocation* of the form *identifier*( $y_1, y_2, \dots, y_n$ ) appears later in the program (where  $y_1, y_2, \dots, y_n$  are sequences of tokens), the preprocessor replaces it by *replacement-list*, with  $y_1$  substituted for  $x_1$ ,  $y_2$  substituted for  $x_2$ , and so forth. For example, suppose that we've defined the max macro as follows:

```
#define max(x,y) ((x)>(y)?(x):(y))
```

Now suppose that the following statement appears later in the program:

```
i = max(j+k,m-n);
```

The preprocessor will replace this line by

```
i = ((j+k)>(m-n)?(j+k):(m-n));
```

Parameterized macros often serve as simple functions. max behaves like a function that computes the larger of two values. Here's another example of a macro that behaves like a function:

```
#define toupper(c)\
('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

This macro tests whether the character c is between 'a' and 'z'. If so, it produces the upper-case version of c by subtracting 'a' and adding 'A'. If not, it leaves c unchanged.

A parameterized macro may have an empty parameter list. Here's an example of such a macro:

```
#define getchar() getc(stdin)
```

The empty parameter list isn't really needed, but it makes getchar resemble a function.

Using a parameterized macro instead of a function has several advantages:

- *The program may be slightly faster.* A function call usually requires some overhead during program execution—context information may be saved, parameters copied, and so forth. A macro invocation, however, requires no runtime overhead.
- *Macros are “generic.”* The formal parameters of a macro—unlike the formal parameters of a function—have no particular type. As a result, a macro can be used with parameters of any type, provided that the resulting program—after preprocessing—is valid. For example, the max macro can be used to find the larger of two values of types int, long int, float, double, etc.

Using a parameterized macro instead of a function also has disadvantages:

- *The compiled code will often be larger.* Each macro invocation causes the insertion of the macro's replacement list, thereby increasing the size of the source program (and hence the

compiled code). The more often the macro is used, the more pronounced this effect is. The problem is compounded when macro invocations are nested. Consider what happens when we use `max` to find the largest of three numbers:

```
n = max(i,max(j,k));
```

Here's the same statement after preprocessing:

```
n =
  ((i)>(((j)>(k)?(j):(k)))?
  (i):
  (((j)>(k)?(j):(k)))));
```

- *It's not possible to have a pointer to a macro.* As we'll see in Chapter 13, C has the concept of "pointer to a function," which is useful in certain programming situations. Macros are removed during preprocessing, so there's no corresponding notion of "pointer to a macro"; as a result, macros can't be used in these situations.
- *A macro may evaluate its parameters more than once.* A function evaluates its parameters only once; a macro may evaluate its parameters any number of times. For example, `max` evaluates one of its parameters once and the other twice. Evaluating a parameter more than once can cause unexpected behavior if the parameter has side effects. This kind of error is quite difficult to find, because a macro invocation looks the same as a function call.

Parameterized macros are useful for more than just simulating functions. For example, they're often used as templates for commonly used segments of code. Consider the following macro, which produces a call of `printf` suitable for printing the value of an integer expression:

```
#define print_int(x) printf("%d\n", x)
```

`print_int` might be used during debugging as a convenient way to print the values of variables and expressions. For example, the preprocessor will turn the line

```
print_int(i/j);
```

into

```
printf("%d\n", i/j);
```

## **The # Operator**

The `#` operator (a new feature of ANSI C) converts a macro parameter into a string literal. (The `#` operator can appear only in the replacement list of a parameterized macro.) For example, suppose that we modify the definition of `print_int` as follows:

```
#define print_int(x) printf(#x " = %d\n", x)
```

The `#` symbol in front of `x` instructs the preprocessor to create a string literal from `print_int`'s actual parameter. Thus, the invocation

```
print_int(i/j);
```

will become

```
printf("i/j " " = %d\n", i/j);
```

In ANSI C, adjacent string literals are concatenated, so this statement is equivalent to

```
printf("i/j = %d\n", i/j);
```

print\_int now labels the value that it prints, making it much more helpful.

## **The ## Operator**

The ## operator (another new feature) can “paste” symbols together. Consider the following macro definition:

```
#define mk_id(n) i##n
```

When mk\_id is invoked (as mk\_id(1), say), the preprocessor will first replace the formal parameter n by the actual parameter (1 in this case). Next, the preprocessor will join i and 1 to make a single symbol (i1).

Here’s an example of how mk\_id could be used to create several identifiers:

```
int mk_id(1), mk_id(2), mk_id(3);
```

After preprocessing, this declaration becomes

```
int i1, i2, i3;
```

## **General Properties of Macros**

In this section, we’ll discuss several general properties of macros. Properties shared by both simple and parameterized macros.

- *A macro’s replacement list may contain invocations of other macros.* For example, we can define the macro TWO\_PI in terms of the macro PI:

```
#define PI 3.14159
#define TWO_PI (2*PI)
```

When the preprocessor encounters the symbol TWO\_PI later in the program, it replaces it by (2\*PI). The preprocessor then *rescans* the replacement list to see if it contains invocations of other macros (PI in this case). The preprocessor will rescan the replacement list as many times as necessary to eliminate all macro names.

- *The preprocessor replaces only entire symbols, not portions of symbols.* As a result, the preprocessor ignores macro names that are embedded in identifiers, character constants, and string literals. For example, suppose that a program contains the following lines:

```
#define SIZE 256

char error_msg[] = "Error: SIZE exceeded";

if (BUFFER_SIZE > SIZE)
    printf("%s\n", error_msg);
```

After preprocessing, these lines will have the following appearance:

```
char error_msg[] = "Error: SIZE exceeded";

if (BUFFER_SIZE > 256)
    printf("%s\n", error_msg);
```

The string “Error: SIZE exceeded” and the identifier `BUFFER_SIZE` weren’t affected by preprocessing.

- *Macros may be “undefined” by the `#undef` directive.* The `#undef` directive has the form

```
#undef identifier
```

where *identifier* is a macro name. For example, the directive

```
#undef N
```

removes any definition of the macro `N` that might currently be in effect. (Attempting to undefine an undefined macro is legal.) A subsequent `#define` directive may give `N` a new definition. Attempting to redefine a macro without undefining it first is illegal unless the new definition is identical to the old one. Unless cancelled by `#undef`, a macro definition remains in effect from the point at which it appears to the end of the file.

- *Several useful macros are predefined in C.* Consult the *Language Reference* for a list of these macros, which provide information about the current compilation: the name of the source file, the date, the time, and so forth.

A final tip: The comma operator can be useful for creating more sophisticated macros. In particular, we can make the replacement list a series of expressions to be evaluated one at a time. For example, the following macro will read a string and then print it:

```
#define echo(s) \
    (scanf("%s", s), printf("%s", s))
```

Function calls are expressions, so it’s perfectly legal to combine them using the comma operator. `echo` can now be used as though it were a function:

```
echo(str);          /* becomes          */
/* (scanf("%s", str),      */
/* printf("%s", str));    */
```

Instead of using the comma operator, we could have enclosed the calls of `scanf` and `printf` in braces to form a compound statement:

```
#define echo(s) \
    {scanf("%s", s); printf("%s", s);}
```

Unfortunately, this method doesn't work as well. Suppose that we use `echo` in an `if` statement:

```
if (echo_flag) echo(str);
else scanf("%s", str);
```

Replacing `echo` gives the following result, which isn't legal according to the rules of C:

```
if (echo_flag)
{scanf("%s", str);
 printf("%s", str);};
else scanf("%s", str);
```

The compiler sees an `if` statement, followed by a null statement, followed by an `else` clause that doesn't belong to any `if`. We could solve the problem by remembering not to put a semicolon after each invocation of `echo`, but then the program would look odd.

## **Parentheses in Macro Definitions**

When defining a macro in which the replacement list is an expression, not just a single symbol, always enclose the replacement list in parentheses. Furthermore, if the macro has parameters, parenthesize each parameter every time it appears in the replacement list. Let's see what can happen if these rules are overlooked.

To illustrate the importance of putting parentheses around the replacement list, consider the following macro definition, in which the parentheses are missing:

```
#define TWO_PI 2*3.14159
/* needs parentheses around
   replacement list */
```

During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

becomes

```
conversion_factor = 360/2*3.14159;
```

The division will be performed before the multiplication, yielding a result different from the one intended.

Putting parentheses around the replacement list alone isn't enough if the macro has parameters. Each occurrence of a parameter should be parenthesized as well. For example, suppose that `scale` is defined as follows:

```
#define scale(x) (x*10)
/* needs parentheses around x */
```

During preprocessing, the statement

```
j = scale(i+1);
```

becomes

```
j = (i+1*10);
```

Since multiplication takes precedence over addition, this statement is equivalent to

```
j = i+10;
```

Of course, what we wanted was

```
j = (i+1)*10;
```

## File Inclusion

---

The `#include` directive causes the preprocessor to insert the contents of a particular file as part of the program. Files that are included in this fashion are called *header files*; we'll discuss them in more detail later in the chapter. By convention, header files have the extension `.h`.

The `#include` directive has two forms. The first form is used for header files that belong to the C library:

```
#include <file-name>
```

The second form is used for header files written by the programmer:

```
#include "file-name"
```

The file name may include a directory path and/or a drive specifier:

```
#include "utils.h"  
#include "c:utils.h"  
#include "\\cprogs\\utils.h"  
#include "c:\\cprogs\\utils.h"
```

File names in `#include` directives are handled by the preprocessor, so they aren't subject to the same rules as string literals. (In a string literal, `\c` and `\u` would be treated as escape sequences.)

**Note:** In TopSpeed C, the two forms of the `#include` directive are equivalent. In both cases, the compiler uses the redirection file to locate the header, even if a path and/or drive specifier is present.

A header file may itself contain `#include` directives.

## Conditional Compilation

---

The C preprocessor recognizes a number of directives that support *conditional compilation*—the inclusion or exclusion of a particular section of program text depending on the outcome of a test performed by the preprocessor.

## The #if and #endif Directives

To include a group of lines in the program if a constant expression is nonzero, just precede the lines with an #if directive and follow the lines with an #endif directive:

When the preprocessor encounters the #if directive, it evaluates the constant expression (which may contain character and integer constants or macros that represent character and integer constants). If the value of the expression is zero, the lines between #if and #endif will be ignored during compilation. If the value of the expression isn't zero, the lines between #if and #endif *will* be processed by the compiler. The #if and #endif directives will have had no effect on the program.

## The defined Operator

When applied to an identifier, the defined operator produces the value 1 if the identifier is a currently defined macro; it produces 0 otherwise. The defined operator is useful in conjunction with the #if directive; it allows us to write

```
#if defined(DEBUG)
...
#endif
```

The lines between the #if and #endif directives will be included in the program only if DEBUG is defined as a macro. Incidentally, the parentheses around DEBUG aren't required.

## The #ifdef and #ifndef Directives

The #ifdef directive is similar to #if, but tests whether a particular identifier is currently defined as a macro:

```
#ifdef identifier
lines to be included if identifier is defined as a macro
#endif
```

Strictly speaking, the #ifdef directive isn't necessary; we can combine the #if directive with the defined operator to get the same effect. That is, the directive

```
#ifdef identifier
```

is equivalent to

```
#if defined(identifier)
```

The #ifndef directive is similar to #ifdef, but tests whether a particular identifier is *not* currently defined as a macro. The directive

```
#ifndef identifier
```

is the same as

```
#if !defined(identifier)
```



## The #elif and #else Directives

#if, #ifdef, and #ifndef blocks can be nested just like ordinary if statements. For additional convenience, the preprocessor provides the #elif and #else directives, which may be used in conjunction with #if, #ifdef, or #ifndef to test a series of conditions. The following example shows how #elif and #else might be combined with #if:

```
#if expr1
lines to be included if expr1 is nonzero
#elif expr2
lines to be included if expr2 is nonzero
#else
lines to be included otherwise
#endif
```

Any number of #elif directives may appear between #if and #endif, but at most one #else may appear between #if and #endif.

## Uses of Conditional Compilation

Conditional compilation has many uses; here are a few of the most common:

- *Writing programs that are portable to several machines or operating systems.* The following example includes one of three groups of lines depending on whether DOS, OS2, or XENIX is defined as a macro:

```
#if defined(DOS)
...
#elif defined(OS2)
...
#elif defined(XENIX)
...
#endif
```

A file might contain many of these constructs. Earlier in the file (or in a header file), one (and only one) of the macros will be defined, thereby indicating a particular operating system:

```
#define OS2
```

- *Providing a default definition for a macro.* When a program includes header files, it may not be clear whether or not a particular macro has been defined in one of those files. Conditional compilation allows us to provide a definition for a macro if none would exist otherwise. For example, the following lines will define the macro BUFFER\_SIZE if it wasn't previously defined:

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

- *Including debugging code.* Conditional compilation is especially useful as a way of including debugging statements in a program. Consider the following example:

```

#ifdef DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif

```

If the macro `DEBUG` is defined, the compiled program will print the values of `i` and `j`. If `DEBUG` isn't defined, the calls of `printf` won't appear in the compiled program. These four lines may be left in the final program, allowing diagnostic information to be produced later (by recompiling with `DEBUG` defined) if errors arise during the operation of the program.

- *Temporarily disabling code that contains comments.* Since comments can't be nested in ANSI C, it's not possible to use a comment to temporarily disable code that contains comments. Instead, we can use an `#if` directive:

```

#if 0
lines containing comments
#endif

```

## The Overall Structure of a C Program

---

Up to this point, we've assumed that a C program consists of a single file. In fact, a program may be divided among any number of *source files*. By convention, source files have the extension `.c`. Each source file contains part of the program, typically definitions of functions and variables. One source file must contain a function named `main`, which serves as the starting point for the program.

Dividing a program into files has several advantages. Related functions and variables can be grouped together into a single file, making the structure of the program clearer. Each source file can be compiled separately. A great convenience if the program is large. Also, source files can be reused in other programs.

### Header Files

Header files contain information—macro definitions and function declarations, in particular—to be shared among several source files.

For example, suppose that we're writing a program that uses the values `TRUE` and `FALSE`. We might create a header file named `boolean.h` that contains the following macro definitions:

```

#define boolean int
#define TRUE 1
#define FALSE 0

```

At the top of each source file, we can put the line

```
#include "boolean.h"
```

to provide access to the definitions of `boolean`, `TRUE`, and `FALSE`.

Putting macro definitions in a header file has several advantages. First, we save time by not having to copy the definitions into each source file. Second, the program becomes easier to modify; changing the definition of a macro requires only that we edit the header file in which the macro is defined; we don't have to modify the files in which the macro is used. Third, we don't have to worry about the possibility that different source files contain different definitions of a macro.

Another common kind of header file contains declarations of the functions in a particular source file. For example, suppose that the file `demo.c` contains definitions of the functions `f1`, `f2`, `f3`, and `f4`. Functions `f1` and `f2` are meant to be called from other source files, while `f3` and `f4` are used only inside `demo.c` itself. It's a good idea to put declarations of `f1` and `f2` in a header file:

```
int f1(int a, int b);  
void f2(float x, int y);
```

Convention dictates that the header file have the same name as the source file, with the extension `.h` instead of `.c`, so this file would have the name `demo.h`. We should include `demo.h` in every file in which `f1` or `f2` is called, so that the compiler will know each function's return type and the number and type of its parameters. We should also include `demo.h` in `demo.c` itself, allowing the compiler to check that the declarations of `f1` and `f2` are consistent with their definitions.

**Warning:** When calling a function `f` that is defined in another file, always make sure that the compiler has seen a declaration of `f` prior to the call. (Including a header file that declares `f` is an excellent way to satisfy this requirement.) If the compiler hasn't seen a declaration of `f`, it assumes that `f`'s return type is `int`. It has no way to know how many parameters `f` should have or what their types should be. When `f` is called, the default argument promotions are performed. If the compiler's assumptions are wrong, then the program won't behave properly.

Beginning C programmers often make the mistake of using the `#include` directive to include one source file in another. `#include` should be used only with header files, not source files. No direct connection between source files is necessary.

## Compilation and Linking

Before a C program can be executed, it must first be compiled and linked. Each source file can be compiled separately, if desired. Header files need not be compiled at all; a header file is implicitly compiled whenever a source file that includes it is compiled.

Compilation leaves certain "gaps" that are filled in during linking. For example, a source file may contain calls of functions that aren't defined in

that file; as a result, the compiler can't completely resolve these calls. The linker checks the object files created by the compiler to make sure that they mesh properly; if so, it fills in any gaps left from compilation and creates a single executable file.

During compilation, functions in the C library are treated in the same way as functions defined by the programmer. For example, calls of functions declared in `<stdio.h>` are resolved by the linker; the compiler knows nothing about these functions except for the information in `<stdio.h>`.

# CHAPTER 11

## Declarations

In previous chapters, we've seen how to declare variables and functions. In this chapter, we'll look at declarations in more detail. After discussing the difference between declarations and definitions, we'll cover storage classes, type qualifiers, declarators, and initializers, all of which can appear in declarations. As we examine storage classes, we'll become acquainted with the important concepts of storage duration, scope, and linkage. The chapter concludes with a discussion of two topics that are closely related to declarations: blocks (compound statements that contain declarations) and scope rules.

### Declarations versus Definitions

---

*Declarations* furnish information to the compiler about the meaning of identifiers (the names of variables and functions, for example). *Definitions* are declarations that cause the compiler to reserve storage space. Function definitions are easy to distinguish from function declarations, since the definition of a function includes the function's body. Distinguishing variable definitions from declarations is sometimes more difficult. (All variable declarations in previous chapters were also definitions, so the difference didn't matter in prior examples.)

Declarations have the form

```
declaration-specifiers declarators ;
```

A declaration specifier is one of the following:

a *storage class* (auto, static, extern, or register)

a *type specifier* (for example, long, short, unsigned, int, or float)

a *type qualifier* (const or volatile)

At most one storage class is allowed; if present, it should come first. There may be several type specifiers and/or qualifiers; these may appear in any order.

Declarators include identifiers (variable names), identifiers followed by square brackets (array names), identifiers preceded by the \* symbol (pointer names), and identifiers followed by parentheses (function names).

Declarators are separated by commas. A declarator may be followed by an initializer.

The following examples display a variety of declarations:

```
static float x, y, *p;
```

The storage class is static, float is a type specifier, and x, y, and \*p are declarators.

```
const char computer[] = "IBM PS/2";
```

There is no storage class, const is a type qualifier, char is a type specifier, computer[] is a declarator, and "IBM PS/2" is an initializer.

```
extern unsigned long int a[10], j;
```

The storage class is extern, the words unsigned, long, and int are type specifiers, and a[10] and j are declarators.

```
extern double f(double a);
```

The storage class is extern, double is a type specifier, and f(double a) is a declarator.

The next few sections discuss storage classes, type qualifiers, declarators, and initializers in more detail.

## Storage Classes

---

Storage classes may be specified for variables and (to a limited extent) functions and formal parameters. We'll concentrate on variables and formal parameters first. (In the following discussion, the term *block* refers to the body of a function. Later in the chapter, we'll see that another kind of block also exists in C.)

### The Storage Class of a Variable

The choice of storage class affects the following properties of a variable:

- *Storage duration.* The *storage duration* of a variable is the portion of program execution during which storage for the variable exists. There are two possible storage durations: *automatic* (storage for the variable is allocated when the surrounding block is executed; storage is deallocated when the block terminates, causing the variable to lose its value) and *static* (the variable has a permanent storage location, hence it retains its value throughout the execution of the program).
- *Scope.* The *scope* of a variable is the portion of the program text in which the variable can be referenced. A variable may have either *block scope* (the variable is visible from its point of declaration to the end of the enclosing block) or *file scope* (the

variable is visible from its point of declaration to the end of the enclosing file).

- *Linkage.* The *linkage* of a variable determines the extent to which it may be shared. A variable with *external linkage* may be shared by all files in a program. A variable with *internal linkage* is restricted to a single file, but may be shared by the functions in that file. (If a variable with the same name appears in another file, it's a different variable.) A variable with *no linkage* belongs to a single function and can't be shared at all.

A variable's default storage duration, scope, and linkage depend on whether the variable is declared inside a block or at the outermost level of a program, outside any function definition. If the variable is declared inside a block, its storage duration is automatic by default; it has block scope and no linkage. If the variable is declared at the outermost level of a program, by default its storage duration is static, it has file scope, and its linkage is external.

For many variables, the default storage duration, scope, and linkage are satisfactory. When they aren't, the programmer can control these properties by specifying one of the following storage classes:

- *auto* Æ Allowed only for a variable that is declared inside a block. The variable's storage duration is automatic; it has block scope and no linkage. (The auto storage class is almost never specified explicitly, since it's the default for variables declared inside a block.)
- *static* Æ The variable's storage duration is static. If the variable is declared inside a block, it has block scope and no linkage. If the variable is declared outside any function definition, it has file scope and internal linkage.
- *extern* Æ The variable's storage duration is static. If the variable is declared inside a block, it has block scope; its linkage depends on whether a previous declaration of the variable appears at the file level, outside of any function definition. (If so, the variable has the linkage specified in that declaration. If not, the variable has external linkage.) If the variable is declared outside any function definition, it has file scope. If the variable's first declaration at the file level uses the static storage class, it has internal linkage; otherwise, it has external linkage.
- *register* Æ Allowed only for a variable that is declared inside a block. The variable's storage duration, scope, and linkage are the same as if it were an auto variable. The use of the register storage class in a variable declaration is a request to the compiler to keep the variable in a register. (Keeping frequently used variables in registers may improve execution speed, since data stored in registers can usually be accessed faster than data stored in main memory.) The compiler may choose to treat a register variable as an ordinary auto variable. (This is the case in

TopSpeed C, which (like other optimizing compilers) prefers to do its own register allocation.)

A variable declaration inside a function body that specifies the extern storage class is *never* a definition; if any other storage class is specified (or none at all), it's *always* a definition. The rules for variable declarations at the file level are more complicated. A variable declaration at the file level is *always* a definition if it contains an initializer for the variable. A declaration that doesn't initialize the variable is *probably* a definition if the storage class is static or omitted; if the storage class is extern, it's *never* a definition. See the *Language Reference* for details.

All parameters have automatic storage duration, block scope, and no linkage. The only storage class that may be specified for parameters is register.

The following example illustrates the various storage classes. (Assume that no other declarations appear in the same file as these.)

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
    auto int g;
    int h;
    static int i;
    extern int j;
    register int k;
}
```

a has static storage duration (a retains its value throughout the execution of the program), file scope (a is visible from its point of declaration to the end of the file), and external linkage (if other files contain declarations of a variable named a with external linkage, these declarations refer to the variable a defined here). b's storage duration is static, it has file scope, and its linkage is external. The declaration of b isn't a definition (since no initializer is present), so a definition of b must appear in another file. c's storage duration is static and it has file scope, but its linkage is internal (it can't be shared with other files). Both d and e have automatic storage duration, block scope, and no linkage. g's storage duration is automatic; it has block scope and no linkage. h has the same storage duration, scope, and linkage as g. i has static storage duration, block scope, and no linkage. j's storage duration is static, it has block scope, and its linkage is external. A definition of j must appear in another file. k's storage duration is automatic; it has block scope and no linkage. We've requested that e and k be kept in registers, but the compiler is free to ignore this request.

## **Variables with External Linkage**

The extern storage class deserves special mention. A variable declaration that specifies the extern storage class isn't a definition—it doesn't reserve storage for the variable—unless it also initializes the variable. The extern



class makes it possible for several source files to share the same variable. One file should contain a *definition* of the variable; the other files contain *declarations* that refer to the variable.

Suppose that the variable to be shared is named `i`. In one file, we declare `i` without the word `extern`:

```
int i;
```

(If we want, we can supply an initial value for `i`.) The compiler treats this as a definition and therefore allocates storage for `i`. The other files contain declarations that are identical except for the presence of the word `extern`:

```
extern int i;
```

The compiler treats these declarations as references to the original variable `i`, so it doesn't allocate any additional storage.

#### Warning:

When declarations of the same variable appear in different files, the compiler can't check that the declarations are consistent. For example, one file may contain the declaration

```
extern int i;
```

while another file contains the declaration

```
extern long int i;
```

An error of this kind may cause the program to behave unpredictably.

To avoid inconsistencies, declarations of shared variables are usually put in header files. A source file that needs access to a particular variable can then include the appropriate header file.

## **The Storage Class of a Function**

Function declarations (and definitions), like variable declarations, may include a storage class; however, the only options are `extern` and `static`. The `extern` storage class indicates that the function's linkage is external, hence it may be called from other files. The `static` storage class indicates internal linkage—the function may be called only within the file in which it is defined. If no storage class is specified, the function is assumed to have external linkage.

Consider the following function declarations:

```
extern int f(int i);  
static int g(int i);  
int h(int i);
```

`f` has external linkage, `g` has internal linkage, and `h` (by default) has external linkage.

## Type Qualifiers

---

There are two type qualifiers: `const` and `volatile`. `volatile` is used only in certain special cases, so we won't discuss it here; see the *Language Reference* for information.

The `const` type qualifier is used to declare objects that resemble variables, but are “read-only”; a program can access the value of a `const` object, but can't change it. For example, the declaration

```
const int n = 10;
```

creates a read-only object named `n` whose value is 10. The declaration

```
const int months[] =
    {31, 28, 31, 30, 31, 30,
     31, 31, 30, 31, 30, 31};
```

creates a read-only array named `months`.

`const` isn't limited to use in declarations of read-only objects with specified values; as we saw in Chapter 8, it can also be used to protect function parameters from change.

Using `const` to indicate that the value of an object won't change has several advantages:

- The compiler can check that the program doesn't inadvertently attempt to change the value of the object.
- The object can be stored in ROM (read-only memory) — an important feature in some applications.
- The compiler may be able to save space by replacing references to the object by the object's value, so that the object need not occupy any memory when the program is run.

The `#define` directive can also be used to define constants; however, using `const` is different than using `#define`:

- The `#define` directive allows the definition of a macro that represents a numerical, character, pointer, or string constant, but `const` can be used to create read-only objects of *any* type.
- `const` objects are subject to the same scope rules as variables; constants created using `#define` aren't.
- Unlike constants created using `#define`, `const` objects can't be used in constant expressions. For example, we can't write

```
const int n = 10;
int a[n];          /* illegal */
```

## Declarators

---

Let's review the declarators that we've seen in previous chapters before we look at more complex declarators.

In the simplest case, a declarator can be a single identifier:

```
int i;
```

Declarators may also include a preceding \*, indicating “pointer to”:

```
int *p;
```

A declarator that is followed by square brackets represents an array:

```
int a[10];
```

(In some cases—for example, if the array is a formal parameter or has an initializer—the array size may be omitted.) A declarator that is followed by a parameter list represents a function:

```
int f(int i);
```

For compatibility with K&R C, the parentheses may be left empty, but this usage isn't recommended. Leaving the parentheses empty isn't the same as putting the word `void` between them. A declaration in which the parentheses are empty doesn't specify how many parameters the function has. A declaration in which `void` appears between the parentheses specifies that the function has no parameters.

A declarator may contain combinations of \*, [], and (); the following declaration shows a few of the possibilities:

```
int *fp(int i), (*pf)(int i), *ap[10], (*pa)[];
```

`fp` is a function that takes an `int` parameter and returns a pointer to an `int` value. `pf` is a pointer to a function that takes an `int` parameter and returns an `int` value. (Chapter 13 discusses the concept of a pointer to a function.) `ap` is an array of ten pointers to `int` values. `pa` is a pointer to an array of `int` values (there's no need to specify the size of the array).

Notice the use of parentheses to change the meaning of a declarator. The declarators for `fp` and `pf` are the same except for the parentheses surrounding `*pf`, which indicate that `pf` is a pointer, not a function. In a declarator, \* has lower precedence than () and [], so parentheses are needed when the declarator is to represent a pointer rather than a function or an array.

There are some restrictions on declarators: functions can't return arrays or functions, nor are arrays of functions possible (although an array of *pointers* to functions is legal).

## Initializers

---

A declarator that represents a variable may be followed by an initializer, as we've seen in previous chapters:

```
int i = 0, *p = NULL, a[5] = {1,2,3,4,5};
```

There are some restrictions on the values that may appear in an initializer:

- A brace-enclosed initializer for an array, structure, or union must contain only constant expressions:

```
#define N 2
```

```
int powers[5] = {1, N, N*N, N*N*N, N*N*N*N};
```

We'll discuss initializers for structures and unions in Chapter 12.

- Similarly, an initializer for a variable with static storage duration must be constant:

```
#define FIRST 1
#define LAST 100
```

```
static int i = LAST - FIRST + 1;
```

- If a variable has automatic storage duration (and isn't an array, structure, or union), its initializer need not be constant:

```
int f(int n)
{
    int last = n - 1;
    ...
}
```

What's the initial value of a variable if we don't explicitly initialize it? C has two simple rules: (1) Variables whose storage duration is static are initialized to 0 by default. (Pointers are initialized to NULL.) (2) Variables with automatic storage duration have no default initial value. Rule (1) also applies to the left-over elements of an array whose initializer is too short.

## Blocks

---

In Chapter 2, we introduced compound statements of the form

```
{
    statements
}
```

However, C allows compound statements to contain declarations as well:

```
{
    declarations
    statements
}
```

We'll use the term *block* to describe a compound statement that contains declarations. Here's an example of a block:

```
if (i < j) {  
    int temp;  
  
    temp = i;  
    i = j;  
    j = temp;  
}
```

By default, the storage duration of a variable declared in a block is automatic: storage for the variable is allocated when the block is entered and deallocated when the block is exited. The variable has block scope; it can't be referenced outside the block.

The body of a function is a block. Blocks are also useful inside a function body when we need one or more variables for temporary use. In our last example, we needed a variable temporarily so that we could exchange the values of *i* and *j*. Declaring a temporary variable in a block has two advantages: (1) It avoids cluttering the declarations at the beginning of the function body with variables that are used only briefly. (2) It reduces name conflicts. In our example, the name *temp* can be used elsewhere in the same function for different purposes. The variable *temp* declared in the block is strictly local to the block.

## Scope Rules

---

In a C program, the same identifier may have several different meanings. C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.

Here's the most important scope rule: When a declaration inside a block names an identifier that's already visible (because it has file scope or because it's declared in an enclosing block), the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning. At the end of the block, the identifier regains its old meaning.

Consider the following example:

```
int i;           /* Declaration 1 */

void f(int i)    /* Declaration 2 */
{
    i = 1;       /* Use 1 */
}

void g(void)
{
    int i = 2;    /* Declaration 3 */
    if (i > 0) {  /* Use 2 */
        int i;    /* Declaration 4 */
        i = 3;    /* Use 3 */
    }
    i = 4;        /* Use 4 */
}

void h(void)
{
    i = 5;        /* Use 5 */
}
```

In this example, the identifier *i* has four different meanings:

- In Declaration 1, *i* is a variable with static storage duration and file scope.
- In Declaration 2, *i* is a formal parameter with block scope.
- In Declaration 3, *i* is an automatic variable with block scope.
- In Declaration 4, *i* is also automatic and has block scope.

*i* is used five times. C's scope rules allow us to determine the meaning of *i* in each case:

- Use 1 refers to the parameter, not the variable in Declaration 1, since Declaration 2 hides Declaration 1.
- Use 2 refers to the variable in Declaration 3, since Declaration 3 hides Declaration 1.
- Use 3 refers to the variable in Declaration 4, which hides Declaration 3.
- Use 4 refers to the variable in Declaration 3. It can't refer to Declaration 4, since Use 4 occurs outside the block in which Declaration 4 appears.
- Use 5 refers to the variable in Declaration 1.

# CHAPTER 12

## Structures, Unions, and Enumerations

This chapter introduces three new types: structures, unions, and enumerations. A structure is collection of objects (members), possibly of different types. A union is similar to a structure; however, the members of a union don't exist simultaneously, allowing their storage to overlap. An enumeration is an integer type whose values are named by the programmer. The chapter also discusses type definitions, which allow the programmer to create new types and rename existing types.

### Declaring Structures

---

A *structure* is a collection of one or more objects (called *members*), which may be of different types. (Structures are called *records* in many other programming languages; members are often called *components* or *fields*.)

C provides three methods for declaring structures. We'll look at two of these methods now; the third is discussed later in the chapter (under "Type Definitions"). Our running example in this chapter will be a part structure that's suitable for use in an inventory program. The structure contains three members: `part_no` (the part number), `part_name` (the name of the part), and `on_hand` (the quantity of the part on hand). All examples assume that the macro `NAME_LEN` has been defined previously.

- *Method 1: Declare structure variables directly, without naming the structure itself.* The following example declares two structure variables named `p1` and `p2`:

```
struct {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
} p1, p2;
```

- *Method 2: Declare a structure tag, then use the structure tag to declare variables.* A *structure tag* is a name used to identify a particular structure. In the following example, we've declared a structure tag named `part`, then used `part` to declare the variables `p1` and `p2`. Notice that `part` itself isn't a type; however, when preceded by the word `struct`, it can be used as a type.

```
struct part {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
};
struct part p1, p2;
```

This is one of the few places in C where it's proper to have a semicolon immediately following a right brace.

**Warning:** The semicolon at the end of a structure declaration is mandatory. Accidentally omitting it can have unexpected effects:

```
struct part {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
}      /* missing semicolon */

f(void)
{
    ...
}
```

**The return type of the function f is missing. Normally, the return type would be int by default, but in this case the compiler assumes a return type of struct part because the preceding structure declaration wasn't terminated properly.**

The declaration of a structure tag can be combined with the declaration of structure variables:

```
struct part {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
} p1, p2;
```

This declaration not only defines part as a structure tag (making it possible to use part later to declare more variables) but also declares the variables p1 and p2.

Normally, we can choose between these two methods based on whether or not there's a reason to name the structure for later use. If so, a structure tag is necessary; otherwise, either method will work. There's one situation in which structure tags are mandatory, however: when a structure contains a reference to itself (see Chapter 13). Subsequent examples in this chapter assume that the part tag has been previously declared.

Regardless of which method is used, structure variables may be initialized at the point of declaration:

```
struct part p1 = {528, "Disk drive", 10},
              p2 = {914, "Printer cable", 5};
```

Expressions used in a structure initializer must be constant.

A note about the scope of member names: All names within a single structure must be different, but member names don't otherwise conflict with



any other names in a program. Similarly, structure tags must all be different, but don't conflict with other names. For example, the following declarations are legal, and both may appear in the same program:

```
struct a {int a, b, c; } a;  
struct d {int b, e; } b;
```

The member named a doesn't conflict with the variable a or the structure tag a, nor do the members named b conflict with each other or with the variable b.

## Operations on Structures

---

To access a member within a structure, we write the name of the structure first, then a period, then the name of the member. For example, the following statements will initialize a variable p1 of type struct part:

```
p1.part_no = 528;  
strcpy(p1.part_name, "Disk drive");  
p1.on_hand = 10;
```

The members of a structure are lvalues and can therefore be used in the same way as ordinary variables.

In addition to allowing the selection of individual members, ANSI C provides a limited set of operations on whole structures:

- *Assignment of whole structures.* For example, the assignment  
p2 = p1;

copies each member of p1 to the corresponding member of p2.

- *Passing a whole structure to a function.* The following function, when given a part structure as a parameter, prints the values of its members:

```
void print_part(struct part p)  
{  
    printf("Part number: %d\n", p.part_no);  
    printf("Part name: %s\n", p.part_name);  
    printf(  
        "Quantity on hand: %d\n",  
        p.on_hand);  
}
```

- *Returning a whole structure from a function.* The following function returns a part structure that it constructs from data supplied by the user:

```

struct part read_part(void)
{
    struct part p;

    printf("Enter part number: ");
    scanf("%d", &p.part_no);
    printf("Enter part name: ");
    scanf(" "); /* skip white space */
    gets(p.part_name);
    printf("Enter quantity on hand: ");
    scanf("%d", &p.on_hand);
    return p;
}

```

Other whole-structure operations, such as testing two structures for equality or inequality, aren't allowed.

## Arrays of Structures

---

Arrays and structures may be nested within one another to create complex data structures. We've already seen an example of an array nested inside a structure—the `part_name` member of the `part` structure is an array of characters. Arrays of structures are also common. For example, the following array of part structures is capable of storing information about 100 parts:

```

#define MAX_PARTS 100

struct part inventory[MAX_PARTS];

```

We can access one of these structures by subscripting the array:

```

inventory[i] = inventory[j];
/* copy a part from          */
/* position j to position i */

```

Accessing a member within a structure requires subscripting, followed by selection:

```

inventory[i].part_no = 883;

```

This statement assigns 883 to the `part_no` member of `inventory[i]`. The operations of subscripting and member selection can be applied as many times as needed. For example, accessing a character in a part name requires subscripting (to select a particular part), followed by selection (to select the `part_name` member), followed by subscripting (to select a character within the part name):

```

inventory[i].part_name[0] = '\0';

```

### **Example Program: Maintaining a Parts Database**

Suppose that we're given the job of maintaining a database of information about a parts inventory. Our program must support the following operations:

- Enter a new part number, part name, and initial quantity on hand.

The program must print an error message if the part is already in the database or if the database is full.

- Given a part number, print the name of the part and the current quantity on hand. The program must print an error message if the part number isn't found in the database.
- Given a part number, increase or decrease the quantity on hand by a specified amount. The program must print an error message if the part number isn't found.
- Print a table showing all information in the database. Parts must be displayed in the order in which they were entered.
- Terminate program execution.

We'll use the codes i (insert), s (search), u (update), p (print), and q (quit) to represent these operations. A session with the program might look like this:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10
```

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```

```
Enter operation code: s
Enter part number: 914
Part not found.
```

```
Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5
```

```
Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8
```

```
Enter operation code: p
Part Number   Part Name      Quantity on Hand
    528        Disk drive           8
    914        Printer cable        5
```

```
Enter operation code: q
```

Here's the program:

```

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

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0; /* number of parts */
                  /* currently stored */

int find_part(int part_no);
struct part read_part(void);
void insert(void);
void search(void);
void update(void);
void print(void);

/*****/
/* main: Prompts the user to enter an */
/* operation code, then calls a function to */ /* perform the requested
action. Repeats */
/* until the user enters the command 'q'. */
/* Prints an error message if the user enters*/
/* an illegal code. */
/*****/

main()
{
    char code;

    for (;;) {
        printf("Enter operation code: ");
        scanf("%c", &code);
        switch (code) {
            case 'i': insert(); break;
            case 's': search(); break;
            case 'u': update(); break;
            case 'p': print(); break;
            case 'q': exit(EXIT_SUCCESS);
            default: printf("Illegal code\n");
        }
        printf("\n");
    }
}

/*****/
/* find_part: Looks up a part number in the */
/* inventory array. Returns the array index */
/* if the part number is found; otherwise, */
/* returns -1. */
/*****/

```

```

int find_part(int part_no)
{
    int i;

    for (i = 0; i < num_parts; i++)
        if (inventory[i].part_no == part_no)
            return i;
    return -1;
}

/*****
/* read_part: Prompts the user to enter a
/* part number, part name, and quantity on
/* hand. Returns a structure containing this
/* information.
*****/

struct part read_part(void)
{
    struct part p;

    printf("Enter part number: ");
    scanf("%d", &p.part_no);
    printf("Enter part name: ");
    scanf(" ");
    gets(p.part_name);
    printf("Enter quantity on hand: ");
    scanf("%d", &p.on_hand);
    return p;
}

/*****
/* insert: Calls read_part to obtain
/* information about a new part, then
/* inserts the part into the database. Prints
/* an error message and returns prematurely
/* if the part already exists or the database
/* is full.
*****/

void insert(void)
{
    struct part p;

    if (num_parts == MAX_PARTS) {
        printf(
            "Database is full; "
            "no more parts can be added.\n");
        return;
    }
    p = read_part();
    if (find_part(p.part_no) >= 0) {
        printf("Part already exists.\n");
        return;
    }
    inventory[num_parts++] = p;
}

```

```

/*****
/* search: Prompts the user to enter a part */ /* number, then looks up
the part in the */ /* database. If the part exists, prints the */ /*
name and quantity on hand; if not, prints */
/* an error message. */
*****/

void search(void)
{
    int i, part_no;

    printf("Enter part number: ");
    scanf("%d", &part_no);
    i = find_part(part_no);
    if (i >= 0) {
        printf(
            "Part name: %s\n",
            inventory[i].part_name);
        printf(
            "Quantity on hand: %d\n",
            inventory[i].on_hand);
    } else
        printf("Part not found.\n");
}

/*****
/* update: Prompts the user to enter a part */ /* number. Prints an
error message if the */
/* part doesn't exist; otherwise, prompts the*/
/* user to enter change in quantity on hand */
/* and updates the database. */
*****/

void update(void)
{
    int i, part_no, change;

    printf("Enter part number: ");
    scanf("%d", &part_no);
    i = find_part(part_no);
    if (i >= 0) {
        printf(
            "Enter change in quantity on hand: ");
        scanf("%d", &change);
        inventory[i].on_hand += change;
    } else
        printf("Part not found.\n");
}

/*****
/* print: Prints a listing of all parts in */
/* the database, showing the part number, */
/* part name, and quantity on hand. Parts */
/* are printed in the order in which they */
/* were entered into the database. */
*****/

```

```

void print(void)
{
    int i;

    printf(
        "Part Number    Part Name"
        "                Quantity on Hand\n");
    for (i = 0; i < num_parts; i++) {
        printf("%7d        ", inventory[i].part_no);
        printf("%-25s", inventory[i].part_name);
        printf("%11d\n", inventory[i].on_hand);
    }
}

```

Several aspects of this program deserve mention:

- The variables `inventory` and `num_parts` have been made external so that all functions in the program will have access to them.
- For clarity, each command has been implemented as a separate function. The “auxiliary” functions `find_part` and `read_part` assist the functions that implement commands.
- In `main`, the format string “%c” allows `scanf` to skip over white space before reading the operation code. The space in the format string is crucial; without it, `scanf` would sometimes read the newline character that terminated a previous line of input. The format string “ “ in `read_part` serves the same purpose.

## Type Definitions

---

Type definitions allow us to define names for types. For example, we can define a type named `part` in the following way:

```

typedef struct {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
} part;

```

`part` may be used in the same way as the built-in types. For example, we might use it to declare variables:

```
part p1, p2;
```

Defining `part` to be a type isn’t the same as defining it to be a structure tag. The most obvious difference is that the word `struct` no longer precedes `part` when it’s used to declare variables (or in other contexts, for that matter).

Type definitions can be used to define types of all kinds, not just structure types. The following examples show definitions of a float type (dollars), an array type (`int_array`), and a pointer type (`ptr_to_int`):

```
typedef float dollars;
typedef int int_array[5], *ptr_to_int;
```

Notice that, except for the presence of the word `typedef` at the beginning, a type definition is syntactically identical to a variable declaration.

Once a type name has been defined, it can be used in the same way as the built-in type names. The following examples illustrate the use of `dollars`, `int_array`, and `ptr_to_int` to declare variables:

```
dollars cash_in, cash_out;
int_array a1, a2;
ptr_to_int p, q, r;
```

These declarations are equivalent to

```
float cash_in, cash_out;
int a1[5], a2[5];
int *p, *q, *r;
```

Type definitions can make a program more understandable (assuming that the programmer has been careful to choose meaningful type names). They can also make a program easier to modify. For example, suppose that we define the type `t` to be equivalent to `int`, then use `t` to declare variables throughout a large program. If we later decide that `t` should be `long int` instead of `int`, we can just modify the definition of `t`; the variable declarations need not be changed. (Type definitions are used in the C library for exactly this reason. The header `<stddef.h>`, for example, contains definitions of several types that can vary from one C implementation to another.)

In previous chapters, we used the `#define` directive to rename types. For example, we defined the boolean type in the following way:

```
#define boolean int
```

If `#define` already provides the ability to name a type, why do we need `typedef`? To answer that question, let's examine the differences between `#define` and `typedef`:

- Type definitions are more powerful than macro definitions. In particular, the `int_array` type can't be defined as a macro.
- Typedef names are subject to the same scope rules as variables; for example, a typedef name defined inside a function body wouldn't be recognized outside the function. Macro names, on the other hand, are replaced by the preprocessor wherever they appear.
- Using a macro definition to define a type doesn't always work. For example, if `ptr_to_int` is defined as

```
#define ptr_to_int int *
```

then the declaration

```
ptr_to_int p, q, r;
```

is equivalent to



```
int *p, q, r;
```

Only `p` is a pointer; `q` and `r` are ordinary integer variables. Type definitions don't have this problem.

As a general rule, it's better to define type names using `typedef` rather than `#define`.

## Unions

---

A *union*, like a structure, consists of one or more members, which may be of different types. However, the members of a union don't exist simultaneously; only one member at a time can have a meaningful value. Because of this property, the members of a union can be stored in the same memory locations, thereby saving space (the compiler allocates space only for the largest member of the union). Saving space is a primary reason for using a union instead of a structure.

We'll use a simple example to illustrate the basic properties of unions. Our union will contain just two members: `i` (of type `int`) and `f` (of type `float`). Here's the declaration of a union variable `u` with these members:

```
union {
    int i;
    float f;
} u;
```

Only the first member of a union can be initialized. For example, we can initialize the `i` member of `u` by writing

```
union {
    int i;
    float f;
} u = {0};
```

Notice the presence of the braces, even though the initializer is a single expression. The expression inside the braces must be constant.

Unions are closely related to structures; in fact, the compiler treats a union as a structure whose members overlap. Consequently, the properties of unions are almost identical to the properties of structures. For example, we can declare union tags and union types in the same way we declare structure tags and types. Also, we can access the members of a union using the same notation that we use to access the members of a structure. The following statement assigns the value 82 to the `i` member of `u`:

```
u.i = 82;
```

Similarly, the following statement assigns the value 74.8 to the `f` member of `u`:

```
u.f = 74.8;
```

Since the compiler overlays storage for the members of a union, storing a value in one member invalidates the values stored in the other members. In our example, `u.i` and `u.f` will have the same address in memory. If we store a value into `u.i` and then store a value into `u.f`, the value stored in `u.i` will be lost. (If we examine the value of `u.i`, it will appear to be meaningless.)

## Enumerations

---

An *enumeration* is a collection of integer values that have been given names by the programmer; these names are called *enumeration constants*. Enumerations are defined in a manner similar to structures and unions. The following example declares two variables, `s1` and `s2`, that can be assigned the values `small`, `medium`, `large`, and `extra_large`:

```
enum
{small, medium, large, extra_large}
s1, s2;
```

Alternatively, we could declare an enumeration tag, then use this tag to declare `s1` and `s2`:

```
enum sizes {small, medium, large, extra_large};
enum sizes s1, s2;
```

Or we could declare `sizes` to be a type:

```
typedef enum {small, medium, large, extra_large} sizes;
sizes s1, s2;
```

The names of enumeration constants must be different from each other and from other identifiers declared in the same scope.

Enumeration constants normally represent the integers 0, 1, 2, and so forth. For example, after the following declaration, `small`, `medium`, `large`, and `extra_large` represent the numbers 0, 1, 2, and 3, respectively:

```
enum sizes {small, medium, large, extra_large};
```

The programmer is free to choose a different representation:

```
enum sizes {
    small = 1,
    medium = 2,
    large = 3,
    extra_large = 4
};
```

(Two or more enumeration constants may represent the same number.) When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant. (If no value is specified for the first enumeration constant, it has the value 0.) Therefore, the following declaration is equivalent to the previous one:

```
enum sizes {
    small = 1,
    medium, large,
    extra_large
};
```

Enumeration constants are similar to constants created by using the `#define` directive. In fact, some programmers even write enumeration constants in upper case, just like macros that represent constant values:

```
typedef enum {FALSE, TRUE} boolean;
```

Creating an enumeration is an easy way to define a number of constants in a single step. The compiler will even generate their values automatically. Enumeration constants aren't equivalent to constants created using `#define`, however. For one thing, enumeration constants are subject to the same scope rules as variables.

Values of an enumeration may be mixed freely with ordinary integers:

```
int i;
enum {small, medium, large, extra_large} s;

i = medium; /* i is now 1 */
s = 0;      /* s is now small */
s++;       /* s is now medium */
i = s + 2;  /* i is now 3 */
```

The compiler treats `s` as an integer variable; the names `small`, `medium`, `large`, and `extra_large` are just synonyms for the numbers 0, 1, 2, and 3.

## Adding a “Tag Field” to a Union

---

There's no easy way to tell which member of a union was last assigned to and therefore contains a meaningful value. One way to retain this information is to embed the union within a structure that has two members: (1) the union itself, and (2) a “tag field” whose value indicates which member of the union was most recently assigned a value. Since the tag field has a relatively small number of possible values (one for each member in the union), it's usually a good idea to declare it as an enumeration.

Let's add a tag field to the union example that we studied earlier in the chapter:

```
struct {
    enum {
        int_kind,
        float_kind
    } kind; /* tag field */
    union {
        int i;
        float f;
    } u;
} s;
```

We've defined a structure variable named `s` that contains two members, `kind` and `u`. `kind` has two possible values, `int_kind` and `float_kind`, while `u` is a union with members `i` and `f`.

Each time we assign a value to a member of `u`, we'll also change `kind` to remind us which member of `u` was modified. For example, an assignment to the `i` member of `u` would have the following appearance:

```
s.kind = int_kind;  
s.u.i = 82;
```

Notice that assigning to `i` requires that we first select the `u` member of `s`, then the `i` member of `u`.

Whenever we need to retrieve the value of one of `u`'s members, we can use `kind` to check that the desired member was the last to be assigned a value. For example, the following if statement uses `kind` to determine how to print the value stored in `u`:

```
if (s.kind == int_kind)  
    printf("%d", s.u.i);  
else if (s.kind == float_kind)  
    printf("%g", s.u.f);  
else  
    printf("Illegal tag value");
```

# CHAPTER 13

## Advanced Uses of Pointers

In this chapter, we'll cover several advanced uses of pointers. First, we'll examine the role of pointers in dynamic storage allocation. We'll then see how pointers enable the construction of linked data structures—linked lists, in particular. We'll also discuss the concept of a pointer to a function; among other uses, a pointer to a function can be passed as a parameter to another function.

### Dynamic Storage Allocation

---

Many programs require *dynamic storage allocation*: the ability to allocate storage as needed during program execution.

Although storage for any kind of data object may be allocated dynamically, the objects most commonly allocated are strings, arrays, and structures. Dynamically allocated structures are often linked together to form lists, trees, and other data structures.

One way to allocate storage dynamically is to call the malloc function. The following declaration of malloc appears in <stdlib.h>:

```
void *malloc(size_t size);
```

malloc allocates a block of size bytes and returns a pointer to it. The memory allocated isn't cleared or otherwise initialized. If a block of the requested size isn't available, malloc returns a null pointer.

Notice the use of void \* as the return type for malloc. In ANSI C, void \* is a “generic” pointer type used when the type being pointed to isn't known. Note also the type size\_t, which some compilers (including TopSpeed C) define to be unsigned int and others define to be unsigned long int.

The only problem with malloc is that it must be given a number representing the size of the desired block. The size of an object in C can vary from one compiler to another and from one machine to another; for portability, we don't want to give malloc a constant as its parameter. Fortunately, C's built-in sizeof operator solves the problem. When applied to a type, sizeof returns the amount of storage required for a value of that type.

Here's how we would use malloc to allocate space for an integer:

```
int *p;  
  
p = malloc(sizeof(int));
```

The generic pointer returned by malloc is automatically converted to type `int *` when the assignment is performed; no cast is necessary. (In general, a value of type `void *` can be assigned to a variable of another pointer type and vice-versa. Values of type `void *` can also be tested for equality or inequality with pointers of other types.)

To release a block of memory allocated by malloc, we can call the free function. The following declaration of free is in `<stdlib.h>`:

```
void free(void *ptr);
```

Since ptr is of type `void *`, a pointer of any type may appear as an actual parameter in a call of free.

malloc and free operate on a storage pool known as the *heap*. Calls of malloc remove blocks of memory from the heap; calls of free return these blocks. Blocks that have been returned to the heap can be reused in subsequent calls of malloc. If malloc is called often or asked for large blocks of memory the heap can be exhausted. Calling free to return blocks that are no longer needed can prevent or at least postpone this event.

## **Dynamically Allocated Strings**

Dynamic storage allocation is often useful in programs that perform string manipulation. For example, suppose that we want to concatenate strings s1 and s2, storing the resulting string in as little space as possible. We first measure the lengths of s1 and s2, then call malloc to allocate just the right amount of space for the result:

```
char *result;

result = malloc(strlen(s1) + strlen(s2) + 1);
strcpy(result, s1);
strcat(result, s2);
```

Warning:

When using malloc to allocate space for a string, don't forget to include room for the null character.

## **Dynamically Allocated Arrays**

It's sometimes difficult to estimate the proper size for an array; it would be convenient to wait until the program is executed to decide how large the array should be. C's mechanism for dynamic storage allocation solves this problem: the programmer need only declare a pointer variable to point to the beginning of the array, then during execution have the program dynamically allocate space for the array itself.

Space for a string is allocated by calling malloc. Space for other kinds of arrays is allocated by a function named calloc. The following declaration of calloc appears in `<stdlib.h>`:

```
void *calloc(size_t nmemb, size_t size);
```

`calloc` allocates space for an array with `nmemb` elements, each of which has size `size`; it returns a null pointer if the requested space isn't available. Unlike `malloc`, `calloc` initializes the block of memory that it finds by setting all bits to 0.

Suppose we're writing a program that needs an array of `n` integers, where `n` is to be computed during the execution of the program. We first declare a pointer variable:

```
int **a;
```

Once the value of `n` is known, the program will call `calloc` to allocate a block of `n` integers:

```
a = calloc(n, sizeof(int));
```

From this point on, we can forget that `a` is a pointer and use it instead as an array name (because of the close relationship between arrays and pointers in C). For example, we could use the following loop to initialize `a`:

```
for (i = 0; i < n; i++)  
    a[i] = i;
```

Of course, we can use pointer arithmetic instead of subscripting to access array elements.

A dynamically allocated array that's no longer needed can be de-allocated by calling `free`.

**Warning:**

The parameter to `free` must be a pointer that was previously returned by `malloc` or `calloc`. Calling `free` with any other parameter may cause unpredictable behavior.

## Linked Lists

---

Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures. A linked structure consists of a collection of *nodes*. Each node contains one or more pointers to other nodes.

The simplest linked structure is the linked list, which consists of a chain of nodes, with each node pointing to the next node in the chain. A node in a linked list might have the following declaration:

```
struct node {  
    int value; /* data stored in the node */  
    struct node *next; /* pointer to the */  
                    /* next node */  
};
```

As we noted in Chapter 12, the use of a structure tag is mandatory in the declaration of a structure that contains a reference to itself. Without the structure tag, we would have no way to declare the type of the next member.

An ordinary pointer variable will point to the first node in the list:

```
struct node *first = NULL;
```

We've initialized `first` to `NULL` to indicate that the list is empty initially.

To make `first` point to an actual node, we must allocate space for the node by calling `malloc`:

```
first = malloc(sizeof(struct node));
```

Next, we should initialize the node's value and next members. One way to initialize the value member is to write

```
(*first).value = 1;
```

Here we've applied the indirection operator `*` (to reference the structure to which `first` points), then the selection operator `.` (to select a member within this structure).

Because pointers often point to structures, C provides a simpler alternative—the `->` operator—for accessing the members of these structures. Using this operator, we could write

```
first->value = 1;
```

The `->` operator is a combination of the `*` and `.` operators; it performs indirection on `list` to locate the structure that it points to, then selects the value member of the structure. Note that `->` produces an lvalue.

Let's look next at implementations of three common list operations: inserting a node at the beginning of a list, searching for a node, and deleting a node.

### **Inserting a Node at the Beginning of a Linked List**

The following function inserts a node at the beginning of a list. It first calls `malloc` to allocate space for the node, then it stores the desired information into the node, and finally it makes the node's next member point to the node that was formerly at the beginning of the list. The parameter `list` is a pointer to the first node in the list (or a null pointer if the list is empty) and `i` is the value to be put in the new node. `add_to_list` returns a pointer to the new list.

```
struct node *add_to_list(
    struct node *list,
    int i)
{
    struct node *temp;

    temp = malloc(sizeof(struct node));
    temp->value = i;
    temp->next = list;
    return temp;
}
```

A call of `add_to_list` might look like this:

```
first = add_to_list(first, 10);
```



The following example uses `add_to_list` to create a linked list containing numbers entered by the user:

```
struct node *first = NULL;
int n;

printf(
    "Enter a series of numbers "
    "(enter 0 to stop): ");
scanf("%d", &n);
while (n != 0) {
    first = add_to_list(first, n);
    scanf("%d", &n);
}
```

Notice that the numbers will appear in the list in the reverse of the order in which they were entered.

### **Searching a Linked List**

Now let's examine a function that searches a list for a node containing a particular data value. The function `search_list` searches a list (pointed to by the parameter `list`) for an integer `i`. If it finds `i`, `search_list` returns a pointer to the node containing `i`; otherwise, `search_list` returns a null pointer.

```
struct node *search_list(
    struct node *list,
    int i)
{
    for ( ;
        list != NULL && list->value != i;
        list = list->next)
        ;
    return list;
}
```

`search_list` takes advantage of the fact that changing the value of `list` doesn't affect the corresponding actual parameter. For example, the call

```
p = search_list(first, 10);
```

doesn't change the value of `first`.

### **Deleting a Node from a Linked List**

Deleting a node is more complicated than inserting a node or searching for a node. As we search the list to locate the node to delete, we'll always keep a pointer to the preceding node. Once we've found the node to be deleted, we make the next member in the *preceding* node point to the *following* node.

The `delete_from_list` function uses this strategy. When given a list and an integer `i`, `delete_from_list` deletes the first node containing `i`. If no node contains `i`, the function does nothing. In either case, the function returns a pointer to the list.

```

struct node *delete_from_list(struct node *list, int i)
{
    struct node *p, *q;

    for (
        p = list, q = NULL;
        p != NULL && p->value != i;
        q = p, p = p->next);

    if (p == NULL)
        return list; /* i was not found */
    if (q == NULL)
        list = list->next; /* i is in the */
                          /* first node */
    else
        q->next = p->next; /* i is in some */
                          /* other node */
    free(p);
    return list;
}

```

`delete_from_list` illustrates the power and flexibility of C's `for` statement. The rather exotic `for` statement in `delete_from_list` with its empty body and liberal use of the comma operator performs all the actions needed to search for `i`. When the loop terminates, the variable `p` points to the node to be deleted, while `q` points to the preceding node.

### **Example Program: Maintaining a Parts Database (Revisited)**

Let's redo the parts database program of Chapter 12, this time storing the database in a linked list instead of an array. The part structure will now contain an additional member (a pointer to the next node in the linked list), and the variable `inventory` will be a pointer to the first node in the list:

```

struct part {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
    struct part *next;
};

struct part *inventory = NULL; /* points to */
                               /* first part */

```

Using a linked list instead of an array has two major advantages: (1) We don't need to put a predetermined limit on the size of the database; it can grow until there's no more memory to store parts. (2) We can easily keep the database sorted by part number when a new part is added to the database, we simply insert it in its proper place in the list. (Notice that we're changing the behavior of the program slightly. In the old program, the database wasn't sorted, so the `p` operation displayed parts in the order in which they were entered.)

Here's the new program:

```

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

#define NAME_LEN 25

struct part {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
    struct part *next;
};

struct part *inventory = NULL; /* points to */
/* first part */
struct part *find_part(int part_no);
struct part *read_part(void);
void insert(void);
void search(void);
void update(void);
void print(void);

/*****
/* main: Prompts the user to enter an */
/* operation code, then calls a function to */
/* perform the requested action. Repeats */
/* until the user enters the command 'q */
/* Prints an error message if the user enters*/
/* an illegal code. */
*****/

main()
{
    char code;

    for (;;) {
        printf("Enter operation code: ");
        scanf(" %c", &code);
        switch (code) {
            case 'i': insert(); break;
            case 's': search(); break;
            case 'u': update(); break;
            case 'p': print(); break;
            case 'q': exit(EXIT_SUCCESS);
            default: printf("Illegal code\n");
        }
        printf("\n");
    }
}

/*****
/* find_part: Looks up a part number in the */
/* inventory list. Returns a pointer to the */ /* node containing the
part number; if the */
/* part number is not found, returns a null */ /* pointer.
*/
*****/

```

```

struct part *find_part(int part_no)
{
    struct part *p;

    for (
        p = inventory;
        p != NULL && part_no > p->part_no;
        p = p->next);
    if (p != NULL && part_no == p->part_no)
        return p;
    return NULL;
}

/*****
/* read_part: Dynamically allocates a part */
/* structure and prompts the user to enter a */
/* part number, part name, and quantity on */
/* hand. (Prints an error message and returns */
/* a null pointer if space cannot be */
/* allocated.) Returns a pointer to the new */
/* structure. */
*****/

struct part *read_part(void)
{
    struct part *p;

    p = malloc(sizeof(struct part));
    if (p == NULL) {
        printf(
            "Database is full; "
            "no more parts can be added.\n");
        return NULL;
    }
    printf("Enter part number: ");
    scanf("%d", &p->part_no);
    printf("Enter part name: ");
    scanf(" ");
    gets(p->part_name);
    printf("Enter quantity on hand: ");
    scanf("%d", &p->on_hand);
    return p;
}

/*****
/* insert: Calls read_part to obtain */
/* information about a new part, then inserts */
/* the part into the inventory list; the list */
/* remains sorted by part number. Returns */
/* prematurely if space for the part could */
/* not be allocated or if the part already */
/* exists (prints an error message in the */
/* latter case). */
*****/

```

```

void insert(void)
{
    struct part *p, *q, *new;

    new = read_part();
    if (new == NULL) return;
    for (p = inventory, q = NULL;
         p != NULL && new->part_no > p->part_no;
         q = p, p = p->next);
    if (p != NULL && new->part_no == p->part_no)
    {
        printf("Part already exists.\n");
        free(new);
        return;
    }
    new->next = p;
    if (q == NULL) inventory = new;
    else q->next = new;
}

/*****
/* search: Prompts the user to enter a part */
/* number, then looks up the part in the */
/* database. If the part exists, prints the */
/* name and quantity on hand; if not, prints */
/* an error message. */
*****/

search(void)
{
    int part_no;
    struct part *p;

    printf("Enter part number: ");
    scanf("%d", &part_no);
    p = find_part(part_no);
    if (p != NULL) {
        printf(
            "Part name: %s\n",
            p->part_name);
        printf(
            "Quantity on hand: %d\n",
            p->on_hand);
    } else
        printf("Part not found.\n");
}

/*****
/* update: Prompts the user to enter a part */
/* number. Prints an error message if the */
/* part doesn't exist; otherwise, prompts the */
/* user to enter change in quantity on hand */
/* and updates the database. */
*****/

```

```

void update(void)
{
    int part_no, change;
    struct part *p;

    printf("Enter part number: ");
    scanf("%d", &part_no);
    p = find_part(part_no);
    if (p != NULL) {
        printf(
            "Enter change in "
            "quantity on hand: ");
        scanf("%d", &change);
        p->on_hand += change;
    } else
        printf("Part not found.\n");
}

/*****
/* print: Prints a listing of all parts in
/* the database, showing the part number,
/* part name, and quantity on hand. Part
/* numbers will appear in increasing order.
*****/

void print(void)
{
    struct part *p;

    printf(
        "Part Number    Part Name"
        "          Quantity on Hand\n");
    for (p = inventory; p != NULL; p = p->next) {
        printf("%7d        ", p->part_no);
        printf("%-25s", p->part_name);
        printf("%11d\n", p->on_hand);
    }
}

```

This program differs in several respects from the one in Chapter 12:

- The `find_part` function returns a pointer to the node that contains the desired part number. If it doesn't find the part number, `find_part` returns a null pointer. Since the inventory list is sorted by part number, `find_part` stops searching when it finds a node containing a part number that is greater than or equal to the desired part number.
- `read_part` allocates space for a new part node, stores the necessary information in the node, and returns a pointer to the node.
- `insert` searches the inventory list for a node whose part number is greater than or equal to the new part number. If it finds a node with the *same* part number, `insert` prints the "Part already exists" message. If it finds a node with a *larger* part number (or if it reaches the end of the list), `insert` adds the new node to the list, immediately preceding the node with the larger part number.

## Pointers to Functions

---

In C, functions can't be treated as data. It's not possible to store a function in a variable, nor can we write a function that takes another function as a parameter or returns a function. However, we can accomplish all of these goals using *pointers* to functions, since pointers can be stored in variables, used as parameters, and returned by functions.

Passing a pointer to a function as a parameter is a fairly common practice in C. For example, suppose that we're writing a function named `integrate` that integrates a function `f` between points `a` and `b`. To make `integrate` as general as possible, we'd prefer to supply `f` (as well as `a` and `b`) as a parameter to `integrate`. To achieve this effect in C, we declare `f` to be a pointer to a function. Assuming that `f` will point to a function that has a double parameter and returns a double result, the declaration of `integrate` will look like this:

```
double integrate(  
    double (*f)(double),  
    double a,  
    double b);
```

The parentheses around `*f` indicate that `f` is a pointer to a function, not a function that returns a pointer. It's also legal to declare `f` as though it were a function:

```
double integrate(  
    double f(double),  
    double a,  
    double b);
```

From the compiler's standpoint, this declaration is identical to the previous one.

When we call `integrate`, we can supply a function name as the first parameter:

```
result = integrate(g, 10.0, 20.0);
```

Normally, each occurrence of a function name in a statement is followed by parentheses, indicating a call of the function. When the name isn't followed by parentheses, the compiler produces a pointer to the function instead of generating code for a function call. In our example, `g` isn't called; instead, a pointer to `g` is passed to `integrate`.

Within the body of `integrate`, here's how we would call the function that `f` points to:

```
sum += (*f)(x);
```

(In ANSI C, it's legal to write `f(x)` instead of `(*f)(x)`.) The compiler treats `(*f)(x)` as a call of the function to which `f` points. For example, during the execution of `integrate(g, 10.0, 20.0)`, each call of `*f` is actually a call of `g`.

A pointer to a function can be used just like any other pointer. In particular, a pointer to a function may be stored in a variable or returned by a function. More exotic uses—*an array of pointers to functions, for example*—are also possible.

## **The qsort Function**

Certain functions in the C library require a function pointer as a parameter. One of the most commonly used is `qsort`, whose declaration can be found in `<stdlib.h>`. `qsort` is a general-purpose function capable of sorting *any* array.

Since the elements of the array may be of any type—even a structure or union type—`qsort` must be told how to determine which of two array elements is “smaller.” `qsort` is given this information in the form of a “comparison function”; when given two pointers `p` and `q` to array elements, the comparison function must return a number that is *negative* if `*p` is “less than” `*q`, *zero* if `*p` is “equal to” `*q`, and *positive* if `*p` is “greater than” `*q`. The words “less than,” “equal to,” and “greater than” are in quotes because it’s the programmer’s responsibility to determine how `*p` and `*q` are compared.

`qsort` has the following declaration:

```
void qsort(
    void *base,
    size_t nmemb,
    size_t size,
    int (*compar)(const void *, const void *));
```

`base` is a pointer to the first element in the array, `nmemb` is the number of elements to be sorted, `size` is the size of each array element, and `compar` is a pointer to the comparison function. When `qsort` is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.

To sort the inventory array of Chapter 12, we would use the call

```
qsort(
    inventory,
    num_parts,
    sizeof(struct part), compare_parts);
```

where `compare_parts` is the following function:

```
int compare_parts(
    const struct part *p,
    const struct part *q)
{
    if (p->part_no < q->part_no)
        return -1;
    else if (p->part_no == q->part_no)
        return 0;
    else
        return 1;
}
```



## **Example Program: Tabulating the Trigonometric Functions**

The following program prints a table showing the values of cos, sin, or tan (all three functions are declared in `<math.h>`). The user chooses one of the three functions and specifies the range of values to which the function should be applied.

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

void tabulate(
    double (*f)(double),
    double first,
    double last,
    double incr);
main()
{
    char function;
    double final, increment, initial;

    printf("Enter initial value: ");
    scanf("%lf", &initial);
    printf("Enter final value: ");
    scanf("%lf", &final);
    printf("Enter increment: ");
    scanf("%lf", &increment);

    printf(
        "Enter function code "
        "(c = cos, s = sin, t = tan): ");
    for (;;) {
        scanf(" %c", &function);
        switch (function) {
            case 'c':
                tabulate(
                    cos,
                    initial,
                    final,
                    increment);
                exit(EXIT_SUCCESS);
            case 's':
                tabulate(
                    sin,
                    initial,
                    final,
                    increment);
                exit(EXIT_SUCCESS);
            case 't':
                tabulate(
                    tan,
                    initial,
                    final,
                    increment);
                exit(EXIT_SUCCESS);
```

```

        default:
            printf(
                "Illegal function code; "
                "please re-enter: ");
    }
}

void tabulate(double (*f)(double), double first, double last, double incr)
{
    double x;
    int i, num_intervals;

    printf("\n      x      f(x)\n");
    printf("  ————\n");

    num_intervals = ceil((last - first) / incr);
    for (i = 0; i <= num_intervals; i++) {
        x = first + i * incr;
        printf("%10.5f %10.5f\n", x, (*f)(x));
    }
}

```

When given a parameter  $x$  of type double, the `ceil` function declared in `<math.h>` returns the smallest integer that is greater than or equal to  $x$ .

# CHAPTER 14

## File I/O

In previous chapters, we've discussed several of the functions in the standard I/O library, including `printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`. In this chapter, we'll see how to use redirection to make these functions write to a file or read from a file. We'll then turn to functions that are designed specifically for use with files. First, we'll introduce functions that open and close files. Next, we'll cover functions that read and write files containing human-readable text. We'll then discuss how to read and write files of binary data. Finally, we'll see how to perform random-access operations on files.

### Streams

---

In C, the term *stream* means any input source or output destination. Small programs, like the examples in previous chapters, often use just two streams—a single input stream (the user's keyboard) and a single output stream (the user's screen). Larger programs might need additional input and/or output streams, usually representing disk files.

In a program, each stream is represented by a *file pointer*. A file pointer is a value of type `FILE *` (`FILE` is defined in `<stdio.h>`). Certain streams are represented by file pointers with standard names; the programmer may also declare additional file pointers as needed. For example, if a program needs two streams in addition to the standard ones, it might include the following declaration:

```
FILE *fp1, *fp2;
```

A program may contain any number of variables of type `FILE *`, although there's a system-defined limit on the number of streams that can be open simultaneously.

There are three standard streams in ANSI C; their file pointers have the following names:

<code>stdin</code>	standard input
<code>stdout</code>	standard output
<code>stderr</code>	standard error

These streams need not be declared nor must they be opened or closed.

The I/O functions that we've used in previous chapters—`printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`—obtain input from `stdin` and send output to `stdout`. By default, `stdin` represents the user's keyboard and `stdout` and `stderr` represent the user's screen. However, these default meanings may be changed by using *redirection* when the program is run. The method of redirection depends on the operating system.

Under MS-DOS, `stdin` can be redirected to a file by putting the file name on the command line (the line entered by the user to execute the program), preceded by the `<` symbol. Similarly, `stdout` can be redirected to a file by putting the file name on the command line, preceded by the `>` symbol. For example, the following command line runs the program `prog`, specifying that input to `prog` will come from the file `in.dat` and output will be written to the file `out.dat`:

```
prog <in.dat >out.dat
```

Redirection of both streams isn't required; we can redirect either `stdin` or `stdout` by itself.

The usefulness of `stderr` is now apparent. By writing error messages to `stderr` instead of `stdout`, a program can guarantee that these messages will appear on the user's screen even when `stdout` has been redirected.

## Opening and Closing Files

---

In addition to `stdin`, `stdout`, and `stderr`, a program may have other input and output streams. These streams, unlike the three standard streams, must be explicitly opened by the program before use and closed before program termination.

Opening a file for use as a stream requires a call of `fopen`, which has the following declaration:

```
FILE *fopen(const char *filename, const char *mode);
```

The first parameter is the name of the file to be opened (possibly including a drive specifier and/or path). The second parameter must be one of the following strings:

"r"	open for reading
"w"	open for writing (file need not exist)
"a"	open for appending (file need not exist)
"r+"	open for reading and writing, starting at beginning
"w+"	open for reading and writing (truncate if file exists)
"a+"	open for reading and writing (append if file exists)

If the file can't be opened, `fopen` returns a null pointer. Otherwise, it returns a file pointer that should be stored for use whenever a subsequent operation is performed on the file.

**Warning:** Be careful when the file name in a call of `fopen` includes the `\` symbol, since the compiler treats an appearance of `\` in a string literal as the beginning of a character escape.

For example, the call

```
fopen("a:\project\test1.dat", "r")
```

will always return a null pointer, because the compiler treats `\p` and `\t` as character escapes. The problem can be avoided by using `\\` instead of `\`:

```
fopen("a:\\project\\test1.dat", "r")
```

Files can be closed by calling `fclose`, which has the following declaration:

```
int fclose(FILE *stream);
```

`fclose` returns 0 if the file was closed successfully; otherwise, it returns EOF (a macro defined in `<stdio.h>`).

To show how `fopen` and `fclose` are used in practice, here's the outline of a program that opens the file `example.dat` for reading, then closes it before terminating:

```
#include <stdio.h>

main()
{
    FILE *fp;

    fp = fopen("example.dat", "r");
    ...
    fclose(fp);
}
```

## **Program Parameters**

There are two obvious ways for a program to obtain the name of a file. The file name could be included in the program as a string literal, or the program could read the file name (using `gets` or `scanf`, for example). There's a third possibility, however: the file name can be put on the command line and then retrieved by the program.

For example, suppose that we compile and link a program named `demo`. When we execute `demo`, we might put a list of file names or other information on the command line:

```
demo names.dat dates.dat
```

To gain access to these *program parameters* (also known as *command-line arguments*), we must include the parameters `argc` and `argv` in `main`'s parameter list:

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

The value of `argc` (“argument count”) is the number of program parameters (including the name of the program itself). `argv` (“argument vector”) is an array of pointers to the program parameters, which are stored in string form. `argv[0]` points to the name of the program. `argv[1]` through `argv[argc-1]` point to the remaining program parameters. `argv[argc]` is always a null pointer.

In the example above, `argc` is 3, `argv[0]` points to the string “demo”, `argv[1]` points to the string “names.dat”, and `argv[2]` points to the string “dates.dat”.

### **Example Program: Checking for the Existence of a File**

The following program (let's call it `exist`) determines if a file exists by trying to open it. When the program is run, the user will give it a file name to check:

```
exist fl.dat
```

The program will then print either “fl.dat exists” or “fl.dat does not exist”. If the user enters the wrong number of arguments on the command line, the program will print the message “usage: exist filename” to remind the user that `exist` requires a single file name.

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

main(int argc, char *argv[])
{
    FILE *fp;

    if (argc != 2) {
        printf("usage: exist filename\n");
        exit(EXIT_FAILURE);
    }
    if ((fp = fopen(argv[1], "r")) == NULL)
        printf("%s does not exist\n", argv[1]);
    else {
        printf("%s exists\n", argv[1]);
        fclose(fp);
    }
}
```

## **Text I/O**

---

C's standard I/O library supports both text streams and binary streams. A *text stream* (such as a file containing the source for a C program) consists of

human-readable data. A *binary stream* (an executable file, for example) consists of data that isn't in human-readable form. From the standpoint of the I/O library, a text stream consists of lines of characters, each terminated by the newline character. A binary stream, on the other hand, consists of a sequence of arbitrary bytes; newline characters have no special significance.

In this section, we'll examine library functions that operate on text streams. Notice that these functions treat characters as values of type `int`. One reason is that some functions indicate an end-of-file (or error) condition by returning EOF, which is a negative integer constant. Returning a `char` value wouldn't work, since some compilers treat `char` as an unsigned type, so these functions return an `int` value instead.

## **Output Functions**

Let's look first at some of the output functions that `<stdio.h>` provides for text streams.

`fputc`, `putc`, and `putchar` write a single character to an output stream. All three functions return EOF if an error occurs; otherwise, they return the character that was written.

- `int fputc(int c, FILE *stream);`

Writes the character `c` to stream.

- `int putc(int c, FILE *stream);`

Equivalent to `fputc` but usually implemented as a macro.

- `int putchar(int c);`

Writes the character `c` to stdout. The call `putchar(c)` is equivalent to `putc(c, stdout)`. `putchar` is usually implemented as a macro.

`fputs` and `puts` write a string. Both functions return EOF if an error occurred; otherwise, they return a nonnegative number.

- `int fputs(const char *s, FILE *stream);`

Writes the string `s` to stream. `fputs` doesn't write a newline character unless one is present in `s`.

- `int puts(const char *s);`

Writes the string `s` to stdout. `puts` writes a newline character to stdout after the last character in `s`.

`fprintf` and `printf` write a variable number of data items to an output stream, using a format string to control the appearance of the output. Both functions return the number of characters written; a negative return value indicates that an error occurred.

- `int fprintf(FILE *stream, const char *format, ...);`  
Writes any number of data items to stream, using format to control the appearance of the output.
- `int printf(const char *format, ...);`  
Writes any number of data items to stdout, using format to control the appearance of the output. A call of `printf` is equivalent to a call of `fprintf` with stdout as the first parameter.

## **Input Functions**

Now we'll look at the input functions for text streams. These are somewhat more complicated than the output functions because of the possibility of reaching the end of a file or encountering badly formed input.

`fgetc`, `getc`, and `getchar` read a single character from an input stream. All three functions return EOF if the end of the input file is reached or if an error occurs; otherwise, they return the character that was read.

- `int fgetc(FILE *stream);`  
Reads a character from stream.
- `int getc(FILE *stream);`  
Equivalent to `fgetc` but usually implemented as a macro.
- `int getchar(void);`  
Reads a character from stdin. The call `getchar()` is equivalent to `getc(stdin)`. `getchar` is usually implemented as a macro.

`fgets` and `gets` read a string. Both functions return a null pointer if the end of the input file is reached or an error occurs; otherwise, both return a pointer to the string read. Also, both functions store a null character at the end of the string.

- `char *fgets(char *s, int n, FILE *stream);`  
Reads from stream into the array that `s` points to, stopping at the first newline character or when `n - 1` characters have been read, whichever happens first. The newline character, if encountered, is stored in the array.
- `char *gets(char *s);`  
Similar to `fgets` but reads characters from stdin until it encounters a newline character (the newline character is *not* stored). (Note the possibility of stepping outside the bounds of the array; `fgets` is safer.)

`fscanf` and `scanf` read a variable number of data items from an input stream, using a format string to indicate the layout of the input. Both functions return the number of data items that were read. (This may be less than the



number requested if the end of the file was reached, if an error occurred, or if some data item didn't match the corresponding conversion specification.) Both functions return EOF if the end of the input file was reached (or an error occurred) before any data items could be read.

- `int fscanf(FILE *stream, const char *format, ...);`  
Reads any number of data items from stream, using format to indicate the layout of the input.
- `int scanf(const char *format, ...);`  
Reads any number of data items from stdin, using format to indicate the layout of the input. A call of `scanf` is equivalent to a call of `fscanf` with `stdin` as the first parameter.

### **Detecting End-of-File and Error Conditions**

Notice that the input functions return the same value if an error occurs as they do if the end of the input file is reached. Fortunately, the `feof` and `ferror` functions allow a program to determine which event occurred:

- `int feof(FILE *stream);`  
Returns a nonzero value if the end of stream has been reached.
- `int ferror(FILE *stream);`  
Returns a nonzero value if an error occurred during an operation on stream.

### **Example Program: Copying a Text File**

The following program copies one text file to another. The names of the original file and the new file will be specified on the command line when the program is executed. For example, if the program is named `fcopy`, the command

```
fcopy f1.c f2.c
```

will copy the file `f1.c` to `f2.c`. `fcopy` will issue an error message if there aren't two file names on the command line or if either file can't be opened.

```

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

main(int argc, char *argv[])
{
    FILE *source_fp, *dest_fp;
    int ch;

    if (argc != 3) {
        printf("Must specify two file names\n");
        exit(EXIT_FAILURE);
    }
    if ((source_fp = fopen(argv[1], "r")) == NULL)
    {
        printf("Can't open source file\n");
        exit(EXIT_FAILURE);
    }
    if ((dest_fp = fopen(argv[2], "w")) == NULL)
    {
        printf("Can't open destination file\n");
        fclose(source_fp);
        exit(EXIT_FAILURE);
    }
    while ((ch = getc(source_fp)) != EOF)
        putc(ch, dest_fp);
    fclose(source_fp);
    fclose(dest_fp);
}

```

## Binary I/O

---

The most important binary I/O functions are `fread` and `fwrite`, which allow a program to read and write blocks of binary data.

`fread` reads any number of equal-size blocks from a binary file, storing them into an array specified by the program. Here's the declaration of `fread`:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

`fread` reads up to `nmemb` elements of `size` from `stream`, storing them in the array specified by `ptr`. The return value is the actual number of elements read. (This number should equal `nmemb` unless the end of the input file was reached or an error occurred. The `feof` and `ferror` functions can be used to determine the reason for any shortage.)

`fwrite` is similar to `fread`:

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

`fwrite` writes up to `nmemb` elements of `size` to `stream`, reading them from the array specified by `ptr`. `fwrite` returns the actual number of elements written, which should equal `nmemb` unless an error occurred.

**Note:** Before performing the `fread` or `fwrite` operations on a file, a program must first use `fopen` to open the file; the mode parameter must include the letter `b`:

“rb”	open for reading
“wb”	open for writing (file need not exist)
“ab”	open for appending (file need not exist)
“r+b” or “rb+”	open for reading and writing, starting at beginning
“w+b” or “wb+”	open for reading and writing (truncate if file exists)
“a+b” or “ab+”	open for reading and writing (append if file exists)

## Random Access

---

The `fseek`, `ftell`, and `rewind` functions support random access within binary files. These functions can also be used with text files, but some restrictions apply.

`fseek` allows repositioning within a file:

```
int fseek(FILE *stream, long int offset, int whence);
```

The new file position within stream is determined by offset and whence. offset is a (possibly negative) byte count relative to whence, which must be one of the following:

```
SEEK_SET  beginning of file
SEEK_CUR  current file position
SEEK_END  end of file
```

Normally, `fseek` returns 0. If an error occurs (for example, if the requested position doesn't exist), `fseek` returns a nonzero value.

The following examples show various uses of `fseek`:

```
fseek(fp, 0L, SEEK_SET); /* move to start */
/* of file */
fseek(fp, -10L, SEEK_CUR); /* move back */
/* 10 bytes */
fseek(fp, 0L, SEEK_END); /* move to end */
/* of file */
```

`ftell` returns the current file position as a long integer:

```
long int ftell(FILE *stream);
```

If an error occurs, `ftell` returns `Æ1L`. The value returned by `ftell` may be saved and later supplied to a call of `fseek`:

```

long int file_pos;
...
file_pos = ftell(fp); /* save current */
                    /* position      */
...
fseek(fp, file_pos, SEEK_SET); /* return to */
                              /* saved position */

```

The `rewind` function positions a file to its beginning:

```
void rewind(FILE *stream);
```

The call `rewind(fp)` is equivalent to `fseek(fp, 0L, SEEK_SET)`, except that `rewind` doesn't return a value.

### **Example Program: Modifying a File of Part Records**

The following program reads a binary file of part records into the array `inventory`, sets the `on_hand` member of each record to 0, then writes the records back to the file. Notice that the file is opened for both reading and writing ("`rb+`").

```

#include <stdio.h>
#include <stdlib.h>
#define NAME_LEN 25
#define MAX_PARTS 100
struct part {
    int part_no;
    char part_name[NAME_LEN+1];
    int on_hand;
} inventory[MAX_PARTS];

int num_parts;
main()
{
    FILE *fp;
    int i;

    if ((fp = fopen("invent.dat", "rb+"))
        == NULL) {
        printf("Can't open inventory file\n");
        exit(EXIT_FAILURE);
    }
    num_parts =
        fread(
            inventory,
            sizeof(struct part),
            MAX_PARTS,
            fp);
    for (i = 0; i < num_parts; i++)
        inventory[i].on_hand = 0;
    rewind(fp);
    fwrite(
        inventory,
        sizeof(struct part),
        num_parts,
        fp);
    fclose(fp);
}

```

# CHAPTER 15

## Low-Level Programming

Previous chapters have described C's high-level, machine-independent features. Although these features are adequate for many applications, some programs require machine-level operations not normally supported by high-level programming languages. In particular, certain programs must be able to access bits and bit-fields—units of storage that aren't meaningful in most high-level languages—and perform bitwise Boolean operations and shifting. These “low-level” operations are particularly useful in programs for which fast execution or efficient use of space is critical.

In this chapter, we'll discuss C's bitwise operators, which provide easy access to both individual bits and bit-fields. We'll also see how to declare structures that contain bit-fields.

### Bitwise Operators

C provides six bitwise operators, which operate on integer operands at the bit level. (Character operands are also allowed, since they're promoted to an integer type.) We'll discuss the bitwise shift operators first.

### Bitwise Shift Operators

Some programs need to perform a shifting operation on an integer—to transform the binary representation of the number by shifting its bits to the left or right.

C provides two bitwise shift operators:

```
<< left shift  
>> right shift
```

The operands for << and >> may be of any integer or character type. The integral promotions are performed on both operands; the result has the type of the left operand after promotion.

The value of `i << j` is the result when `i` is shifted left `j` positions. Zero bits are added at the right end to replace bits that are “shifted off.”

The value of `i >> j` is the result when `i` is shifted right `j` positions. If `i` is of an unsigned type or if the value of `i` is nonnegative, then zero bits are added at the left as needed. If `i` is a negative number, the result depends on the implementation. In TopSpeed C, the sign bit is propagated, so the number remains negative.

The following examples illustrate the effect of the shift operators. Note that neither operator modifies its operands.

```

int i, j;

i = 13;
/* i is now 13 (binary 0000000000001101) */

j = i << 2;
/* j is now 52 (binary 0000000000110100) */

j = i >> 2;
/* j is now 3 (binary 0000000000000011) */

```

The compound assignment operators `<<=` and `>>=` correspond to the shift operators `<<` and `>>`, as the following examples show:

```

i = 13;
/* i is now 13 (binary 0000000000001101) */
i <<= 2;
/* i is now 52 (binary 0000000000110100) */
i >>= 2;
/* i is now 13 (binary 0000000000001101) */

```

**Note:** The bitwise shift operators have lower precedence than the arithmetic operators. For example, `i << 2 + 1` means `i << (2 + 1)`, not `(i << 2) + 1`.

### Bitwise Complement, And, Exclusive Or, and Inclusive Or

In addition to the bitwise shift operators, C provides four other bitwise operators:

<code>~</code>	bitwise complement
<code>&amp;</code>	bitwise <i>and</i>
<code>^</code>	bitwise exclusive <i>or</i>
<code> </code>	bitwise inclusive <i>or</i>

The `~` operator is unary; the integral promotions are performed on its operand. The other operators are binary; the usual arithmetic conversions are performed on their operands.

The following examples illustrate the effect of these operators:

```

int i, j, k;

i = 21;
/* i is now 21 (binary 0000000000010101) */
j = 56;
/* j is now 56 (binary 0000000000111000) */
k = ~i;
/* k is now -22 (binary 111111111101010) */
k = i & j;
/* k is now 16 (binary 0000000000010000) */
k = i ^ j;
/* k is now 45 (binary 0000000000101101) */
k = i | j;
/* k is now 61 (binary 0000000000111101) */

```

These examples assume two's-complement arithmetic—the kind performed by the members of the IBM PC family.

**Warning:**

**Don't confuse the bitwise operators & and | with the logical operators && and ||. The bitwise operators may sometimes produce the same results as the logical operators, but they aren't equivalent.**

The ~, &, ^, and | operators have the following precedence:

```
highest: ~
&
^
lowest: |
```

For example, `i & ~j | k` means `(i & (~j)) | k` and `i ^ j & ~k` means `i ^ (j & (~k))`.

**Warning:**

**The precedence of &, ^, and | is lower than the precedence of the relational and equality operators. As a result, the following statement won't have the desired effect:**

```
if (status & 0400 != 0) ...
```

**Instead of testing whether `status & 0400` isn't zero, this statement will evaluate `0400 != 0` (which has the value 1), then test whether the value of `status & 1` isn't zero.**

The compound assignment operators `&=`, `^=`, and `|=` correspond to the bitwise operators `&`, `^`, and `|`:

```
i = 21;
/* i is now 21 (binary 0000000000010101) */
j = 56;
/* j is now 56 (binary 0000000000111000) */
i &= j;
/* i is now 16 (binary 0000000000010000) */
i ^= j;
/* i is now 40 (binary 0000000000101000) */
i |= j;
/* i is now 56 (binary 0000000000111000) */
```

## Using the Bitwise Operators to Access Bits and Bit-Fields

The bitwise operators make it easy to perform operations on individual bits:

- *Setting a bit.* Suppose that we want to set bit 4 of the integer variable `i`. (On the IBM PC, integers are 16 bits long. Let's assume that the most significant bit is numbered 15 and the least significant is numbered 0.) The following statement will do just that:

```
i |= 0x0010;
```

The number 0x0010 is a “mask” that contains a 1 bit in position 4. If the position of the bit is stored in a variable (j, say), then we can use a shift operator to create the mask:

```
i |= 1 << j;
```

- *Clearing a bit.* The following statement will clear bit 4 of i:

```
i &= 0xffef;
```

Here the mask contains a 0 bit in the position that we want to clear. We could have used the following statement instead:

```
i &= ~ 0x0010;
```

The latter form is particularly useful if the position of the bit is stored in a variable. For example, we can use the following statement to clear bit j:

```
i &= ~ (1 << j);
```

- *Testing a bit.* The following if statement tests whether bit 4 of i is set:

```
if (i & 0x0010) ...
```

To test whether bit j is set, we can use the following statement:

```
if (i & 1 << j) ...
```

Dealing with several consecutive bits (a *bit-field*) is slightly more complicated:

- *Modifying a bit-field.* The following statement will store the binary value 101 in bits 4Æ6 of i:

```
i = i & 0xff8f | 0x0050;
```

The & operator clears bits 4Æ6 of i; the | operator then sets bits 6 and 4. Notice that i |= 0x0050 would set bits 6 and 4, but wouldn’t change bit 5. Now suppose that the variable j contains the value to be stored in bits 4Æ6 of i. The following statement will update i:

```
i = i & 0xff8f | j << 4;
```

The << operator has higher precedence than & and |, so this statement is equivalent to

```
i = (i & 0xff8f) | (j << 4);
```

- *Retrieving a bit-field.* Let’s first assume that the bit-field is located at the right end of i, say in bits 0Æ2. The following statement will store this bit-field in the variable j:

```
j = i & 0x0007;
```

The mask 0x0007 contains 1 bits in each of the desired



positions. If the bit-field is somewhere in the middle of `i`, then we must first extract the field using the `&` operator, then shift the result to the right. For example, the following statement extracts bits 4Æ6 of `i`:

```
j = (i & 0x0070) >> 4;
```

## Bit-Fields in Structures

---

The techniques discussed in the previous section allow a program to modify or retrieve the value of a bit-field. These techniques are tricky to use, however, not to mention confusing for someone reading the program. Fortunately, C provides a better methodÆdeclaring structures whose members represent bit-fields.

For example, consider the layout of a file date in the MS-DOS operating system. A file date requires 16 bits, with 5 bits for the day, 4 bits for the month, and 7 bits for the year. (Years are stored relative to 1980.) Using bit-fields, we can define a C structure with an identical layout:

```
struct file_date {
    unsigned int day: 5;
    unsigned int month: 4;
    unsigned int year: 7;
};
```

The number after each member indicates its length in bits. C compilers are required to allocate exactly the number of bits requested for each member. Furthermore, compilers may not leave space between bit-fields; they must be packed together as tightly as possible. As a result, we're guaranteed that a `file_date` structure will occupy exactly 16 bits of memory (one word).

The order in which bit-fields are allocated within a word depends on the implementation. TopSpeed C allocates bit-fields from right to left (the first bit-field occupies the low-order bits of the word); here's how the `file_date` structure would look:

A bit-field can be used just like any other member of a structure, as the following example shows:

```
struct file_date fd;

fd.day = 28;
fd.month = 12;
fd.year = 8;
```

After these assignments, `fd` has the following appearance:

Bit-fields can have type `int`, `unsigned int`, or `signed int`. (`int` by itself is ambiguous when used to declare a bit-field; some compilersÆincluding TopSpeed CÆtreat the field's high-order bit as a sign bit, while others don't.) A bit-field need not be given a name; unnamed bit-fields are useful as "padding" between other bit-fields.

# APPENDIX A

## Differences between ANSI C and K&R C

This appendix lists most of the significant differences between ANSI C and original K&R C (the language described in the first edition of Kernighan and Ritchie's *The C Programming Language*). The chapter headings indicate where each ANSI feature is discussed in this tutorial.

### Chapter 3. Operators and Expressions

- K&R C doesn't provide the unary + operator.

### Chapter 4. Basic Types

- K&R C provides only one unsigned type (unsigned int).
- K&R C doesn't support the signed type specifier.
- K&R C doesn't provide the U (or u) suffix to specify that an integer constant is unsigned nor does it provide the F (or f) suffix to indicate that a floating constant is to be stored as a float value instead of a double value. In K&R C, the L (or l) suffix can't be used with floating constants.
- K&R C allows the use of long float as a synonym for double; this usage isn't legal in ANSI C.
- K&R C doesn't provide the type long double.
- The escape sequences \a, \v, and \? don't exist in K&R C. Also, K&R C doesn't provide hexadecimal escape sequences.
- K&R C doesn't support trigraph sequences and multibyte characters.
- In K&R C, all floating-point arithmetic is performed in double precision (float operands in an expression are always converted to type double). ANSI C removes this requirement.

### Chapter 5. Statements

- In K&R C, the controlling expression in a switch statement must have type int after promotion. In ANSI C, the expression may be of any integer type after promotion, including unsigned int and long int.

### Chapter 7. Functions

- In an ANSI C function *definition*, the types of the formal parameters are included in the parameter list:

**double square(double x)**

```

{
    return x * x;
}

```

K&R C requires that the types of formal parameters be specified in separate lists:

```

double square(x)
double x;
{
    return x * x;
}

```

An ANSI C function *declaration* specifies the types of the function's formal parameters (and the names as well, if desired):

```

double square(double x);
double square(double); /* alternate form */
int getchar(void);     /* no parameters */

```

(This form of declaration is known as a *function prototype*.) A K&R function declaration omits all information about parameters:

```

double square();
int getchar();

```

When a K&R-style definition or declaration is used, the compiler doesn't check that the function is called with parameters of the proper number and type. Furthermore, the actual parameters aren't automatically converted to the types of the corresponding formal parameters. Instead, the integral promotions are always performed, and float parameters are always converted to type double.

- K&R C doesn't support the void type.

## **Chapter 9. Strings**

- In K&R C, adjacent string literals aren't concatenated.
- K&R C doesn't prohibit the modification of string literals.
- In K&R C, an initializer for a string variable of length  $n$  is limited to  $n - 1$  characters (leaving room for a null character at the end). ANSI C allows the initializer to have length  $n$ .

## **Chapter 10. The Preprocessor**

- K&R C doesn't provide the `#elif`, `#error`, and `#pragma` directives.
- K&R C doesn't provide the `#`, `##`, and `defined` operators.

## **Chapter 11. Declarations**

- K&R C doesn't provide the `const` and `volatile` type qualifiers.
- K&R C doesn't allow the initialization of automatic arrays and structures nor does it allow initialization of unions (regardless of storage duration).

## **Chapter 12. Structures, Unions, and Enumerations**

- K&R C doesn't allow structures to be assigned, passed as parameters, or returned by functions.
- K&R C doesn't support enumerations.

## **Chapter 13. Advanced Uses of Pointers**

- In ANSI C, the type `void *` is used as a “generic” pointer type; for example, `malloc` returns a value of type `void *`. In K&R C, the type `char *` is used for this purpose.
- K&R C allows pointers of different types to be mixed in assignments and comparisons. In ANSI C, pointers of type `void *` can be mixed with pointers of other types, but any other mixing isn't allowed without casting. Similarly, K&R C allows mixing of integers and pointers in assignments and comparisons; ANSI C requires the use of casting.
- In K&R C, the `sizeof` operator returns a value of type `int`; in ANSI C, it returns a value of type `size_t` (an integer type that may vary from one implementation to another).
- If `f` is a pointer to a function, ANSI C allows the use of either `(*f)(Ö)` or `f(Ö)` to call the function pointed to by `f`. K&R C allows only `(*f)(Ö)`.

# APPENDIX B

## The Standard Library

The ANSI C standard library is divided into fifteen parts, with each part described by a header. Most headers contain function declarations, type definitions, and/or macro definitions. If a source file contains a call of a function declared in a header or uses one of the types or macros defined there, the header should be included at the beginning of the source file.

The following rules govern the use of headers:

- When a file includes several headers, the order of `#include` directives doesn't matter.
- The names of functions, types, and macros defined in a header are reserved; files that include the header shouldn't use these names for any other purpose. Identifiers with external linkage (function names in particular) are *always* reserved, even if the header isn't included. To avoid conflicts, it's a good idea to become familiar with the names used in the C library.
- Some headers may define macros with the same names as library functions. (Declarations of the functions are also present.) By default, calls of these functions will be treated as macro invocations (since macro names are replaced during preprocessing). The macro definition can be removed (allowing access to the true function) by using the `#undef` directive. For example, the following lines allow access to the function `islower` (instead of the macro `islower`):

```
#include <ctype.h>  
#undef islower
```

The remainder of this appendix lists the fifteen headers in the standard library. Each header is followed by a brief description and a few examples of the declarations and definitions that appear in the header. (Ellipses indicate the omission of implementation-defined details.) This appendix is intended to serve as a “road map” to help you determine which part of the library you need. For a complete description of the library, see the *Library Reference*.

### <assert.h> Diagnostics

Contains only the `assert` macro, which enables programs to perform self-checks. If any check fails, the program terminates.

```
#include <assert.h>

void assert(int expression);
/* if expression is false, writes a */
/* diagnostic message and terminates */
/* execution */
```

### **<ctype.h> Character Handling**

Provides functions for testing and converting characters.

```
#include <ctype.h>

int islower(int c);
/* is c a lower-case letter? */
int tolower(int c);
/* if c is an upper-case letter, returns */
/* the lower- case version; otherwise, */
/* returns c unchanged */
```

### **<errno.h> Errors**

Provides macros that allow the program to detect whether an error has occurred during calls of certain library functions. `errno` (which may be a macro or a variable) is an lvalue of type `int` that is set to a positive value when an error occurs.

```
#include <errno.h>

#define errno 0
```

### **<float.h> Characteristics of Floating Types**

Provides macros that define the range and accuracy of floating types.

```
#include <float.h>

#define FLT_DIG 0 /* digits of precision */
#define FLT_MAX 0 /* largest number of type float */
```

### **<limits.h> Sizes of Integral Types**

Provides macros that define the range of integer and character types.

```
#include <limits.h>

#define CHAR_BIT 0 /* number of bits per character */
#define INT_MAX 0 /* largest number of type int */
```

### **<locale.h> Localization**

Provides functions to control locale-specific aspects of the library (aspects that depend on the country or other geographic location). These include the format of numerical values (for example, the character used as the decimal point), the format of monetary values (for example, the currency symbol), the character set, and the appearance of the date and time.

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

### **<math.h> Mathematics**

Provides common mathematical functions, including trigonometric, hyperbolic, exponential, and logarithmic functions. Most functions in <math.h> expect parameters of type double and return values of type double.

```
#include <math.h>

double sin(double x); /* sine of x */
double pow(double x, double y); /* x raised to the power y */
```

### **<setjmp.h> Nonlocal Jumps**

Provides the setjmp macro and the longjmp function. setjmp “marks” a place in a program; longjmp can then be used to return to that place later (for example, when an error occurs).

```
#include <setjmp.h>

typedef ̈ jmp_buf;

int setjmp(jmp_buf env); /* saves the current environment in env and
returns 0 */

void longjmp(jmp_buf env, int val); /* restores the environment stored in
env, then causes setjmp to return val */
```

### **<signal.h> Signal Handling**

Provides facilities for handling exceptional conditions, including interrupts and runtime errors.

```
#include <signal.h>

void (*signal(
    int sig,
    void (*func)(int)))(int);
    /* installs a signal handler (a function */
    /* to be called when a signal occurs) */

int raise(int sig); /* raises a signal */
```

### **<stdarg.h> Variable Arguments**

Allows the programmer to write functions like printf and scanf that have variable-length parameter lists. Contains the type va\_list and the macros va\_start, va\_arg, and va\_end.

```
#include <stdarg.h>

typedef ̈ va_list;
```

### **<stddef.h> Common Definitions**

Provides definitions of frequently used types and macros.

```
#include <stddef.h>

typedef unsigned size_t; /* unsigned integer type returned by sizeof */
#define NULL 0 /* null pointer */
```

### **<stdio.h> Input/Output**

Provides functions for text and binary I/O and file operations.

```
#include <stdio.h>

int printf(const char *format, ...);
/* prints a list of data items */
int scanf(const char *format, ...);
/* reads a list of data items */
int getchar(void);
/* reads a character */
int putchar(int c);
/* prints a character */
```

### **<stdlib.h> General Utilities**

Provides functions for converting numbers to strings and vice-versa, random number generation, memory management, communication with the environment, searching and sorting, integer arithmetic, and performing operations on multibyte characters and strings.

```
#include <stdlib.h>

int atoi(const char *nptr);
/* converts string to integer */
void *malloc(size_t size);
/* allocates size bytes */
void exit(int status);
/* causes program to terminate with */
/* the specified status */
```

### **<string.h> String Handling**

Provides functions that perform string operations.

```
#include <string.h>

char *strcpy(char *s1, const char *s2);
/* copies s2 to s1 */
char *strcat(char *s1, const char *s2);
/* concatenates s2 to end of s1 */
```

### **<time.h> Date and Time**

Provides functions for manipulating the date and time.

```
#include <time.h>
typedef unsigned clock_t;
/* used to declare time variables; time is
/* measured in "clock ticks" */
#define CLOCKS_PER_SEC 0 /* number of "clock ticks" per second */
clock_t clock(void);
/* returns processor time used since
/* execution began, measured in
/* "clock ticks" */
```



## Bibliography

*Draft Proposed American National Standard for Information Systems—Programming Language C*, Document Number X3J11/88Æ158, December 7, 1988.

*Rationale for Draft Proposed American National Standard for Information Systems—Programming Language C*, Document Number X3J11/88Æ151, November 14, 1988. Explains the reasons for various decisions made during the creation of the ANSI standard.

Feuer, A. R., *The C Puzzle Book*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1989. Contains numerous “puzzles” (small ANSI C programs for which the reader must predict the output). Good for testing one’s C knowledge and reviewing the fine points of the language. (In the first edition, published in 1982, the puzzles are written in original K&R C.)

Harbison, S. P., and G. L. Steele, Jr., *C: A Reference Manual*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1987. The ultimate C reference—essential reading for the would-be C expert. Covers both ANSI C and K&R C in considerable detail, with frequent discussions of implementation differences found in C compilers. Not a tutorial—assumes that the reader is already familiar with C.

Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1988. The original C book. Includes both a tutorial and a complete C reference manual. Reflects the changes made in ANSI C. (The first edition, published in 1978, is still in print but will soon be obsolete as ANSI C replaces original K&R C.)

Koenig, A., *C Traps and Pitfalls*, Addison-Wesley, Reading, Mass., 1989. An excellent compendium of common (and some not-so-common) C pitfalls. Forewarned is forearmed.

Tondo, C. L., and S. E. Gimpel, *The C Answer Book*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1988. Contains answers to the exercises in the second edition of K&R. (The first edition of *The C Answer Book*, published in 1985, provides answers to the exercises in the first edition of K&R.)

## Index

---

### Symbols

#define directive 130, 145, 147  
 #define symbol 17  
 #elif directive 121  
 #else directive 121  
 #include directive 123  
 % symbol 19  
 %d operator 15  
 %d symbol 18  
 & operator 174  
 & symbol 18, 23  
 \* symbol 125  
 \*/ symbol 13  
 /\* symbol 13  
 < symbol 164  
 << operator 173  
 >> operator 173  
 >symbol 164  
 ^ operator 174  
 { symbol 24, 26, 132  
 | operator 174  
 } symbol 24, 26, 132  
 ~ operator 174

### A

accessing a structure 137  
 accessing bit-fields 175  
 accessing bits 175  
 ANSI C 16, 137, 150, 159, 163  
 arrays  
   dynamic allocation 151  
   names 125  
   of structures 138  
 assigning values to variables 14  
 auxiliary functions 143

### B

Bell Laboratories 8  
 binary input/output 170  
 binary streams 166  
 bit-fields in structures 177  
 bitfields 173  
 bitwise AND operation 174  
 bitwise complement operation 174  
 bitwise operators 173  
 bitwise OR operation 174  
 bitwise shift operators 173

blank lines 26  
 blocks 132  
 boolean operations 173  
 boolean types 145  
 braces  
   use of 24, 146

### C

C  
   advantages 8  
   case sensitivity 16  
   disadvantages 9  
   fundamentals 12  
   layout restrictions 25  
   preprocessor 17  
   program layout 25  
   tips 10  
 calloc function 151  
 calls  
   definition 13  
 case sensitivity 16  
 character 13  
 clearing bits 176  
 closing files 165  
 code libraries 10  
 coding conventions 10  
 command line arguments 165  
 commas  
   use of 125  
 comments 13  
 compiling a program 123  
 compound statements 132  
 compound statements:definition 24  
 const qualifier 130  
 constants  
   defining 17  
   definition 17  
 constants naming 17  
 conventions 123  
   use of 16, 17  
 conversion specifiers:most common 20  
 conversion specifications:form 20  
 conversion specifications:minimum field width 20  
 conversion specifiers:definition 19  
 conversion specifiers:usage 19, 22  
 conversion specifications:precision 20

### D

d specifier 20  
 data items 23

- debugging 10
- declarations
  - declaration specifiers 125
  - purpose 125
  - use 125
- declarators 125
  - complex 131
  - initializers 132
  - restrictions on 131
  - simple 131
  - summary of 131
- declaring structures 135
- declaring variables 14, 125
- defining constants 17
- defining macros 17
- defining variables 125
- definitions
  - purpose 125
- deleting nodes 154
- detecting EOF 169
- detecting errors 169
- directives 17
- dividing symbols 26
- dynamic array allocation 151
- dynamic storage allocation 149
- dynamic string allocation 150

## E

- e specifier 20
- enumeration constants 146
- enumerations
  - defining 146
  - properties of 146
  - uses 146
- EOF macro 165, 167, 168, 169
- errors
  - detecting 169
- explanatory remarks:use of 13
- expressions
  - definition 14

## F

- f specifier 20
- feof function 169
- ferror function 169
- fgetc function 168
- fgets function 168
- file pointers 163
- files
  - closing 165

- opening 164
- float type 18
- fopen function 164
- format string 22
- format strings 18, 19
- fprintf function 167
- fputc function 167
- fputs function 167
- fread function 170
- free function 150
- fscanf function 168
- fseek function 171
- ftell function 171
- function free 150
- functions 168
  - auxiliary 143
  - calloc 151
  - fgets 168
  - fopen 164
  - fprintf 167
  - fputc 167
  - fputs 167
  - fread 170
  - fscanf 168
  - fwrite 170
  - getc 168
  - getchar 168
  - gets 168
  - malloc 149
  - names 125
  - pointers to 159
  - printf 13, 19, 167
  - putc 167
  - putchar 167
  - puts 167
  - qsort 160
  - scanf 18, 168
  - storage classes 129
- functionios
  - scanf 22
- fwrite function 170

## G

- g specifier 20
- getc function 168
- getchar function 168
- gets function 168

## H

- header files 144

- conventions for 123
- names of 123

## I

- identifiers
  - content 15
  - definition 15
  - meanings of 133
  - rules for 15
- indentation
  - use of 26
- initializers 132
- input characters 22
- input functions 168
- input/output
  - text 166
- inserting nodes 153
- int type 14
- integers 14, 23, 146

## K

- keywords
  - definition 16
  - list of 16

## L

- layout of a C program 25
- layout rules 26
- legal identifiers 15
- linked list 152
- linking a program 123
- loops 25
- low-level programming 173
- lvalues 137

## M

- machine level programming 173
- macros
  - defining 17
- malloc function 149
- minimum field width 20
- modifying bit-fields 176

## N

- naming constants 17
- nesting 26
- newline character 13, 166
- newlinw character 168

- nodes 152
- null character 98, 100, 151, 168
- null pointer 168
- numbers
  - integers 14

## O

- opening files 164
- operands 14
- operation on structures 137
- operators 14
  - %d 15
  - & 174
  - << 173
  - ^ 174
  - | 174
  - ~ 174
  - bitwise shift 173
  - precedence 174
  - unary 174
- output functions 167

## P

- parameters
  - pointers as 160
  - to programs 165
- parentheses
  - use of 17, 25, 125
- parentheses
  - use of 131
- pointers
  - advanced uses 149
  - as parameters 160
  - names 125
  - to files 163
  - to functions 159
- precision 20
- printf function 13, 19, 47, 167
- program compilation 123
- program layout 25
- program linking 123
- program parameters 165
- programs at low-level 173
- putc function 167
- putchar function 167
- puts function 167

## Q

- qsort function 160
- qualifiers

const 130  
volatile 130

## R

random access 171  
read-only memory 130  
records 135  
redirection 163  
retrieving bit-fields 176  
rewind function 171  
ROM 130

## S

scanf function 18, 22, 47, 168  
scope rules 133  
searching linked lists 153  
semicolon  
    use of 13, 24, 136  
setting bits 175  
space characters 22, 25  
specifiers  
    d 20  
    e 20  
    f 20  
    g 20  
splitting strings 26  
square brackets  
    use of 125  
standard streams 163  
statements  
    compound 24, 132  
    sequential 132  
    while 25  
stddef.h 144  
stderr 163  
stdin 163  
stdout 163  
storage allocation 149  
storage classes  
    scope 126  
storage class 125  
storage classe  
    of variables 126  
storage classes  
    duration 126  
    inkages 126  
    use 126  
storage classes of functions 129  
streams 163  
strings

definition 13  
dynamic allocation 150  
format string 22  
format strings 18, 19  
iterals 165  
splitting 26  
structures  
    accessing 137  
    in arrays 138  
    operations on 137  
symbol  
    % 19  
    } 24  
symbols  
    #define 17  
    %d 18  
    & 18, 23  
    \* 125  
    < 164  
    > 164  
    { 24, 26, 132  
    } 26, 132  
dividing 26

## T

tag fields  
    adding to unions 148  
temporary variables 133  
testing bits 176  
text input functions 168  
text input/output 166  
text output 167  
text streams 166  
type qualifiers 125  
type specifiers 125  
types  
    components 135  
    declaring 135  
    defining 135  
    float 18  
    int 14  
    records 135  
    structures 135

## U

unary operators 174  
underscore character:use of 16  
unions  
    adding tag fields 148  
    explanation 145

- properties of 145
- purpose 145
- UNIX 8, 9

## V

variables

- assignment 14
- declarations 125
- declaring 14
- definition 14
- definitions 125
- names 125
- storage classes 126
- temporary 133
- with external linkages 128

volatile qualifier 130

volume = height \* length \* width; 14

## W

while statement 25

white space characters 22

white space characterss 25