9

File I/O

chapter objectives

9.1 Understand streams

9.2 Master file-system basics

9.3 Understand **feof( )** and **ferror( )**

9.4 Learn some higher-level text functions

9.5 Learn to read and write binary data

9.6 Understand random access

9.7 Learn about various file-system functions

9.8 Learn about the standard streams

ALTHOUGH C does not define any keywords that perform file I/O, the C standard library contains a very rich set of I/O functions. As you will see in this chapter, C's approach to I/O is efficient, powerful, and flexible.

Note

Most C compilers supply two complete sets of file I/O functions. One is called the ANSI file system (sometimes called the buffered file system). This file system is defined by the ANSI C standard. The second file system is based on the original UNIX operating environment and is called the UNIX-like file system (sometimes called the unbuffered file system). This file system is not defined by the ANSI C standard. The ANSI standard only defines one file system because the two file systems are redundant. Further, not all environments may be able to adapt to the UNIX-like system. For these reasons, this book only discusses the ANSI file system. For a discussion of the UNIX-like file system, see my book C: The Complete Reference *(Berkeley, CA, Osborne/McGraw-Hill).*

Review Skills Check

Before proceeding you should be able to perform these exercises and answer these questions:

1. What is the difference between **getchar( )** and **getche( )**?
2. Give one reason why you probably won't use **scanf( )**'s **%s** option to read strings from the keyboard.
3. Write a program that prints a four-column table of the prime numbers between 2 and 1000. Make sure that the columns are aligned.
4. Write a program that inputs a **double**, a character, and a string not longer than 20 characters. Redisplay the values to confirm that they were input correctly.
5. Write a program that reads and discards leading digits and then reads a string. (Hint: Use a scanset to read past any leading digits.)

9.1 UNDERSTAND STREAMS

Before we can begin our discussion of file I/O, you must understand two very important concepts: the *stream* and the *file*. The C I/O system supplies a consistent interface to the programmer, independent of the actual I/O device being used. To accomplish this, C provides a level of abstraction between the programmer and the hardware. This abstraction is called a stream. The actual device providing I/O is called a file. Thus, a stream is a logical interface to a file. As C defines the term file, it can refer to a disk file, the screen, the keyboard, memory, a port, a file on tape, and various other types of I/O devices. The most common form of file is, of course, the disk file. Although files differ in form and capabilities, all streams are the same. The advantage to this approach is that to you, the programmer, one hardware device will look much like any other. The stream automatically handles the differences.

A stream is linked to a file using an *open operation*. A stream is disassociated from a file using a *close operation*.

There are two types of streams: text and binary. A *text stream* contains ASCII characters. When a text stream is being used, some character translations may take place. For example, when the newline character is output, it is usually converted into a carriage return, linefeed pair. For this reason, there may not be a one-to-one correspondence between what is sent to the stream and what is written to the file. A *binary stream* may be used with any type of data. No character translations will occur, and there is a one-to-one correspondence between what is sent to the stream and what is actually contained in the file.

One final concept you need to understand is that of the *current location*. The current location, also referred to as the *current position*, is the location in a file where the next file access will occur. For example, if a file is 100 bytes long and half the file has been read, the next read operation will occur at byte 50, which is the current location.

To summarize: In C, disk I/O (like certain other types of I/O) is performed through a logical interface called a stream. All streams have similar properties, and all are operated on by the same I/O functions, no matter what type of file the stream is associated with. A file is the actual physical entity that receives or supplies the data. Even though files differ, streams do not. (Of course, some devices may not support random-access operations, for example, so their associated streams will not support such operations either.)

Now that you are familiar with the theory behind C's file system, it is time to begin learning about it in practice.

9.2 MASTER FILE-SYSTEM BASICS

In this section you will learn how to open and close a file. You will also learn how to read characters from and write characters to a file.

To open a file and associate it with a stream, use **fopen( )**. Its prototype is shown here:

FILE *fopen(char **fname*, char **mode*);

The **fopen( )** function, like all the file-system functions, uses the header STDIO.H. The name of the file to open is pointed to by *fname*. It must be a valid file name, as defined by the operating system. The string pointed to by *mode* determines how the file may be accessed. The legal values for *mode* as defined by the ANSI C standard are shown in Table 9-1. Your compiler may allow additional modes.

If the open operation is successful, **fopen( )** returns a valid file pointer. The type **FILE** is defined in STDIO.H. It is a structure that holds various kinds of information about the file, such as its size, the current location of the file, and its access modes. It essentially identifies the file. (A structure is a group of variables accessed under one name. You will learn about structures in the next chapter, but you do not need to know anything about them to learn and fully use C's file system.) The **fopen( )** function returns a pointer to the structure associated with the file by the open process. You will use this pointer with all other functions that operate on the file. However, you must never alter it or the object it points to.

If the **fopen( )** function fails, it returns a null pointer. The header STDIO.H defines the macro **NULL**, which is defined to be a null pointer. It is very important to ensure that a valid file pointer has been returned. To do so, check the value returned by **fopen( )** to make sure that it is not **NULL**. For example, the proper way to open a file called **myfile** for text input is shown in this fragment:

```
FILE *fp;

if((fp = fopen("myfile", "r")) == NULL) {
  printf("Error opening file.\n");
  exit(1); /* or substitute your own error handler */
}
```

Although most of the file modes are self-explanatory, a few comments are in order. If, when opening a file for read-only operations, the file does not exist, **fopen( )** will fail. When opening a file using append mode, if the file does not exist, it will be created. Further, when a file is opened for append all new data written to the file will be written to the end of the file. The original contents will remain unchanged. If, when a file is opened for writing, the file does not exist, it will be created. If it does exist, the contents of the original file will be destroyed and a new file created. The difference between modes **r+** and **w+** is that **r+** will not create a file if it does not exist; however, **w+** will. Further, if the file already exists, opening it with **w+** destroys its contents; opening it with **r+** does not.

Mode	Meaning
"r"	Open a text file for reading.
"w"	Create a text file for writing.
"a"	Append to a text file.
"rb"	Open a binary file for reading.
"wb"	Create a binary file for writing.
"ab"	Append to a binary file.
"r+"	Open a text file for read/write
"w+"	Create a text file for read/write
"a+"	Append or create a text file for read/write.
"r+b"	Open a binary file for read/write. You may also use "rb+".
"w+b"	Create a binary file for read/write. You may also use "wb+".
"a+b"	Append or create a binary file for read/write. You may also use "ab+".

TABLE 9-1 *The Legal Values for Mode*

To close a file, use **fclose( )**, whose prototype is

```
int fclose(FILE *fp);
```

The **fclose( )** function closes the file associated with *fp*, which must be a valid file pointer previously obtained using **fopen( )**, and disassociates the stream from the file. In order to improve efficiency, most file system implementations write data to disk one sector at a time. Therefore, data is buffered until a sector's worth of information has been output before the buffer is physically written to disk. When you call **fclose( )**, it automatically writes any information remaining in a partially full buffer to disk. This is often referred to as *flushing the buffer*.

You must never call **fclose( )** with an invalid argument. Doing so will damage the file system and possibly cause irretrievable data loss.

The **fclose( )** function returns zero if successful. If an error occurs, **EOF** is returned.

Once a file has been opened, depending upon its mode, you may read and/or write bytes (i.e., characters) using these two functions:

```
int fgetc(FILE *fp);

int fputc(int ch, FILE *fp);
```

The **fgetc( )** function reads the next byte from the file described by *fp* as an **unsigned char** and returns it as an integer. (The character is returned in the low-order byte.) If an error occurs, **fgetc( )** returns **EOF**. As you should recall from Chapter 8, **EOF** is a negative integer (usually −1). The **fgetc( )** function also returns **EOF** when the end of the file is reached. Although **fgetc( )** returns an integer value, your program can assign it to a **char** variable since the low-order byte contains the character read from the file.

The **fputc( )** function writes the byte contained in the low-order byte of *ch* to the file associated with *fp* as an **unsigned char**. Although *ch* is defined as an **int**, you may call it using a **char**, which is the common procedure. The **fputc( )** function returns the character written if successful or **EOF** if an error occurs.

Historical note: The traditional names for **fgetc( )** and **fputc( )** are **getc( )** and **putc( )**. The ANSI C standard still defines these names, and they are essentially interchangeable with **fgetc( )** and **fputc( )**. One reason the new names were added was for consistency. All other ANSI file system function names begin with 'f,' so 'f' was added to

getc( ) and **putc( )**. The ANSI standard still supports the traditional names, however, because there are so many existing programs that use them. If you see programs that use **getc( )** and **putc( )**, don't worry. They are essentially different names for **fgetc( )** and **fputc( )**.

EXAMPLES

1. This program demonstrates the four file-system functions you have learned about so far. First, it opens a file called MYFILE for output. Next, it writes the string "This is a file system test." to the file. Then, it closes the file and reopens it for read operations. Finally, it displays the contents of the file on the screen and closes the file.

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

int main(void)
{
  char str[80] = "This is a file system test.\n";
  FILE *fp;
  char *p;
  int i;

  /* open myfile for output */
  if((fp = fopen("myfile", "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* write str to disk */
  p = str;
  while(*p) {
    if(fputc(*p, fp)==EOF) {
      printf("Error writing file.\n");
      exit(1);
    }
    p++;
  }
  fclose(fp);
  /* open myfile for input */
  if((fp = fopen("myfile", "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* read back the file */
  for(;;) {
    i = fgetc(fp);
    if(i == EOF) break;
    putchar(i);
  }
  fclose(fp);

  return 0;
}
```

In this version, when reading from the file, the return value of **fgetc( )** is assigned to an integer variable called **i**. The value of this integer is then checked to see if the end of the file has been reached. For most compilers, however, you can simply assign the value returned by **fgetc( )** to a **char** and still check for **EOF**, as is shown in the following version:

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

int main(void)
{
  char str[80] = "This is a file system test.\n";
  FILE *fp;
  char ch, *p;

  /* open myfile for output */
  if((fp = fopen("myfile", "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* write str to disk */
  p = str;
  while(*p) {
    if(fputc(*p, fp)==EOF) {
      printf("Error writing file.\n");
      exit(1);
    }
    p++ ;
  }
  fclose(fp);

  /* open myfile for input */
  if((fp = fopen("myfile", "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* read back the file */
  for(;;) {
    ch = fgetc(fp);
    if(ch == EOF) break;
    putchar(ch);
  }
  fclose(fp);

  return 0;
}
```

Sourav

The reason this approach works is that when a **char** is being compared to an **int**, the **char** value is automatically elevated to an equivalent **int** value.

There is, however, an even better way to code this program. For example, there is no need for a separate comparison step because the assignment and the comparison can be performed at the same time, within the **if**, as shown here:

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

int main(void)
{
  char str[80] = "This is a file system test.\n";
  FILE *fp;
  char ch, *p;

  /* open myfile for output */
  if((fp = fopen("myfile", "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }
  /* write str to disk */
  p = str;
  while(*p) {
    if(fputc(*p, fp)==EOF) {
      printf("Error writing file.\n");
      exit(1);
    }
    p++ ;
  }
  fclose(fp);

  /* open myfile for input */
  if((fp = fopen("myfile", "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* read back the file */
  for(;;) {
    if((ch = fgetc(fp)) == EOF) break;
    putchar(ch);
  }
  fclose(fp);

  return 0;
}
```

Don't let the statement

```
if((ch = fgetc(fp)) == EOF) break;
```

fool you. Here's what is happening. First, inside the **if**, the return value of **fgetc( )** is assigned to **ch**. As you may recall, the assignment operation in C is an expression. The entire value of **(ch = fgetc(fp))** is equal to the return value of **fgetc( )**. Therefore, it is this integer value that is tested against **EOF**.

Expanding upon this approach, you will normally see this program written by a professional C programmer as follows:

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

int main(void)
{
  char str[80] = "This is a file system test.\n";
  FILE *fp;
  char ch, *p;

  /* open myfile for output */
  if((fp = fopen("myfile", "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* write str to disk */
  p = str;
  while(*p)
    if(fputc(*p++, fp)==EOF) {
      printf("Error writing file.\n");
      exit(1);
    }

  fclose(fp);

  /* open myfile for input */
  if((fp = fopen("myfile", "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* read back the file */
  while((ch = fgetc(fp)) != EOF) putchar(ch);
  fclose(fp);

  return 0;
}
```

Notice that now, each character is read, assigned to **ch**, and tested against **EOF**, all within the expression of the **while** loop that controls the input process. If you compare this with the original version, you can see how much more efficient this one is. In fact, the ability to integrate such operations is one reason C is so powerful. It is important that you get used to the kind of approach just shown. Later on in this book we will explore such assignment statements more fully.

2. The following program takes two command-line arguments. The first is the name of a file, the second is a character. The program searches the specified file, looking for the character. If the file

contains at least one of these characters, it reports this fact. Notice how it uses **argv** to access the file name and the character for which to search.

```
/* Search specified file for specified character. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  /* see if correct number of command line arguments */
  if(argc!=3) {
    printf("Usage: find <file name> <ch>\n");
    exit(1);
  }

  /* open file for input */
  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* look for character */
  while((ch = fgetc(fp)) != EOF)
    if(ch==*argv[2]) {
      printf("%c found", ch);
      break;
    }
  fclose(fp);

  return 0;
}
```

EXERCISES

1. Write a program that displays the contents of the text file specified on the command line.

2. Write a program that reads a text file and counts how many times each letter from 'A' to 'Z' occurs. Have it display the results. (Do not differentiate between upper- and lowercase letters.)
3. Write a program that copies the contents of one text file to another. Have the program accept three command-line arguments. The first is the name of the source file, the second is the name of the destination file, the third is optional. If present and if it equals "watch," have the program display each character as it copies the files; otherwise, do not have the program display any screen output. If the destination file does not exist, create it.

9.3 UNDERSTAND feof( ) AND ferror( )

As you know, when **fgetc( )** returns **EOF**, either an error has occurred or the end of the file has been reached, but how do you know which event has taken place? Further if you are operating on a binary file, all values are valid. This means it is possible that a byte will have the same value (when elevated to an **int**) as **EOF**, so how do you know if valid data has been returned or if the end of the file has been reached? The solution to these problems are the functions **feof( )** and **ferror( )**, whose prototypes are shown here:

```
int feof(FILE *fp);

int ferror(FILE *fp);
```

The **feof( )** function returns nonzero if the file associated with *fp* has reached the end of the file. Otherwise it returns zero. This function works for both binary files and text files. The **ferror( )** function returns nonzero if the file associated with *fp* has experienced an error; otherwise, it returns zero.

Using the **feof( )** function, this code fragment shows how to read to the end of a file:

```
FILE *fp;
.
.
.
while(!feof(fp)) ch = fgetc(fp);
```

This code works for any type of file and is better in general than checking for **EOF**. However, it still does not provide any error checking. Error checking is added here:

```
FILE *fp;
.
.
.
while(!feof(fp)) {
  ch = fgetc(fp);
  if(ferror(fp)) {
    printf("File Error\n");
    break;
  }
}
```

Keep in mind that **ferror( )** only reports the status of the file system relative to the last file access. Therefore, to provide the fullest error checking, you must call it after each file operation.

The most damaging file errors occur at the operating-system level. Frequently, it is the operating system that intercepts these errors and displays its own error messages. For example, if a bad sector is found on the disk, most operating systems will, themselves, stop the execution of the program and report the error. Often the only types of errors that actually get passed back to your program are those caused by mistakes on your part, such as accessing a file in a way inconsistent with the mode used to open it or when you cause an out-of-range condition. Usually these types of errors can be trapped by checking the return type of the other file system functions rather than by calling **ferror( )**. For this reason, you will frequently see examples of C code in which there are relatively few (if any) calls to **ferror( )**. One last point: Not all of the file system examples in this book will provide full error checking, mostly in the interest of keeping the programs short and easy to understand. However, if you are writing programs for actual use, you should pay special attention to error checking.

EXAMPLES

1. This program copies any type of file, binary or text. It takes two command-line arguments. The first is the name of the source file, the second is the name of the destination file. If the destination file does not exist, it is created. It includes full error checking. (You might want to compare this version with the copy program you wrote for text files in the preceding section.)

```
/* Copy a file. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *from, *to;
  char ch;

  /* see if correct number of command line arguments */
  if(argc!=3) {
    printf("Usage: copy <source> <destination>\n");
    exit(1);
  }

  /* open source file */
  if((from = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  }

  /* open destination file */
  if((to = fopen(argv[2], "wb"))==NULL) {
    printf("Cannot open destination file.\n");
    exit(1);
  }

  /* copy the file */
```

```
    while(!feof(from)) {
      ch = fgetc(from);
      if(ferror(from)) {
        printf("Error reading source file.\n");
        exit(1);
      }
      if(!feof(from)) fputc(ch, to);
      if(ferror(to)) {
        printf("Error writing destination file.\n");
        exit(1);
      }
    }

    if(fclose(from)==EOF) {
      printf("Error closing source file.\n");
      exit(1);
    }

    if(fclose(to)==EOF) {
      printf("Error closing destination file.\n");
      exit(1);
    }

    return 0;
  }
```

2. This program compares the two files whose names are specified on the command line. It either prints **Files are the same**, or it displays the byte of the first mismatch. It also uses full error checking.

```
  /* Compare files. */
  #include <stdio.h>
  #include <stdlib.h>

  int main(int argc, char *argv[])
  {
    FILE *fp1, *fp2;
    char ch1, ch2, same;
    unsigned long l;

    /* see if correct number of command line arguments */
    if(argc!=3) {
      printf("Usage: compare <file 1> <file 2>\n");
      exit(1);
```

```
/* open first file */
if((fp1 = fopen(argv[1], "rb"))==NULL) {
  printf("Cannot open first file.\n");
  exit(1);
}

/* open second file */
if((fp2 = fopen(argv [2], "rb"))==NULL) {
  printf("Cannot open second file.\n");
  exit(1);
}

l = 0;
same = 1;
/* compare the files */
while(!feof(fp1)) {
  ch1 = fgetc(fp1);
  if(ferror(fp1)) {
    printf("Error reading first file.\n");
    exit(1);
  }
  ch2 = fgetc(fp2);
  if(ferror(fp2)) {
    printf("Error reading second file.\n");
    exit(1);
  }
  if(ch1!=ch2) {
    printf("Files differ at byte number %lu", l);
    same = 0;
    break;
  }
  l++;
}
if(same) printf("Files are the same.\n");

if(fclose(fp1)==EOF) {
  printf("Error closing first file.\n");
  exit(1);
}

if(fclose(fp2)==EOF) {
  printf("Error closing second file.\n");
  exit(1);
```

```
    }

    return 0;
}
```

EXERCISES

1. Write a program that counts the number of bytes in a file (text or binary) and displays the result. Have the user specify the file to count on the command line.
2. Write a program that exchanges the contents of the two files whose names are specified on the command line. That is, given two files called FILE1 and FILE2, after the program has run, FILE1 will contain the contents that originally were in FILE2, and FILE2 will contain FILE1's original contents. (Hint: Use a temporary file to aid in the exchange process.)

9.4 LEARN SOME HIGHER-LEVEL TEXT FUNCTIONS

When working with text files, C provides four functions that make file operations easier. The first two are called **fputs( )** and **fgets( )**, which write a string to and read a string from a file, respectively. Their prototypes are

int fputs(char **str*, FILE **fp*);

char *fgets(char **str*, int *num*, FILE **fp*);

The **fputs( )** function writes the string pointed to by *str* to the file associated with *fp*. It returns **EOF** if an error occurs and a non-negative value if successful. The null that terminates *str* is not written. Also, unlike its related function **puts( )** it does not automatically append a carriage return, linefeed pair.

The **fgets( )** function reads characters from the file associated with *fp* into the string pointed to by *str* until *num*–1 characters have been read, a newline character is encountered, or the end of the file is reached. In any case, the string is null-terminated. Unlike its related function **gets( )**, the newline character is retained. The function returns *str* if successful and a null pointer if an error occurs.

The C file system contains two very powerful functions similar to two you already know. They are **fprintf( )** and **fscanf( )**. These functions operate exactly like **printf( )** and **scanf( )** except that they work with files. Their prototypes are:

int fprintf(FILE **fp*, char **control-string*, ...);

int fscanf(FILE **fp*, char **control-string*, ...);

Instead of directing their I/O operations to the console, these functions operate on the file specified by *fp*. Otherwise their operations are the same as their console-based relatives. The advantage to **fprintf( )** and **fscanf( )** is that they make it very easy to write a wide variety of data to a file using a text format.

EXAMPLES

1. This program demonstrates **fputs( )** and **fgets( )**. It reads lines entered by the user and writes them to the file specified on the command line. When the user enters a blank line, the input phase terminates, and the file is closed. Next, the file is reopened for input, and the program uses **fgets( )** to display the contents of the file.

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

int main(int argc, char *argv[])
{
  FILE *fp;
  char str[80];

  /* check for command line arg */
  if(argc!=2) {
    printf("Specify file name.\n");
    exit(1);
  }

  /* open file for output */
  if((fp = fopen(argv[1], "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  printf("Enter a blank line to stop.\n");
  do {
    printf(": ");
    gets(str);
    strcat(str, "\n"); /* add newline */
    if(*str != '\n') fputs(str, fp);
  } while(*str != '\n');
  fclose(fp);

  /* open file for input */
  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* read back the file */
  do {
    fgets(str, 79, fp);
    if(!feof(fp)) printf(str);
  } while(!feof(fp));
  fclose(fp);

  return 0;
}
```

2. This program demonstrates **fprintf( )** and **fscanf( )**. It first writes a **double**, an **int**, and a string to the file specified on the command line. Next, it reads them back and displays their values as verification. If you examine the file created by this program, you will see that it contains human-readable text. This

is because **fprintf( )** writes to a disk file what **printf( )** would write to the screen. No internal data formats are used.

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

int main(int argc, char *argv[])
{
  FILE *fp;
  double ld;
  int d;
  char str[80];

  /* check for command line arg */
  if(argc!=2) {
    printf("Specify file name.\n");
    exit(1);
  }

  /* open file for output */
  if((fp = fopen(argv[1], "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  fprintf(fp, "%f %d %s", 12345.342, 1908, "hello");
  fclose(fp);

  /* open file for input */
  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  fscanf(fp, "%lf%d%s", &ld, &d, str);
  printf("%f %d %s", ld, d, str);
  fclose(fp);

  return 0;
}
```

EXERCISES

1. In Chapter 6 you wrote a very simple telephone-directory program. Write a program that expands on this concept by allowing the directory to be saved to a disk file. Have the program present a menu that looks like this:

   ```
   1. Enter the names and numbers
   2. Find numbers
   3. Save directory to disk
   4. Load directory from disk
   5. Quit
   ```

   The program should be capable of storing 100 names and numbers. (Use only first names if you like.) Use **fprintf( )** to save the directory to disk and **fscanf( )** to read it back into memory.

2. Write a program that uses **fgets( )** to display the contents of a text file, one screenful at a time. After each screen is displayed, have the program prompt the user for more.

3. Write a program that copies a text file. Specify both the source and destination file names on the command line. Use **fgets( )** and **fputs( )** to copy the file. Include full error checking.

9.5 LEARN TO READ AND WRITE BINARY DATA

As useful and convenient as **fprintf( )** and **fscanf( )** are, they are not necessarily the most efficient way to read and write numeric data. The reason for this is that both functions perform conversions on the data. For example, when you output a number using **fprintf( )** the number is converted from its binary format into ASCII text. Conversely, when you read a number using **fscanf( )**, it must be converted back into its binary representation. For many applications, this conversion time will not be meaningful; for others, it will be a severe limitation. Further, for some types of data, a file created by **fprintf( )** will also be larger than one that contains a mirror image of the data using its binary format. For these reasons, the C file system includes two important functions: **fread( )** and **fwrite( )**. These functions can read and write any type of data, using its binary representation. Their prototypes are

```
size_t fread(void *buffer, size_t size, size_t num, FILE *fp);

size_t fwrite(void *buffer, size_t size, size_t num, FILE *fp);
```

As you can see, these prototypes introduce some unfamiliar elements. However, before discussing them, a brief description of each function is necessary.

The **fread( )** function reads from the file associated with *fp*, *num* number of objects, each object *size* bytes long, into the buffer pointed to by *buffer*. It returns the number of objects actually read. If this value is less than *num*, either the end of the file has been encountered or an error has occurred. You can use **feof( )** or **ferror( )** to find out which.

The **fwrite( )** function is the opposite of **fread( )**. It writes to the file associated with *fp*, *num* number of objects, each object *size* bytes long, from the buffer pointed to by *buffer*. It returns the number of objects written. This value will be less than *num* only if an output error has occurred.

Before looking at any examples, let's examine the new concepts introduced by the functions' prototypes.

The first concept is that of the **void** pointer. A **void** pointer is a pointer that can point to any type of data without the use of a type cast. This is generally referred to as a *generic pointer*. In C, **void** pointers are used for two primary purposes. First, as illustrated by **fread( )** and **fwrite( )**, they are a way for a function to receive a pointer to any type of data without causing a type mismatch error. As stated earlier, **fread( )** and **fwrite( )** can be used to read or write any type of data. Therefore, the functions must be capable of receiving any sort of data pointed to by *buffer*. **void** pointers make this possible. A second purpose they serve is to allow a function to return a generic pointer. You will see an example of this later in this book.

The second new item is the type **size_t**. This type is defined in the STDIO.H header file. (You will learn how to define types later in this book.) A variable of this type is defined by the ANSI C standard as being able to hold a value equal to the size of the largest object supported by the compiler. For our purposes, you can think of **size_t** as being the same as **unsigned** or **unsigned long**. The reason that **size_t** is used instead of its equivalent built-in type is to allow C compilers running in different environments to accommodate the needs and confines of those environments.

When using **fread( )** or **fwrite( )** to input or output binary data, the file must be opened for binary operations. Forgetting this can cause hard-to-find problems.

To understand the operation of **fread( )** and **fwrite( )**, let's begin with a simple example. The following program writes an integer to a file called MYFILE using its internal, binary representation and then reads it back. (The program assumes that integers are 2 bytes long.)

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

int main(void)
{
  FILE *fp;
  int i;

  /* open file for output */
  if((fp = fopen("myfile", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  i = 100;

  if(fwrite(&i, 2, 1, fp) != 1) {
    printf("Write error occurred.\n");
    exit(1);
  }
  fclose(fp);

  /* open file for input */
  if((fp = fopen("myfile", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  if(fread(&i, 2, 1, fp) != 1) {
    printf("Read error occurred.\n");
    exit(1);
  }
  printf("i is %d", i);
  fclose(fp);
  return 0;
}
```

Notice how error checking is easily performed in this program by simply comparing the number of items written or read with that requested. In some situations, however, you will still need to use **feof( ) or ferror( )** to determine if the end of the file has been reached or if an error has occurred.

One thing wrong with the preceding example is that an assumption about the size of an integer has been made and this size is hardcoded into the program. Therefore, the program will not work properly with compilers that use 4-byte integers, for example. More generally, the size of many types of data changes between systems or is difficult to determine manually. For this reason, C includes the keyword **sizeof**, which is a compile-time operator that returns the size, in bytes, of a data type or variable. It takes the general forms

sizeof(*type*)

or

sizeof *var_name*;

For example, if **floats** are four bytes long and **f** is a **float** variable, both of the following expressions evaluate to 4:

```
sizeof f
sizeof(float)
```

When using **sizeof** with a type, the type must be enclosed between parentheses. No parentheses are needed when using a variable name, although the use of parentheses in this context is not an error.

By using **sizeof**, not only do you save yourself the drudgery of computing the size of some object by hand, but you also ensure the portability of your code to new environments. An improved version of the preceding program is shown here, using **sizeof**.

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

int main(void)
{
  FILE *fp;
  int i;

  /* open file for output */
  if((fp = fopen("myfile", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  i = 100;

  if(fwrite(&i, sizeof(int), 1, fp) != 1) {
    printf("Write error occurred.\n");
    exit(1);
  }
  fclose(fp);

  /* open file for input */
  if((fp = fopen("myfile", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  if(fread(&i, sizeof i, 1, fp) != 1) {
    printf("Read error occurred.\n");
    exit(1);
  }
  printf("i is %d",i);
  fclose(fp);

  return 0;
}
```

EXAMPLES

1. This program fills a ten-element array with floating-point numbers, writes them to a file, and then reads them back. This program writes each element of the array separately. Because binary data is being written using its internal format, the file must be opened for binary I/O operations.

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

```
double d[10] = {
  10.23, 19.87, 1002.23, 12.9, 0.897,
  11.45, 75.34, 0.0, 1.01, 875.875
};

int main(void)
{
  int i;
  FILE *fp;

  if((fp = fopen("myfile", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  for(i=0; i<10; i++)
    if(fwrite(&d[i], sizeof(double), 1, fp) != 1) {
      printf("Write error.\n");
      exit(1);
    }
  fclose(fp);

  if((fp = fopen("myfile", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* clear the array */
  for(i=0; i<10; i++) d[i] = -1.0;

  for(i=0; i<10; i++)
    if(fread(&d[i], sizeof(double), 1, fp) != 1) {
      printf("Read error.\n");
      exit(1);
    }
  fclose(fp);

  /* display the array */
  for(i=0; i<10; i++) printf("%f ", d[i]);

  return 0;
}
```

The array is cleared between the write and read operations only to "prove" that it is being filled by the **fread( )** statement.

2. The following program does the same thing as the first, but here only one call to **fwrite( )** and **fread( )** is used because the entire array is written in one step, which is much more efficient. This example helps illustrate how powerful these functions are.

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

double d[10] = {
  10.23, 19.87, 1002.23, 12.9, 0.897,
  11.45, 75.34, 0.0, 1.01, 875.875
};

int main(void)
{
  int i;
  FILE *fp;

  if((fp = fopen("myfile", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* write the entire array in one step */
  if(fwrite(d, sizeof d, 1, fp) != 1) {
    printf("Write error.\n");
    exit(1);
  }
  fclose(fp);

  if((fp = fopen("myfile", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* clear the array */
  for(i=0; i<10; i++) d[i] = -1.0;

  /* read the entire array in one step */
  if(fread(d, sizeof d, 1, fp) != 1) {
    printf("Read error.\n");
```

```
      exit(1);
    }
    fclose(fp);

    /* display the array */
    for(i=0; i<10; i++) printf("%f ", d[i]);

    return 0;
  }
```

EXERCISES

1. Write a program that allows a user to input as many **double** values as desired (up to 32,767) and writes them to a disk file as they are entered. Call this file VALUES. Keep a count of the number of values entered, and write this number to a file called COUNT.
2. Using the file you created in Exercise 1, write a program that first reads the number of items in VALUES from COUNT. Next, read the values in VALUES and display them.

9.6 UNDERSTAND RANDOM ACCESS

So far, the examples have either written or read a file sequentially from its beginning to its end. However, using another of C's file system functions, you can access any point in a file at any time. The function that lets you do this is called **fseek( )**, and its prototype is

int fseek(FILE **fp*, long *offset*, int *origin*);

Here, *fp* is associated with the file being accessed. The value of *offset* determines the number of bytes from *origin* to make the new current position. *origin* must be one of these macros, shown here with their meanings:

Origin	**Meaning**
SEEK_SET	Seek from start of file
SEEK_CUR	Seek from current location
SEEK_END	Seek from end of file

These macros are defined in STDIO.H. For example, if you wanted to set the current location 100 bytes from the start of the file, then *origin* will be **SEEK_SET** and *offset* will be 100.

The **fseek( )** function returns zero when successful and nonzero if a failure occurs. In most implementations, you may seek past the end of the file, but you may never seek to a point before the start of the file.

You can determine the current location of a file using **ftell( )**, another of C's file system functions. Its prototype is

```
long ftell(FILE *fp);
```

It returns the location of the current position of the file associated with *fp*. If a failure occurs, it returns –1.

In general, you will want to use random access only on binary files. The reason for this is simple. Because text files may have character translations performed on them, there may not be a direct correspondence between what is in the file and the byte to which it would appear that you want to seek. The only time you should use **fseek( )** with a text file is when seeking to a position previously determined by **ftell( )**, using **SEEK_SET** as the origin.

Remember one important point: Even a file that contains only text can be opened as a binary file, if you like. There is no inherent restriction about random access on files containing text. The restriction applies only to files opened *as* text files.

EXAMPLES

1. The following program uses **fseek( )** to report the value of any byte within the file specified on the command line.

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

int main(int argc, char *argv[])
{
  long loc;
  FILE *fp;

  /* see if file name is specified */
  if(argc!=2) {
    printf("File name missing.\n");
    exit(1);
  }

  if((fp = fopen(argv[1] , "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  printf("Enter byte to seek to: ");
  scanf("%ld", &loc);
  if(fseek(fp, loc, SEEK_SET)) {
    printf("Seek error.\n");
    exit(1);
  }

  printf("Value at loc %ld is %d", loc, getc(fp));
  fclose(fp);

  return 0;
}
```

2. The following program uses **ftell( )** and **fseek( )** to copy the contents of one file into another in reverse order. Pay special attention to how the end of the input file is found. Since the program has sought to the end of the file, the program backs up one byte so that the current location of the file associated with **in** is at the last actual character in the file.

```
/* Copy a file in reverse order */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
```

```
long loc;
FILE *in, *out;
char ch;

/* see if correct number of command line arguments */
if(argc!=3) {
  printf("Usage: revcopy <source> <destination>.\n");
  exit(1);
}

if((in = fopen(argv[1], "rb"))==NULL) {
  printf("Cannot open input file.\n");
  exit(1);
}
if((out = fopen(argv[2], "wb"))==NULL) {
  printf("Cannot open output file.\n");
  exit(1);
}

/  find end of source file */
fseek(in, 0L, SEEK_END);
loc = ftell(in);

/* copy file in reverse order */
loc = loc-1; /* back up past end-of-file mark */
while(loc >= 0L) {
  fseek(in, loc, SEEK_SET);
  ch = fgetc(in);
  fputc(ch, out);
  loc--;
}
fclose(in);
fclose(out);

return 0;
}
```

3. This program writes ten **double** values to disk. It then asks you which one you want to see. This example shows how you can randomly access data of any type. You simply need to multiply the size of the base data type by its index in the file.

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

```
double d[10] = {
  10.23, 19.87, 1002.23, 12.9, 0.897,
  11.45, 75.34, 0.0, 1.01, 875.875
};

int main(void)
{
  long loc;
  double value;
  FILE *fp;

  if((fp = fopen("myfile", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* write the entire array in one step */
  if(fwrite(d, sizeof d, 1, fp) != 1) {
    printf("Write error.\n");
    exit(1);
  }
  fclose(fp);

  if((fp = fopen("myfile", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  printf("Which element? ");
  scanf("%ld", &loc);
  if(fseek(fp, loc*sizeof(double), SEEK_SET)) {
    printf("Seek error.\n");
    exit(1);
  }

  fread(&value, sizeof(double), 1, fp);
  printf("Element %ld is %f", loc, value);

  fclose(fp);

  return 0;
```

1. Write a program that uses **fseek( )** to display every other byte in a text file. (Remember, you must open the text file as a binary file in order for **fseek( )** to work properly.) Have the user specify the file on the command line.
2. Write a program that searches a file, specified on the command line, for a specific integer value (also specified on the command line). If this value is found, have the program display its location, in bytes, relative to the start of the file.

9.7 LEARN ABOUT VARIOUS FILE-SYSTEM FUNCTIONS

You can rename a file using **rename( )**, shown here:

```
int rename(char *oldname, char *newname);
```

Here, *oldname* points to the original name of the file and *newname* points to its new name. The function returns zero if successful and nonzero if an error occurs.

You can erase a file using **remove( )**. Its prototype is

```
int remove(char *file-name);
```

This function will erase the file whose name matches that pointed to by *file-name*. It returns zero if successful and nonzero if an error occurs.

You can position a file's current location to the start of the file using **rewind( )**. Its prototype is

```
void rewind(FILE *fp);
```

It rewinds the file associated with *fp*. The **rewind( )** function has no return value, because any file that has been successfully opened can be rewound.

Although seldom necessary because of the way C's file system works, you can cause a file's disk buffer to be flushed using **fflush( )**. Its prototype is

int fflush(FILE **fp*);

It flushes the buffer of the file associated with *fp*. The function returns zero if successful, **EOF** if a failure occurs. If you call **fflush( )** using a **NULL** for *fp*, all existing disk buffers are flushed.

EXAMPLES

1. This program demonstrates **remove( )**. It prompts the user for the file to erase and also provides a safety check in case the user entered the wrong name.

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

int main(void)
{
  char fname[80];

  printf("Enter name of file to erase: ");
  gets(fname);
  printf("Are you sure? (Y/N) ");
  if(toupper(getchar())=='Y') remove(fname);

  return 0;
}
```

2. The following program demonstrates **rewind( )** by displaying the contents of the file specified on the command line twice.

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

int main(int argc, char *argv[])
```

```
{
  FILE *fp;

  /* see if file name is specified */
  if(argc!=2) {
    printf("File name missing.\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* show it once */
  while(!feof(fp))
    putchar(getc(fp));

  rewind(fp);

  /* show it twice */
  while(!feof(fp))
    putchar(getc(fp));

  fclose(fp);

  return 0;
}
```

3. This fragment causes the buffer associated with **fp** to be flushed to disk.

```
FILE *fp;
.
.
.
fflush(fp);
```

4. This program renames a file called MYFILE.TXT to YOURFILE.TXT.

```
#include <stdio.h>

int main(void)
{
  if(rename("myfile.txt", "yourfile.txt"))
```

```
      printf("Rename failed.\n");
    else
      printf("Rename successful.\n");

    return 0;
  }
```

EXERCISES

1. Improve the erase program so that it notifies the user if he or she tries to remove a nonexistent file.
2. On your own, think of ways that **rewind( )** and **fflush( )** could be useful in real applications.

9.8 LEARN ABOUT THE STANDARD STREAMS

When a C program begins execution, three streams are automatically opened and available for use. These streams are called *standard input* (**stdin**), *standard output* (**stdout**), and *standard error* (**stderr**). By default, they refer to the console, but in environments that support redirectable I/O, they can be redirected by the operating system to some other device.

Normally, **stdin** inputs from the keyboard; **stdout** and **stderr** write to the screen. These standard streams are **FILE** pointers and may be used with any function that requires a variable of type **FILE ***. For example, you can use **fprintf( )** to print formatted output to the screen by specifying **stdout** as its output stream. The following two statements are functionally the same:

```
fprintf(stdout, "%d %c %s", 100, 'c', "this is a string");
printf("%d %c %s", 100, 'c', "this is a string");
```

In actuality, C makes little distinction between console I/O and file I/O. As just shown, it is possible to perform console I/O using several of the file-system functions. Although it may come as a bit of a surprise, it is also possible to perform disk file I/O using console I/O functions, such as **printf( )**. Here's why.

All of the functions described in Chapter 8 and referred to as "console I/O functions" are actually special-case file-system functions that automatically operate on **stdin** and **stdout**. Thus, the console I/O functions are just conveniences for you, the programmer. As far as C is concerned, the console is simply another hardware device. You don't actually need the console functions to access the console. Any file-system function can access it. (Of course, non-standard I/O functions like **getche( )** are differentiated from the standard file-system functions and do, in fact, operate only on the console.) In environments that allow redirection of I/O, **stdin** and **stdout** could refer to devices other than the keyboard and screen. Since the console functions operate on **stdin** and **stdout**, if these streams are redirected, the "console" functions can be made to operate on other devices. For example, by redirecting the **stdout** to a disk file, you can use a "console" I/O function to write to a disk file.

One important point: **stdin**, **stdout**, and **stderr** are not variables. They may not be assigned a value using **fopen( )**, nor should you attempt to close them using **fclose( )**. These streams are maintained internally by the compiler. You are free to use them, but not to change them.

EXAMPLES

1. Consider this program:

```
#include <stdio.h>

int main(void)
{
  printf("This is an example of redirection.\n");

  return 0;
}
```

Assume that this program is called TEST. If you execute TEST normally, it displays the string on the screen. However, if an environment supports redirection of I/O, **stdout** can be redirected to a file. For example, in a DOS, OS/2, Windows, or UNIX environment, executing TEST like this

```
TEST > OUTPUT
```

causes the output of TEST to be written to a file called OUTPUT. You might want to try this now for yourself.

2. Input can also be redirected. For example, consider the following program:

```
#include <stdio.h>

int main(void)
{
  int i;

  scanf("%d", &i);
  printf("%d", i);

  return 0;
}
```

Assuming it is called TEST, executing it as

```
TEST < INPUT
```

causes **stdin** to be directed to the file called INPUT. Assuming that INPUT contained the ASCII representation for an integer, the value of this integer will be read from the file and printed on the screen.

3. As mentioned earlier in this book, when using **gets( )** it is possible to overrun the array that is being used to receive the characters entered by the user because **gets( )** provides no bounds checking. One way around this problem is to use **fgets( )**, specifying **stdin** for the input stream. Since **fgets( )** requires you to specify a maximum length, it is possible to prevent an array overrun. The only trouble is that **fgets( )** does not remove the newline character and **gets( )** does. This means that you will have to manually remove it, as shown in the following program:

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

```
int main(void)
{
  char str[10];
  int i;

  printf("Enter a string: ");
  fgets(str, 10, stdin);

  /* remove newline, if present */
  i = strlen(str)-1;
  if(str[i]=='\n') str[i] = '\0';

  printf("This is your string: %s", str);

  return 0;
}
```

EXERCISES

1. Write a program that copies the contents of one text file to another. However, use only "console" I/O functions and redirection to accomplish the file copy.
2. On your own, experiment using **fgets( )** to read strings entered from the keyboard.

Mastery Skills Check

Before continuing, you should be able to answer these questions and complete these exercises:

1. Write a program that displays the contents of a text file (specified on the command line), one line at a time. After each line is displayed, ask the user if he or she wants to see another line.

2. Write a program that copies a text file. Have the user specify both file names on the command line. Have the copy program convert all lowercase letters into uppercase ones.
3. What do **fprintf( )** and **fscanf( )** do?
4. Write a program that uses **fwrite( )** to write 100 randomly generated integers to a file called RAND.
5. Write a program that uses **fread( )** to display the integers stored in the file called RAND, created in Exercise 4.
6. Using the file called RAND, write a program that uses **fseek( )** to allow the user to access and display the value of any integer in the file.
7. How do the "console" I/O functions relate to the file system?

Cumulative Skills Check

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. Enhance the card-catalog program you wrote in Chapter 8 so that it stores its information in a disk file called CATALOG. When the program begins, have it read the catalog into memory. Also, add an option to save the information to disk.
2. Write a program that copies a file. Have the user specify both the source and destination files on the command line. Have the program remove tab characters, substituting the appropriate number of spaces.
3. On your own, create a small database to keep track of anything you desire—your CD collection, for example.

10

Structures and Unions

chapter objectives

10.1 Master structure basics

10.2 Declare pointers to structures

10.3 Work with nested structures

10.4 Understand bit-fields

10.5 Create unions

IN this chapter you will learn about two of C's most important user-defined types: the structure and the union.

Review Skills Check

Before proceeding you should be able to answer these questions and perform these exercises:

1. Write a program that copies a file. Have the user specify both the source and destination file names on the command line. Include full error checking.
2. Write a program using **fprintf( )** to create a file that contains this information:

   ```
   this is a string 1230.23 1FFF A
   ```

   Use a string, a **double**, a hexadecimal integer, and character format specifiers and values.
3. Write a program that contains a 20-element integer array. Initialize the array so that it contains the numbers 1 through 20. Using only one **fwrite( )** statement, save this array to a file called TEMP.
4. Write a program that reads the TEMP file created in Exercise 3 into an integer array using only one **fread( )** statement. Display the contents of the array.
5. What are **stdin**, **stdout**, and **stderr**?
6. How do functions like **printf( )** and **scanf( )** relate to the C file system?

10.1 MASTER STRUCTURE BASICS

A *structure* is an *aggregate* (or *conglomerate*) data type that is composed of two or more related variables called *members*. Unlike an array in which each element is of the same type, each member of a structure can have its own type, which may differ from the types of the other members. Structures are defined in C using this general form:

```
struct tag-name {
  type member1;
  type member2;
  type member3;
  .
  .
  .
  type memberN;
} variable-list;
```

The keyword **struct** tells the compiler that a structure type is being defined. Each *type* is a valid C type. The *tag-name* is essentially the type name of the structure, and the *variable-list* is where actual instances of the structure are declared. Either the *tag-name* or the *variable-list* is optional, but one must be present (you will see why shortly). The members of a structure are also commonly referred to as *fields* or *elements*. This book will use these terms interchangeably.

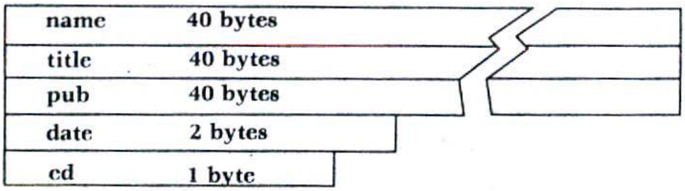
Generally, the information contained in a structure is logically related. For example, you might use a structure to hold a person's address. Another structure might be used to support an inventory program in which each item's name, retail and wholesale cost, and the quantity on hand are stored. The structure shown here defines fields that can hold card-catalog information:

```
struct catalog  {
  char name[40];     /* author name */
  char title[40];    /* title */
  char pub[40];      /* publisher */
  unsigned date;     /* copyright date */
  unsigned char ed;  /* edition */
} card;
```

Here, **catalog** is the type name of the structure. It is not the name of a variable. The only variable defined by this fragment is **card**. It is important to understand that a structure declaration defines only a logical entity, which is a new data type. It is not until variables of that type are declared than an object of that type actually exists. Thus, **catalog** is a logical template; **card** has physical reality. Figure 10-1

FIGURE 10-1

*How the **card** structure variable appears in memory (assuming 2-byte integers)*



shows how this structure will appear in memory (using 2-byte integers).

To access a member of a structure, you must specify both the structure variable name and the member name, separated by a period. For example, using **card**, the following statement assigns the **date** field the value 1776:

```
card.date = 1776;
```

C programmers often refer to the period as the *dot operator*. To print the copyright date, you can use a statement such as:

```
printf("Copyright date: %u", card.date);
```

To input the date, use a **scanf( )** statement such as:

```
scanf("%u", &card.date);
```

Notice that the **&** goes before the structure name, not before the member name. In a similar fashion, these statements input the author's name and output the title:

```
gets(card.name);
printf("%s", card.title);
```

To access an individual character in the **title** field, simply index **title**. For example, the following statement prints the third letter:

```
printf("%c", card.title[2]);
```

Once you have defined a structure type, you can create additional variables of that type using this general form:

struct *tag_name var_list*;

Assuming, for example, that **catalog** has been defined as shown earlier in this section, this statement declares three variables of type **struct catalog**:

```
struct catalog var1, var2, var3;
```

This is why it is not necessary to declare any variables when the structure type is defined. You can declare them separately, as needed.

A key concept to understand is that each instance of a structure contains its own copy of the members of the structure. For example, given the preceding declaration, the **title** field of **var1** is completely separate from the **title** field of **var2**. In fact, the only relationship that **var1**, **var2**, and **var3** have with one another is that they are all variables of the same type of structure. There is no other linkage among the three.

If you know you only need a fixed number of structure variables, you do not need to specify the tag name. For example, this code creates two structure variables, but the structure itself is unnamed:

```
struct {
  int a;
  char ch;
} var1, var2;
```

In actual practice, however, you will usually want to specify the tag name.

Structures can be arrayed in the same fashion as other data types. For example, the following structure definition creates a 100-element array of structures of type **catalog**:

```
struct catalog cat[100];
```

To access an individual structure of the array, you must index the array name. For example, the following accesses the first structure:

```
cat[0]
```

To access a member within a specified structure, follow the index with a period and the name of the member you want. For example, the following statement loads the **ed** field of structure 33 with the value of 2:

```
cat[33].ed = 2;
```

Structures may be passed as parameters to functions just like any other type of value. A function may also return a structure.

You may assign the contents of one instance of a structure to another as long as they are both of the same type. For example, this fragment is perfectly valid:

```
struct s_type {
  int a;
  float f;
} var1, var2;

var1.a = 10;
var1.f = 100.23;

var2 = var1;
```

After this fragment executes, **var2** will contain exactly the same thing as **var1**.

EXAMPLES

1. This program demonstrates some ways to access structure members:

```
#include <stdio.h>

struct s_type {
  int i;
  char ch;
  double d;
  char str[80];
} s;

int main(void)
{
  printf("Enter an integer: ");
  scanf("%d:", &s.i);
  printf("Enter a character: ");
  scanf(" %c", &s.ch);
  printf("Enter a floating point number: ");
  scanf("%lf", &s.d);
  printf("Enter a string: ");
  scanf("%s", s.str);

  printf("%d %c %f %s", s.i, s.ch, s.d, s.str);

  return 0;
}
```

2. When you need to know the size of a structure, you should use the **sizeof** compile-time operator. Do not try to manually add up the number of bytes in each field. There are three good reasons for this. First, as you learned in the preceding chapter, using **sizeof** ensures that your code is portable to different environments. Second, in some situations, the compiler may need to align certain types of data on even word boundaries. In this case, the size of the structure will be larger than the sum of its individual elements. Finally, for computers based on the 8086 family of CPUs (such as the 80486 or the Pentium), there are several different ways the compiler can organize memory. Some of these ways cause pointers to take up twice the space they do when memory is arranged differently.

   When using **sizeof** with a structure type, you must precede the tag name with the keyword **struct**, as shown in this program:

```
#include <stdio.h>

struct s_type {
  int i;
  char ch;
  int *p;
  double d;
} s;

int main(void)
{
  printf("s_type is %d bytes long", sizeof(struct s_type));

  return 0;
}
```

3. To see how useful arrays of structures are, examine an improved version of the card-catalog program developed in the preceding two chapters. Notice how using a structure makes it easier to organize the information about each book. Also notice

how the entire structure array is written and read from disk in a single operation.

```
/* An electronic card catalog. */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

int menu(void);
void display(int i);
void author_search(void);
void title_search(void);
void enter(void);
void save(void);
void load(void);

struct catalog {
  char name[80];     /* author name */
  char title[80];    /* title */
  char pub[80];      /* publisher */
  unsigned date;     /* copyright date */
  unsigned char ed; /* edition */
} cat[MAX];

int top = 0; /* last location used */

int main(void)
{
  int choice;

  load(); /* read in catalog */

  do {
    choice = menu();
    switch(choice) {
      case 1: enter(); /* enter books */
        break;
      case 2: author_search(); /* search by author */
        break;
      case 3: title_search(); /* search by title */
        break;
      case 4: save();
    }
  } while(choice!=5);

  return 0;
}

/* Return a menu selection. */
menu(void)
{
  int i;
  char str[80];

  printf("Card catalog:\n");
  printf("  1. Enter\n");
  printf("  2. Search by Author\n");
  printf("  3. Search by Title\n");
  printf("  4. Save catalog\n");
  printf("  5. Quit\n");

  do {
    printf("Choose your selection: ");
    gets(str);
    i = atoi(str);
    printf("\n");
  } while(i<1 || i>5);

  return i;
}

/* Enter books into database. */
void enter(void)
{
  int i;
  char temp[80];

  for(i=top; i<MAX; i++) {
    printf("Enter author name (ENTER to quit): ");
    gets(cat[i].name);
    if(!*cat[i].name) break;
    printf("Enter title: ");
    gets(cat[i].title);
    printf("Enter publisher: ");
    gets(cat[i].pub);
    printf("Enter copyright date: ");
    gets(temp);
    cat[i].date = (unsigned) atoi(temp);
```

```
      printf("Enter edition: ");
      gets(temp);
      cat[i].ed = (unsigned char) atoi(temp);
    }
    top = i;
}

/* Search by author. */
void author_search(void)
{
    char name[80];
    int i, found;

    printf("Name: ");
    gets(name);

    found = 0;
    for(i=0; i<top; i++)
      if(!strcmp(name, cat[i].name)) {
        display(i);
        found = 1;
        printf("\n");
      }

    if(!found) printf("Not Found\n");
}

/* Search by title. */
void title_search(void)
{
    char title[80];
    int i, found;

    printf("Title: ");
    gets(title);

    found = 0;
    for(i=0; i<top; i++)
      if(!strcmp(title, cat[i].title)) {
        display(i);
        found = 1;
        printf("\n");
      }
    if(!found) printf("Not Found\n");
}
```

```
/* Display catalog entry. */
void display(int i)
{
  printf("%s\n", cat[i].title);
  printf("by %s\n", cat[i].name);
  printf("Published by %s\n", cat[i].pub);
  printf("Copyright: %u, %u edition\n", cat[i].date,
          cat[i].ed);
}

/* Load the catalog file. */
void load(void)
{
  FILE *fp;

  if((fp = fopen("catalog", "rb"))==NULL) {
    printf("Catalog file not on disk.\n");
    return;
  }

  if(fread(&top, sizeof top, 1, fp) != 1) {  /* read count */
    printf("Error reading count.\n");
    exit(1);
  }
  if(fread(cat, sizeof cat, 1, fp) != 1) { /* read data */
    printf("Error reading catalog data.\n");
    exit(1);

  }

  fclose(fp);
}

/* Save the catalog file. */
void save(void)
{
  FILE *fp;

  if((fp = fopen("catalog", "wb"))==NULL) {
    printf("Cannot open catalog file.\n");
    exit(1);
  }
  if(fwrite(&top, sizeof top, 1, fp) != 1) {  /* write count */
    printf("Error writing count.\n");
    exit(1);
  }
  if(fwrite(cat, sizeof cat, 1, fp) != 1) { /* write data */
    printf("Error writing catalog data.\n");
    exit(1);
  }

  fclose(fp);
}
```

4. In the preceding example, the entire catalog array is stored on disk, even if the array is not full. If you like, you can change the **load( )** and **save( )** routines as follows, so that only structures actually holding data are stored on disk:

```
/* Load the catalog file. */
void load(void)
{
  FILE *fp;
  int i;

  if((fp = fopen("catalog", "rb"))==NULL) {
    printf("Catalog file not on disk.\n");
    return;
  }

  if(fread(&top, sizeof top, 1, fp) != 1) {  /* read count */
    printf("Error reading count.\n");
    exit(1);
  }
  for(i=0; i<=top; i++) /* read data */
    if(fread(&cat[i], sizeof(struct catalog), 1, fp)!= 1) {
      printf("Error reading catalog data.\n");
      exit(1);
    }

  fclose(fp);
}
```

```
/* Save the catalog file. */
void save(void)
{
  FILE *fp;
  int i;

  if((fp = fopen("catalog", "wb"))==NULL) {
    printf("Cannot open catalog file.\n");
    exit(1);
  }

  if(fwrite(&top, sizeof top, 1, fp) != 1) {  /* write count */
    printf("Error writing count.\n");
    exit(1);
  }
  for(i=0; i<=top; i++) /* write data */
    if(fwrite(&cat[i], sizeof(struct catalog), 1, fp)!= 1) {
      printf("Error writing catalog data.\n");
      exit(1);
    }

  fclose(fp);
}
```

5. The names of structure members will not conflict with other variables using the same names. Because the member name is linked with the structure name, it is separate from other variables of the same name. For example, this program prints **10 100 101** on the screen.

```
#include <stdio.h>

int main(void)
{
  struct s_type {
    int i;
    int j;
  } s;

  int i;

  i = 10;
```

```
    s.i = 100;
    s.j = 101;

    printf("%d %d %d", i, s.i, s.j);

    return 0;
  }
```

The variable **i** and the structure member **i** have no relationship to each other.

6. As stated earlier, a function may return a structure to the calling procedure. The following program, for example, loads the members of **var1** with the values **100** and **123.23** and then displays them on the screen:

```
#include <stdio.h>

struct s_type {
  int i;
  double d;
};

struct s_type f(void);

int main(void)
{
  struct s_type var1;

  var1 = f();
  printf("%d %f", var1.i, var1.d);

  return 0;
}

struct s_type f(void)
{
  struct s_type temp;

  temp.i = 100;
  temp.d = 123.23;

  return temp;
}
```

7. This program passes a structure to a function:

```
#include <stdio.h>

struct s_type {
  int i;
  double d;
};

void f(struct s_type temp);

int main(void)
{
  struct s_type var1;

  var1.i = 99;
  var1.d = 98.6;
  f(var1);

  return 0;
}

void f(struct s_type temp)
{
  printf("%d %f", temp.i, temp.d);
}
```

EXERCISES

1. In Chapter 9, you wrote a program that created a telephone directory that was stored on disk. Improve the program so that it uses an array of structures, each containing a person's name, area code, and telephone number. Store the area code as an integer. Store the name and telephone number as strings. Make the array **MAX** elements long, where **MAX** is any convenient value that you choose.
2. What is wrong with this fragment?

```
struct s_type {
  int i;
```

```
    long l;
    char str[80];
  } s;
  .
  .
  .
  i = 10;
```

3. On your own, examine the header file STDIO.H and look at how the **FILE** structure is defined.

DECLARE POINTERS TO STRUCTURES

It is very common to access a structure through a pointer. You declare a pointer to a structure in the same way that you declare a pointer to any other type of variable. For example, the following fragment defines a structure called **s_type** and declares two variables. The first, **s**, is an actual structure variable. The second, **p**, is a pointer to structures of type **s_type**.

```
struct s_type {
  int i;
  char str[80];
} s, *p;
```

Given this definition, the following statement assigns to **p** the address of **s**:

```
p = &s:
```

Now that **p** points to **s** you can access **s** through **p**. However, to access an individual element of **s** using **p** you cannot use the dot operator. Instead, you must use the *arrow operator*, as shown in the following example:

```
p->i = 1;
```

This statement assigns the value 1 to element **i** of **s** through **p**. The arrow operator is formed using a minus sign followed by a greater-than sign. There must be no spaces between the two.

C passes structures to functions in their entirety. However, if the structure is very large, the passing of a structure can cause a considerable reduction in a program's execution speed. For this reason, when working with large structures, you might want to pass a pointer to a structure in situations that allow it instead of passing the structure itself.

When accessing a member using a structure variable, use the dot operator. When accessing a member using a pointer, use the arrow operator.

EXAMPLES

1. The following program illustrates how to use a pointer to a structure:

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

struct s_type {
  int i;
  char str[80];
} s, *p;

int main(void)
{
  p = &s;

  s.i = 10;  /* this is functionally the same */
  p->i = 10; /* as this */
  strcpy(p->str, "I like structures.");

  printf("%d %d %s", s.i, p->i, p->str);

  return 0;
}
```

2. One very useful application of structure pointers is found in C's time and date functions. Several of these functions use a pointer to the current time and date of the system. The time and date functions require the header file TIME.H, in which a structure called **tm** is defined. This structure can hold the date and time broken down into its elements. This is called the *broken-down time*. The **tm** structure is defined as follows:

```
struct tm {
  int tm_sec;    /* seconds, 0-61 */
  int tm_min;    /* minutes, 0-59; */
  int tm_hour;   /* hours, 0-23 */
  int tm_mday;   /* day of the month, 1-31*/;
  int tm_mon;    /* months since Jan, 0-11 */
  int tm_year;   /* years from 1900 */
  int tm_wday;   /* days since Sunday, 0-6*/
  int tm_yday;   /* days since Jan 1,  0-365 */
  int tm_isdst; /* Daylight Saving Time indicator */
};
```

The value of **tm_isdst** will be positive if Daylight Saving Time is in effect, zero if it is not in effect, and negative if there is no information available. Also defined in TIME.H is the type **time_t**. It is essentially a **long** integer capable of representing the time and date of the system in an encoded implementation-specific internal format. This is referred to as the *calendar time*. To obtain the calendar time of the system, you must use the **time( )** function, whose prototype is:

time_t time(time_t **systime*);

The **time( )** function returns the encoded calendar time of the system or –1 if no system time is available. It also places this encoded form of the time into the variable pointed to by *systime*. However, if *systime* is null, the argument is ignored.

Since the calendar time is represented using an implementation-specified internal format, you must use another of C's time and date functions to convert it into a form that is easier to use. One of these functions is called **localtime( )**. Its prototype is

struct tm *localtime(time_t **systime*);

The **localtime( )** function returns a pointer to the broken-down form of *systime*. The structure that holds the broken-down time is internally allocated by the compiler and will be overwritten by each subsequent call.

This program demonstrates **time( )** and **localtime( )** by displaying the current time of the system:

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

int main(void)
{
  struct tm *systime;
  time_t t;

  t = time(NULL);
  systime = localtime(&t);

  printf("Time is %.2d:%.2d:%.2d\n", systime->tm_hour,
         systime->tm_min, systime->tm_sec);
  printf("Date: %.2d/%.2d/%.2d", systime->tm_mon+1,
         systime->tm_mday, systime->tm_year);

  return 0;
}
```

Here is sample output produced by this program:

```
Time is 10:32:49
Date: 03/15/97
```

EXERCISES

1. Is this program fragment correct?

```
struct s_type {
  int a;
  int b;
} s, *p
```

```
  int main(void)
  {
    p = &s;

    p.a = 100;
  .
  .
  .
```

2. Another of C's time and date functions is called **gmtime( )**. Its prototype is

   struct tm *gmtime(time_t **time*);

   The **gmtime( )** function works exactly like **localtime( )**, except that it returns the Coordinated Universal Time (which is, essentially, Greenwich Mean Time) of the system. Change the program in Example 2 so that it displays both local time and Coordinated Universal Time. (Note: Coordinated Universal Time may not be available on your system.)

WORK WITH NESTED STRUCTURES

So far, we have only been working with structures whose members consist solely of C's basic types. However, members can also be other structures. These are referred to as *nested structures*. Here is an example that uses nested structures to hold information on the performance of two assembly lines, each with ten workers:

```
#define NUM_ON_LINE 10

struct worker {
  char name[80];
  int avg_units_per_hour;
  int avg_errs_per_hour;
};

struct asm_line {
  int product_code;
  double material_cost;
  struct worker wkers[NUM_ON_LINE];
} line1, line2;
```

To assign the value 12 to the **avg_units_per_hour** of the second **wkers** structure of **line1**, use this statement:

```
line1.wkers[1].avg_units_per_hour = 12;
```

As you see, the structures are accessed from the outer to the inner. This is also the general case. Whenever you have nested structures, you begin with the outermost and end with the innermost.

EXAMPLE

1. A nested structure can be used to improve the card catalog program. Here, the mechanical information about each book is stored in its own structure, which, in turn, is part of the **catalog** structure. The entire catalog program using this approach is shown here. Notice how the program now stores the length of the book in pages.

```
/* An electronic card catalog--3rd Improvement. */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

int menu(void);
void display(int i);
void author_search(void);
void title_search(void);
void enter(void);
void save(void);
void load(void);

struct book_type {
  unsigned date;     /* copyright date */
  unsigned char ed;  /* edition */
  unsigned pages;    /* length of book */
} ;
```

```
struct catalog {
  char name[80];  /* author name */
  char title[80]; /* title */
  char pub[80];   /* publisher */
  struct book_type book; /* mechanical info */
} cat[MAX];

int top = 0; /* last location used */

int main(void)
{
  int choice;

  load(); /* read in catalog */

  do {
    choice = menu();
    switch(choice) {
      case 1: enter(); /* enter books */
        break;
      case 2: author_search(); /* search by author */
        break;
      case 3: title_search(); /* search by title */
        break;
      case 4: save();
    }
  } while(choice!=5);

  return 0;
}

/* Return a menu selection. */
menu(void)
{
  int i;
  char str[80];

  printf("Card catalog:\n");
  printf("  1. Enter\n");
  printf("  2. Search by Author\n");
  printf("  3. Search by Title\n");
  printf("  4. Save catalog\n");
  printf("  5. Quit\n");

  do {
      printf("Choose your selection: ");
      gets(str);
      i = atoi(str);
      printf("\n");
    } while(i<1 || i>5);

    return i;
  }

  /* Enter books into database. */
  void enter(void)
  {
    int i;
    char temp[80];

    for(i=top; i<MAX; i++) {
      printf("Enter author name (ENTER to quit): ");
      gets(cat[i].name);
      if(!*cat[i].name) break;
      printf("Enter title: ");
      gets(cat[i].title);
      printf("Enter publisher: ");
      gets(cat[i].pub);
      printf("Enter copyright date: ");
      gets(temp);
      cat[i].book.date = (unsigned) atoi(temp);
      printf("Enter edition: ");
      gets(temp);
      cat[i].book.ed = (unsigned char) atoi(temp);
      printf("Enter number of pages: ");
      gets(temp);
      cat[i].book.pages = (unsigned) atoi(temp);
    }
    top = i;
  }

  /* Search by author. */
  void author_search(void)
  {
    char name[80];
    int i, found;

    printf("Name: ");
    gets(name);
```

✓

```
  found = 0;
  for(i=0; i<top; i++)
    if(!strcmp(name, cat[i].name)) {
    display(i);
    found = 1;
    printf("\n");
  }

  if(!found) printf("Not Found\n");
}

/* Search by title. */
void title_search(void)
{
  char title[80];
  int i, found;

  printf("Title: ");
  gets(title);

  found = 0;

  for(i=0; i<top; i++)
    if(!strcmp(title, cat[i].title)) {
     display(i);
     found = 1;
     printf("\n");
    }
  if(!found) printf("Not Found\n");
}

/* Display catalog entry. */
void display(int i)
{
  printf("%s\n", cat[i].title);
  printf("by %s\n", cat[i].name);
  printf("Published by %s\n", cat[i].pub);
  printf("Copyright: %u, edition: %u\n",
         cat[i].book.date, cat[i].book.ed);
  printf("Pages: %u\n", cat[i].book.pages);
}

/* Load the catalog file. */
```

```
void load(void)
{
  FILE *fp;

  if((fp = fopen("catalog", "rb"))==NULL) {
    printf("Catalog file not on disk.\n");

    return;
  }

  if(fread(&top, sizeof top, 1, fp) != 1) {  /* read count */
    printf("Error reading count.\n");
    exit(1);
  }
  if(fread(cat, sizeof cat, 1, fp) != 1) { /* read data */
    printf("Error reading catalog data.\n");
    exit(1);
  }

  fclose(fp);
}

/* Save the catalog file. */
void save(void)
{
  FILE *fp;

  if((fp = fopen("catalog", "wb"))==NULL) {
    printf("Cannot open catalog file.\n");
    exit(1);
  }

  if(fwrite(&top, sizeof top, 1, fp) != 1) {  /* write count */
    printf("Error writing count.\n");
    exit(1);
  }
  if(fwrite(cat, sizeof cat, 1, fp) != 1) { /* write data */
    printf("Error writing catalog data.\n");
    exit(1);
    }

    fclose(fp);
  }
```

EXERCISES

1. Improve the telephone-directory program you wrote earlier in this chapter so that it includes each person's mailing address. Store the address in its own structure, called **address**, which is nested inside the directory structure.

10.4 UNDERSTAND BIT-FIELDS

C allows a variation on a structure member called a *bit-field. A bit-field* is composed of one or more bits. Using a bit-field, you can access by name one or more bits within a byte or word. To define a bit-field, use this general form:

type name : *size*;

Here, *type* is either **int** or **unsigned**. If you specify a signed bit-field, then the high-order bit is treated as a sign bit, if possible. The number of bits in the field is specified by *size*. Notice that a colon separates the name of the bit-field from its size in bits.

Bit-fields are useful when you want to pack information into the smallest possible space. For example, here is a structure that uses bit-fields to hold inventory information.

```
struct b_type {
  unsigned department: 3;  /* up to 7 departments */
  unsigned instock: 1;     /* 1 if in stock, 0 if out */
  unsigned backordered: 1; /* 1 if backordered, 0 if not */
  unsigned lead_time: 3;    /* order lead time in months */
} inv[MAX_ITEM];
```

In this case one byte can be used to store information on an inventory item that would normally have taken four bytes without the use of bit-fields. You refer to a bit-field just like any other member of a structure. The following statement, for example, assigns the value 3 to the **department** field of item 10:

```
inv[9].department = 3;
```

The following statement determines whether item 5 is out of stock:

```
if(!inv[4].instock) printf("Out of Stock");
else printf("In Stock");
```

It is not necessary to completely define all bits within a byte or word. For example, this is perfectly valid:

```
struct b_type {
  int a: 2;
  int b: 3;
} ;
```

The C compiler is free to store bit-fields as it sees fit. However, usually the compiler will automatically store bit-fields in the smallest unit of memory that will hold them. Whether the bit-fields are stored high-order to low-order or the other way around is implementation-dependent. However, many compilers use high-order to low-order.

You can mix bit-fields with other types of members in a structure's definition. For example, this version of the inventory structure also includes room for the name of each item:

```
struct b_type {
  char name[40];            /* name of item */
  unsigned department: 3;   /* up to 7 departments */
  unsigned instock: 1;      /* 1 if in stock, 0 if not */
  unsigned backordered: 1;  /* 1 if backordered, 0 if not */
  unsigned lead_time: 3;    /* order lead time in months */
} inv[MAX_ITEM];
```

Because the smallest addressable unit of memory is a byte, you cannot obtain the address of a bit-field variable.

Bit-fields are often used to store Boolean (true/false) data because they allow the efficient use of memory—remember, you can pack eight Boolean values into a single byte.

EXAMPLES

1. It is not necessary to name every bit when using bit-fields. Here, for example, is a structure that uses bit-fields to access the first and last bit in a byte.

   ```
   struct b_type {
     unsigned first: 1;
     int : 6;
     unsigned last: 1;
   };
   ```

   The use of unnamed bit-fields makes it easy to reach the bits you are interested in.

2. To see how useful bit-fields can be when working with Boolean data, here is a crude simulation of a spaceship flight recorder. By packing all the relevant information into one byte, comparatively little disk space is used to record a flight.

   ```
   /* Simulation of a 100 minute spaceship
      flight recorder.
   */
   #include <stdlib.h>
   #include <stdio.h>

   /* all fields indicate OK if 1,
      malfunctioning or low if 0 */
   struct telemetry {
     unsigned fuel: 1;
     unsigned radio: 1;
     unsigned tv: 1;
     unsigned water: 1;
     unsigned food: 1;
     unsigned waste: 1;
   } flt_recd;

   void display(struct telemetry i);
   ```

```
int main(void)
{
  FILE *fp;
  int i;

  if((fp = fopen("flight", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* Imagine that each minute a status report of
     the spaceship is recorded on disk.
  */
  for(i=0; i<100; i++) {
    flt_recd.fuel = rand()%2;
    flt_recd.radio = rand()%2;
    flt_recd.tv = rand()%2;
    flt_recd.water = rand()%2;
    flt_recd.food = rand()%2;
    flt_recd.waste = rand()%2;

    display(flt_recd);
    fwrite(&flt_recd, sizeof flt_recd, 1, fp);
  }

  fclose(fp);

  return 0;
}

void display(struct telemetry i)
{
  if(i.fuel) printf("Fuel OK\n");
  else printf("Fuel low\n");
  if(i.radio) printf("Radio OK\n");
  else printf("Radio failure\n");
  if(i.tv) printf("TV system OK\n");
  else printf("TV malfunction\n");
  if(i.water) printf("Water supply OK\n");
  else printf("Water supply low\n");
  if(i.food) printf("Food supply OK\n");
  else printf("Food supply low\n");
  if(i.waste) printf("Waste containment OK\n");
```

```
    else printf("Waste containment failure\n");
    printf("\n");
  }
```

Depending on how your compiler packs the bit-fields, after you run this program, the file on disk may be as short as 100 bytes long. Now try the program after modifying the **telemetry** structure as shown here:

```
struct telemetry {
  char fuel;
  char radio;
  char tv;
  char water;
  char food;
  char waste;
} flt_recd;
```

In this version, no bit-fields are used and the resulting file is at least 600 bytes long. As you can see, using bit-fields can provide substantial space savings.

EXERCISES

1. Write a program that creates a structure that contains three bit-fields called **a**, **b**, and **c**. Make **a** and **b** three bits long and make **c** two bits long. Next, assign each a value and display the values.
2. Many compilers supply library functions that return the status of various hardware devices, such as a serial port or the keyboard, by encoding information in a bit-by-bit fashion. On your own, consult the user's manual for your compiler to see if it supports such functions. If it does, write some programs that read and decode the status of one or more devices.

10.5 CREATE UNIONS

In C, a *union* is a single piece of memory that is shared by two or more variables. The variables that share the memory may be of different types. However, only one variable may be in use at any one time. A union is defined much like a structure. Its general form is

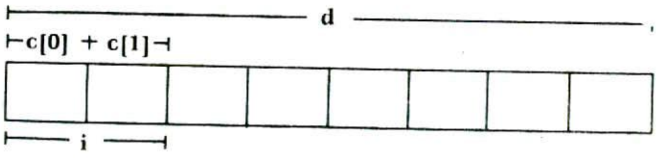
```
union tag-name {
  type member1;
  type member2;
  type member3;
  .
  .
  .
  type memberN;
} variable-names;
```

Like a structure, either the *tag-name* or the *variable-names* may be missing. Members may be of any valid C data type. For example, here is a union that contains three elements: an integer, a character array, and a **double**:

```
union u_type {
  int i;
  char c[2];
  double d;
} sample;
```

This union will appear in memory as shown in Figure 10-2.

FIGURE 10-2



*How an instance of the union **u_type** appears in memory (assuming 2-byte **ints** and 8-byte **doubles**)*

To access a member of a union, use the dot and arrow operators just as you do for structures. For example, this statement assigns **123.098** to **d** of **sample**:

```
sample.d = 123.098;
```

If you are accessing a union through a pointer, you must use the arrow operator. For example, assume that **p** points to **sample**. The following statement assigns **i** the value **101**:

```
p->i = 101;
```

It is important to understand that the size of a union is fixed at compile time and is large enough to accommodate the largest member of the union. Assuming 8-byte **doubles**, this means that **sample** will be 8 bytes long. Even if **sample** is currently used to hold an **int** value, it will still occupy 8 bytes of memory. As is the case with structures, you should use the **sizeof** compile-time operator to determine the size of a union. You should not simply assume that it will be the size of the largest element, because in some environments, the compiler may pad the union so that it aligns on a word boundary.

EXAMPLES

1. Unions are very useful when you need to interpret data in two or more different ways. For example, the **encode( )** function shown below uses a union to encode an integer by swapping its two low-order bytes. The same function can also be used to decode an encoded integer by swapping the already exchanged bytes back to their original positions.

```
#include <stdio.h>

int encode(int i);

int main(void)
{
  int i;

  i = encode(10); /* encode it */
  printf("10 encoded is %d\n", i);
  i = encode(i); /* decode it */
  printf("i decoded is %d", i);

  return 0;
}

/* Encode an integer, decode an encoded integer. */
int encode(int i)
{
  union crypt_type {
    int num;
    char c[2];
  } crypt;
  unsigned char ch;

  crypt.num = i;

  /* swap bytes */
  ch = crypt.c[0];
  crypt.c[0] = crypt.c[1];
  crypt.c[1] = ch;

  /* return encoded integer */
  return crypt.num;
}
```

The program displays the following:

```
10 encoded is 2560
i decoded is 10
```

2. The following program uses the union of a structure containing bit-fields and a character to display the binary representation of a character typed at the keyboard:

```
/* This program displays the binary code for a
   character entered at the keyboard.
*/
#include <stdio.h>
#include <conio.h>

struct sample {
  unsigned a: 1;
  unsigned b: 1;
  unsigned c: 1;
  unsigned d: 1;
  unsigned e: 1;
  unsigned f: 1;
  unsigned g: 1;
  unsigned h: 1;
};

union key_type {
  char ch;
  struct sample bits;
} key;

int main(void)
{
  printf("Strike a key: ");

  key.ch = getche();
  printf("\nBinary code is: ");

  if(key.bits.h) printf("1 ");
  else printf("0 ");
  if(key.bits.g) printf("1 ");
  else printf("0 ");
  if(key.bits.f) printf("1 ");
  else printf("0 ");
  if(key.bits.e) printf("1 ");
  else printf("0 ");
  if(key.bits.d) printf("1 ");
  else printf("0 ");
  if(key.bits.c) printf("1 ");
  else printf("0 ");
  if(key.bits.b) printf("1 ");
  else printf("0 ");
  if(key.bits.a) printf("1 ");
  else printf("0 ");

  return 0;
}
```

When a key is pressed, its ASCII code is assigned to **key.ch**, which is a **char**. This data is reinterpreted as a series of bit-fields, which allow the binary representation of the key to be displayed. Sample output is shown here:

```
Strike a key: X
Binary code is: 0 1 0 1 1 0 0 0
```

EXERCISES

1. Using a union composed of a **double** and an 8-byte character array, write a function that writes a **double** to a disk file, a character at a time. Write another function that reads this value from the file and reconstructs the value using the same union. (Note: If the length of a **double** for your compiler is not 8 bytes, use an appropriately sized character array.)
2. Write a program that uses a union to convert an **int** into a **long**. Demonstrate that it works.

Mastery Skills Check

At this point you should be able to answer these questions and perform these exercises:

1. In general terms what is a structure, and what is a union?
2. Show how to create a structure type called **s_type** that contains these five members:

   ```
   char ch;
   float d;
   int i;
   char str[80];
   double balance;
   ```

   Also, define one variable called **s_var** using this structure.
3. What is wrong with this fragment?

   ```
   struct s_type {
     int a;
   ```

```
    char b;
    float bal;
  } myvar, *p;

  p = &myvar;

  p.a = 10;
```

4. Write a program that uses an array of structures to store employee names, telephone numbers, hours worked, and hourly wages. Allow for 10 employees. Have the program input the information and save it to a disk file. Call the file EMP.
5. Write a program that reads the EMP file created in Exercise 4 and displays the information on the screen.
6. What is a bit-field?
7. Write a program that displays individually the values of the high- and low-order bytes of a short integer. (Hint: Use a union that contains as its two elements a short integer and a two-byte character array.)

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. Write a program that contains two structure variables defined as:

   ```
   struct s_type {
     int i;
     char ch;
     double d;
   } var1, var2;
   ```

   Have the program give each member of both structures initial values, but make sure that the values differ between the two structures. Using a function called **struct_swap( )**, have the program swap the contents of **var1** and **var2**.
2. As you know from Chapter 9, **fgetc( )** returns an integer value, even though it only reads a character from a file. Write a

program that copies one file to another. Assign the return value of **fgetc( )** to a union that contains an integer and character member. Use the integer element to check for **EOF**. Write the character element to the destination file. Have the user specify both the source and destination file names on the command line.

3. What is wrong with this fragment?

```
struct s_type {
  int a;
  int b: 2;
  int c: 6;
} var;
.
.
.
scanf("%d", &var);
```

4. In C, as you know, you cannot pass an array to a function as a parameter. (Only a pointer to an array can be passed.) However, there is one way around this restriction. If you enclose the array within a structure, the array is passed using the standard call-by value convention. Write a program that demonstrates this by passing a string inside a structure to a function, altering its contents inside the function and demonstrating that the original string is not altered after the function returns.

11

Advanced Data Types and Operators

chapter objectives

11.1 Use the storage class specifiers

11.2 Use the access modifiers

11.3 Define enumerations

11.4 Understand **typedef**

11.5 Use C's bitwise operators

11.6 Master the shift operators

11.7 Understand the **?** operator

11.8 Do more with the assignment operator

11.9 Understand the comma operator

11.10 Know the precedence summary

THE C language includes a rich set of data type modifiers that allow you to better fit the type of a variable to the information it will be storing. Also, C includes a number of special operators that permit the creation of very efficient routines. Both of these items are the subject of this chapter.

Review Skills Check

Before proceeding, you should be able to answer these questions and perform these exercises:

1. Write a program that uses an array of structures to hold the squares and cubes of the numbers 1 through 10. Display the contents of the array.
2. Write a program that uses a **union** to display as a character the individual bytes that make up a short integer entered by the user.
3. What does this fragment display? (Assume two-byte **ints** and eight-byte **doubles**.)

```
union {
  int i;
  double d;
} uvar;

printf("%d", sizeof uvar);
```

4. What is wrong with this fragment?

```
struct {
  int i;
  char str[80];
  double balance;
} svar;

svar->i = 100;
```

5. What is a bit-field?

11.1 USE THE STORAGE CLASS SPECIFIERS

C defines four type modifiers that affect how a variable is stored. They are

auto
extern
register
static

These specifiers precede the type name. Let's look at each now.

The specifier **auto** is completely unnecessary. It is provided in C to allow compatibility with its predecessor, B. Its use is to declare *automatic variables*. Automatic variables are simply local variables, which are **auto** by default. You will almost never see **auto** used in any C program.

Although the programs we have been working with in this book are fairly short, programs in the real world tend to be quite long. As the size of a program grows, it takes longer to compile. For this reason, C allows you to break a program into two or more files. You can separately compile these files and then link them together. This saves compilation time and makes your projects easier to work with. (The actual method of separate compilation and linking will be explained in the instructions that accompany your compiler.) When working with multiple source files there is, however, one issue that needs to be addressed. As a general rule, global data can only be defined once. However, global data may need to be accessed by two or more files that form a program. In this case, each source file must inform the compiler about the global data it uses. To accomplish this you will need to use the keyword **extern**. To understand why, consider the following program, which is split between two files:

FILE #1:

```
#include <stdio.h>

int count;

void f1(void);

int main(void)
{
  int i;
  f1(); /* set count's value */

  for(i=0; i<count; i++)
    printf("%d ", i);

  return 0;
}
```

FILE #2:

```
#include <stdlib.h>

void f1(void)
{
  count = rand();
}
```

If you try to compile the second file, an error will be reported because **count** is not defined. However, you *cannot* change FILE #2 as follows:

```
#include <stdlib.h>

int count;

void f1(void)
{
  count = rand();
}
```

If you declare **count** a second time, many linkers will report a duplicate-symbol error, which means that **count** is defined twice, and the linker doesn't know which to use.

The solution to this problem is C's **extern** specifier. By placing **extern** in front of **count**'s declaration in FILE #2, you are telling the compiler that **count** is an integer defined elsewhere. In other words, using **extern** informs the compiler about the existence and the type of the variable it precedes, but it does not cause storage for that variable to be allocated. The correct version of FILE #2 is

```
#include <stdlib.h>

extern int count;

void f1(void)
```

```
{
  count = rand();
}
```

Although rarely done, it is not incorrect to use **extern** inside a function to declare a global variable defined elsewhere in the same file. For example, the following is valid:

```
#include <stdio.h>

int count;

int main(void)
{
  extern int count; /* this refers to global count */

  count = 10;
  printf("%d", count);

  return 0;
}
```

The reason you will rarely see this use of **extern** is that it is redundant. Whenever the compiler encounters a variable name not defined by the function as a local variable, it assumes that it is global.

One very important storage-class specifier is **register**. When you specify a **register** variable you are telling the compiler that you want access to that variable to be as fast as possible. In early versions of C, **register** could only be applied to local variables (including formal parameters) of types **int** or **char**, or to a pointer type. It caused the variables to be held in a register of the CPU. (This is how the name **register** came about.) By using a register of the CPU, extremely fast access times are achieved. In modern versions of C, the definition of **register** has been broadened to include all types of variables and the requirement that **register** variables must be held in a CPU register was removed. Instead, the ANSI C standard stipulates that a **register** variable will be stored in such a way as to minimize access time. In practice, however, this means that **register** variables of type **int** and **char** continue to be held in a CPU register—this is still the fastest way to access them.

No matter what storage method is used, only so many variables can be granted the fastest possible access time. For example, the CPU has a limited number of registers. When fast-access locations are exhausted, the compiler is free to make **register** variables into regular variables. For this reason, you must choose carefully which variables you modify with **register**.

One good choice is to make a frequently used variable, such as the variable that controls a loop, into a **register** variable. The more times a variable is accessed, the greater the increase in performance when its access time is decreased. Generally, you can assume that at least two variables per function can be truly optimized for access speed.

Important: Because a **register** variable may be stored in a register of the CPU, it may not have a memory address. This means that you *cannot* use the **&** to find the address of a register variable.

When you use the **static** modifier, you cause the contents of a local variable to be preserved between function calls. Also, unlike normal local variables, which are initialized each time a function is entered, a **static** local variable is initialized only once. For example, take a look at
this program,

```
#include <stdio.h>

void f(void);

int main(void)
{
  int i;

  for(i=0; i<10; i++) f();

  return 0;
}

void f(void)
{
  static int count = 0;

  count++;
  printf("count is %d\n", count);
}
```

which displays the following output:

```
count is 1
count is 2
count is 3
count is 4
count is 5
count is 6
count is 7
count is 8
count is 9
count is 10
```

As you can see, **count** retains its value between function calls. The advantage to using a **static** local variable over a global one is that the **static** local variable is still known to and accessible by only the function in which it is declared.

The **static** modifier may also be used on global variables. When it is, it causes the global variable to be known to and accessible by only the functions in the same file in which it is declared. Not only is a function not declared in the same file as a **static** global variable unable to access that global variable, it does not even know its name. This means that there are no name conflicts if a **static** global variable in one file has the same name as another global variable in a different file of the same program. For example, consider these two fragments, which are parts of the same program:

FILE #1

```
int count;
.
.
.
count = 10;
printf("%d", count);
```

FILE #2

```
static int count;
.
.
.
count = 5;
printf("%d", count);
```

Because **count** is declared as **static** in FILE #2, no name conflicts arise. The **printf( )** statement in FILE #1 displays **10** and the **printf( )** statement in FILE #2 displays **5** because the two **counts** are different variables.

EXAMPLES

1. To get an idea about how much faster access to a **register** variable is, try the following program. It makes use of another of C's standard library functions called **clock( )**, which returns the

number of system clock ticks since the program began execution. It has this prototype:

clock_t clock(void);

It uses the TIME.H header. TIME.H also defines the **clock_t** type, which is more or less the same as **long**. To time an event using **clock( )**, call it immediately before the event you wish to time and save its return value. Next, call it a second time after the event finishes and subtract the starting value from the ending value. This is the approach used by the program to time how long it takes two loops to execute. One set of loops is controlled by a **register** variable, the other is controlled by a non-**register** variable.

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

int i;  /* This will not be transformed into a
           register variable because it is global.*/

int main(void)
{
  register int j;

  int k;
  clock_t start, finish;

  start = clock();
  for(k=0; k<100; k++)
    for(i=0; i<32000; i++) ;
  finish = clock();
  printf("Non-register loop: %ld ticks\n", finish - start);

  start = clock();
  for(k=0; k<100; k++)
    for(j=0; j<32000; j++);
  finish = clock();
  printf("Register loop: %ld ticks\n", finish - start);

  return 0;
}
```

For most compilers, the **register**-controlled loop will execute about twice as fast as the non-**register** controlled loop.

The non-**register** variable is global because, when feasible, virtually all compilers will automatically convert local variables not specified as **register** types into **register** types as an automatic optimization. If you do not see the predicted results, it may mean that the compiler has automatically optimized **i** into a register variable, too. Although you can't declare global variables as **register**, there is nothing that prevents a compiler from optimizing your program to this effect. If you don't see much difference between the two loops, try creating extra global variables prior to **i** so that it will not be automatically optimized.

2. As you know, the compiler can optimize access speed for only a limited number of **register** variables in any one function (perhaps as few as two). However, this does not mean that your program can only have a few **register** variables. Because of the way a C program executes, each function may utilize the maximum number of **register** variables. For example, for the average compiler, all the variables shown in the next program will be optimized for speed:

```
#include <stdio.h>

void f2(void);
void f(void);

int main(void)
{
  register int a, b;
  .
  .
  .
}

void f(void)
{
  register int i, j;
  .
  .
  .
}
```

```
void f2(void)
{
  register int j, k;
   .
   .
   .
}
```

3. Local **static** variables have several uses. One is to allow a function to perform various initializations only once, when it is first called. For example, consider this function:

   ```
   void myfunc(void)
   {
     static int first = 1;

     if(first) {  /* initialize the system */
       rewind(fp);
       a = 0;
       loc = 0;
       fprintf("System Initialized");
       first = 0;
     }
     .
     .
     .
   }
   ```

   Because **first** is **static**, it will hold its value between calls. Thus, the initialization code will be executed only the first time the function is called.

4. Another interesting use for a local **static** variable is to control a recursive function. For example, this program prints the numbers **1** through **9** on the screen:

   ```
   #include <stdio.h>

   void f(void);

   int main(void)
   {
     f();

     return 0;
   }
   ```

```
void f(void)
{
  static int stop=0;

  stop++;

  if(stop==10) return;
  printf("%d ", stop);
  f(); /* recursive call */
}
```

Notice how **stop** is used to prevent a recursive call to **f( )** when it equals 10.

5. Here is another example of using **extern** to allow global data to be accessed by two files:

FILE #1:

```
#include <stdio.h>

char str[80];

void getname(void);

int main(void)
{
  getname();
  printf("Hello %s", str);

  return 0;
}
```

FILE #2:

```
#include <stdio.h>

extern char str[80];

void getname(void)
{
  printf("Enter your first name: ");
  gets(str);
}
```

EXERCISES

1. Assume that your compiler will actually optimize access time of only two **register** variables per function. In this program, which two variables are the best ones to be made into **register** variables?

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

int main(void)
{
  int i, j, k, m;

  do {
    printf("Enter a value: ");
    scanf("%d", &i);

    m = 0;
    for(j=0; j<i; j++)
      for(k=0; k<100; k++)
        m = k + m;
  } while(i>0);

  return 0;
}
```

2. Write a program that contains a function called **sum_it( )** that has this prototype:

```
void sum_it (int value);
```

   Have this function use a local **static** integer variable to maintain and display a running total of the values of the parameters it is called with. For example, if **sum_it( )** is called three times with the values 3, 6, 4, then **sum_it( )** will display 3, 9, and 13.

3. Try the program described in Example 5. Be sure to actually use two files. If you are unsure how to compile and link a program consisting of two files, check your compiler's user manual.

4. What is wrong with this fragment?

```
register int i;
int *p;

p = &i;
```

11.2 USE THE ACCESS MODIFIERS

C includes two type modifiers that affect the way variables are accessed by both your program and the compiler. These modifiers are **const** and **volatile**. This section examines these type modifiers.

If you precede a variable's type with **const**, you prevent that variable from being modified by your program. The variable may be given an initial value, however, through the use of an initialization when it is declared. The compiler is free to locate **const** variables in ROM (read-only memory) in environments that support it. A **const** variable may also have its value changed by hardware-dependent means.

The **const** modifier has a second use. It can prevent a function from modifying the object that a parameter points to. That is, when a pointer parameter is preceded by **const**, no statement in the function can modify the variable pointed to by that parameter.

When you precede a variable's type with **volatile**, you are telling the compiler that the value of the variable may be changed in ways not explicitly defined in the program. For example, a variable's address might be given to an interrupt service routine, and its value changed each time an interrupt occurs. The reason that **volatile** is important is that most C compilers apply complex and sophisticated optimizations to your program to create faster and more efficient executable programs. If the compiler does not know that the contents of a variable may change in ways not explicitly specified by the program, it may not actually examine the contents of the variable each time it is referenced. (Unless it occurs on the left side of an assignment statement, of course.)

EXAMPLES

1. The following short program shows how a **const** variable can be given an initial value and be used in the program, as long as it is not on the left side of an assignment statement.

```
#include <stdio.h>

int main(void)
{
  const int i = 10;
  printf("%d", i); /* this is OK */

  return 0;
}
```

The following program tries to assign **i** another value. This program will not compile because **i** cannot be modified by the program.

```
#include <stdio.h>

int main(void)
{
  const int i = 10;

  i = 20; /* this is wrong */

  printf("%d", i);

  return 0;
}
```

2. The next program shows how a pointer parameter can be declared as **const** to prevent the object it points to from being modified.

```
#include <stdio.h>

void pr_str(const char *p);

int main(void)
{
  char str[80];

  printf("Enter a string: ");
  gets(str);

  pr_str(str);

  return 0;
}

void pr_str(const char *p)
{
  while(*p) putchar(*p++); /* this is ok */
}
```

If you change the program as shown below, it will not compile because this version attempts to alter the string pointed to by **p**.

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

void pr_str(const char *p);

int main(void)
{
  char str[80];

  printf("Enter a string: ");
  gets(str);
  pr_str(str);

  return 0;
}

void pr_str(const char *p)
{
  while(*p) {
    *p = toupper(*p); /* this will not compile */
    putchar(*p++);
  }
}
```

3. Perhaps the most important feature of **const** pointer parameters is that they guarantee that many standard library functions will not modify the variables pointed to by their parameters. For example, here is the actual prototype to **strlen( )** specified by the ANSI standard:

   size_t strlen(const char **str*);

   Since *str* is specified as **const**, the string it points to cannot be changed.

4. While short examples of **volatile** are hard to find, the following fragment gives you the flavor of its use:

```
volatile unsigned u;

give_address_to_some_interrupt(&u);
```

```
for(;;) { /* watch value of u */
  printf("%d", u);
  .
  .
  .
```

In this example, if **u** had not been declared as **volatile**, the compiler could have optimized the repeated calls to **printf( )** in such a way that **u** was not reexamined each time. The use of **volatile** forces the compiler to actually obtain the value of **u** whenever it is used.

EXERCISES

1. One good time to use **const** is when you want to embed a version control number into a program. By using a **const** variable to hold the version, you prevent it from accidentally being changed. Write a short program that illustrates how this can be done. Use 6.01 as the version number.
2. Write your own version of **strcpy( )** called **mystrcpy( )**, which has the prototype

   char *mystrcpy (char **to*, const char **from*);

   The function returns a pointer to *to*. Demonstrate your version of **mystrcpy( )** in a program.
3. On your own, see if you can think of any ways to use **volatile**.

11.3 DEFINE ENUMERATIONS

In C you can define a list of named integer constants called an *enumeration*. These constants can then be used any place an integer can. To define an enumeration, use this general form:

enum *tag-name* { *enumeration list* } *variable-list;*

Either the *tag-name* or the *variable-list* is optional. The *tag-name* is essentially the type name of the enumeration. For example,

```
enum color_type {red, green, yellow} color;
```

Here, an enumeration consisting of the constants **red**, **green**, and **yellow** is created. The enumeration tag is **color_type** and one variable, called **color**, has been created.

By default, the compiler assigns integer values to enumeration constants, beginning with 0 at the far left side of the list. Each constant to the right is one greater than the constant that precedes it. Therefore, in the color enumeration, **red** is 0, **green** is 1, and **yellow** is 2. However, you can override the compiler's default values by explicitly giving a constant a value. For example, in this statement

```
enum color_type {red, green=9, yellow} color;
```

red is still 0, but **green** is 9, and **yellow** is 10.

Once you have defined an enumeration, you can use its tag name to declare enumeration variables at other points in the program. For example, assuming the **color_type** enumeration, this statement is perfectly valid and declares **mycolor** as a **color_type** variable:

```
enum color_type mycolor;
```

An enumeration is essentially an integer type and an enumeration variable can hold any integer value—not just those defined by the enumeration. But for clarity and structure, you should use enumeration variables to hold only values that are defined by their enumeration type.

Two of the main uses of an enumeration are to help provide self-documenting code and to clarify the structure of your program.

EXAMPLES

1. This short program creates an enumeration consisting of the parts of a computer. It assigns **comp** the value **CPU** and then displays its value (which is 1). Notice how the enumeration tag name is used to declare **comp** as an enumeration variable separately from the actual declaration of **computer**.

```
#include <stdio.h>

enum computer {keyboard, CPU, screen, printer};

int main(void)
{
  enum computer comp;
  comp = CPU;

  printf("%d", comp);

  return 0;
}
```

2. It takes a little work to display the string equivalent of an enumerated constant. Remember, enumerated constants are not strings; they are named integer constants. The following program uses a **switch** statement to output the string equivalent of an enumerated value. The program uses C's random-number generator to choose a means of transportation. It then displays the means on the screen. (This program is for people who can't make up their minds!)

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

enum transport {car, train, airplane, bus} tp;

int main(void)
{
  printf("Press a key to select transport: ");

  /* generate a new random number each time
     the program is run
  */
  while(!kbhit()) rand();
  getch(); /* read and discard character */

  tp = rand() % 4;
  switch(tp) {
    case car: printf("car");
      break;
    case train: printf("train");
      break;
    case airplane: printf("airplane");
      break;
    case bus: printf("bus");
  }

  return 0;
}
```

In some cases, there is an easier way to obtain a string equivalent of an enumerated value. As long as you do not initialize any of the constants, you can create a two-dimensional string array that contains the string equivalents of the enumerated values in the same order that the constants appear in the enumeration. You can then index the array using an enumeration value to obtain its corresponding string. The following version of the transportation-choosing program, for example, uses this approach:

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

enum transport {car, train, airplane, bus} tp;

char trans[][20] = {
  "car", "train", "airplane", "bus"
};

int main(void)
{
  printf("Press a key to select transport: ");

  /* Generate a new random number each time
     the program is run
  */
  while(!kbhit()) rand();
  getch(); /* read and discard character */

  tp = rand() % 4;
  printf("%s", trans[tp]);

  return 0;
}
```

3. Remember, the names of enumerated constants are known only to the program, not to any library functions. For example, given the fragment

```
enum numbers {zero, one, two, three} num;

printf("Enter a number: ");
scanf("%d", &num);
```

you cannot respond to **scanf( )** by entering **one**.

EXERCISES

1. Compile and run the example programs.
2. Create an enumeration of the coins of the U.S. from penny to dollar.
3. Is this fragment correct? If not, why not?

```
enum cars {Ford, Chrysler, GM} make;

make = GM;
printf("car is %s", make);
```

UNDERSTAND typedef

In C you can create a new name for an existing type using **typedef**. The general form of **typedef** is

typedef *old-name new-name;*

This new name can be used to declare variables. For example, in the following program, **smallint** is a new name for a **signed char** and is used to declare **i**.

```
#include <stdio.h>

typedef signed char smallint;

int main(void)
{
  smallint i;

  for(i=0; i<10; i++)
    printf("%d ", i);

  return 0;
}
```

Keep two points firmly in mind: First, a **typedef** does not cause the original name to be deactivated. For example, in the program, **signed char** is still a valid type. Second, you can use several **typedef** statements to create many different, new names for the same type.

There are basically two reasons to use **typedef**. The first is to create portable programs. For example, if you know that you will be writing a program that will be executed on computers using 16-bit integers as well as on computers using 32-bit integers, and you want to ensure that certain variables are 16 bits long in both environments, you might want to use a **typedef** when compiling the program for the 16-bit machines as follows:

```
typedef int myint;
```

Then, before compiling the code for a 32-bit computer, you can change the **typedef** statement like this:

```
typedef short int myint;
```

This works because on computers using 32-bit integers, a **short int** will be 16 bits long. Assuming that you used **myint** to declare all integer values that you wanted to be 16 bits long, you need change only one statement to change the type of all variables declared using **myint**.

The second reason you might want to use **typedef** is to help provide self-documenting code. For example, if you are writing an inventory program, you might use this **typedef** statement.

```
typedef double subtotal;
```

Now, when anyone reading your program sees a variable declared as **subtotal**, he or she will know that it is used to hold a subtotal.

EXAMPLES

1. The new name created by one **typedef** can be used in a subsequent **typedef** to create another name. For example, consider this fragment:

   ```
   typedef int height;
   typedef height length;
   typedef length depth;

   depth d;
   ```

   Here, **d** is still an integer.

2. In addition to the the basic types, you can use **typedef** on more complicated types. For example, the following is perfectly valid:

```
enum e_type {one, two, three } ;

typedef enum e_type mynums;

mynums num; /* declare a variable */
```

Here, **num** is a variable of type **e_type**.

EXERCISES

1. Show how to make **UL** a new name for **unsigned long**. Show that it works by writing a short program that declares a variable using **UL**, assigns it a value, and displays the value on the screen.
2. What is wrong with this fragment?

   ```
   typedef balance float;
   ```

USE C'S BITWISE OPERATORS

C contains four special operators that perform their operations on a bit-by-bit level. These operators are

&	bitwise AND
|	bitwise OR
^	bitwise XOR (eXclusive OR)
~	1's complement

These operators work with character and integer types; they cannot be used with floating-point types.

The AND, OR, and XOR operators produce a result based on a comparison of corresponding bits in each operand. The AND operator sets a bit if both bits being compared are set. The OR sets a bit if either of the bits being compared is set. The XOR operation sets a bit when either of the two bits involved is 1, but not when both are 1 or both are 0. Here is an example of a bitwise AND:

```
  1010 0110
& 0011 1011
  ---------
  0010 0010
```

Notice how the resulting bit is set, based on the outcome of the operation being applied to the corresponding bits in each operand.

The 1's complement operator is a unary operator that reverses the state of each bit within an integer or character.

EXAMPLES

1. The XOR operation has one interesting property. Given two values A and B, when the outcome of A XOR B is XORed with B a second time, A is produced. For example, this output

   ```
   initial value of i: 100
   i after first XOR: 21895
   i after second XOR: 100
   ```

   is produced by the following program:

   ```
   #include <stdio.h>

   int main(void)
   {
     int i;

     i = 100;
     printf("initial value of i: %d\n", i);

     i = i ^ 21987;
     printf("i after first XOR: %d\n", i);

     i = i ^ 21987;
     printf("i after second XOR: %d\n", i);

     return 0;
   }
   ```

2. The following program uses a bitwise AND to display, in binary, the ASCII value of a character typed at the keyboard:

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

```
int main(void)
{
  char ch;
  int i;

  printf("Enter a character: ");
  ch = getche();
  printf("\n");

  /* display binary representation */
  for(i=128; i>0; i=i/2)
    if(i & ch) printf("1 ");
    else printf("0 ");

  return 0;
}
```

The program works by adjusting the value of **i** so that only one bit is set each time a comparison is made. Since the high-order bit in a byte represents 128, this value is used as a starting point. Each time through the loop, **i** is halved. This causes the next bit position to be set and all others cleared. Thus, each time through the loop, a bit in **ch** is tested. If it is 1, the comparison produces a true result and a **1** is output. Otherwise a **0** is displayed. This process continues until all bits have been tested.

3. By modifying the program from Example 2, it can be used to show the effect of the 1's complement operator.

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

int main(void)
{
  char ch;
  int i;

  ch = 'a';

  /* display binary representation */
  for(i=128; i>0; i=i/2)
    if(i & ch) printf("1 ");
    else printf("0 ");

  /* reverse bit pattern */
  ch = ~ch;
  printf("\n");

  /* display binary representation */
  for(i=128; i>0; i=i/2)
    if(i & ch) printf("1 ");
    else printf("0 ");

  return 0;
}
```

When you run this program, you will see that the state of bits in **ch** are reversed after the ~ operation has occurred.

4. The following program shows how to use the *&* operator to determine if a signed integer is positive or negative. (The program assumes short integers are 16 bits long.) Since negative numbers are represented with their high-order bit set, the comparison will be true only if **i** is negative. (The value 32768 is the value of an unsigned short integer when only its high-order bit is set. This value is 1000 0000 in binary.)

```
#include <stdio.h>

int main(void)
{
  short i;

  printf("Enter a number: ");
  scanf("%hd", &i);

  if(i & 32768) printf("Number is negative.\n");

  return 0;
}
```

5. The following program makes **i** into a negative number by setting its high-order bit. (Again, 16-bit short integers are assumed.)

```
#include <stdio.h>

int main(void)
{
  short i;
```

```
    i = 1;
    i = i | 32768;
    printf("%hd", i);

    return 0;
  }
```

It displays **-32,767.**

EXERCISES

1. One very easy way to encode a file is to reverse the state of each bit using the ~ operator. Write a program that encodes a file using this method. (To decode the file, simply run the program a second time.) Have the user specify the name of the file on the command line.
2. A better method of coding a file uses the XOR operation combined with a user-defined key. Write a program that encodes a file using this method. Have the user specify the file to code as well as a single character key on the command line. (To decode the file, run the program a second time using the same key.)
3. What is the outcome of these operations?

   A. 1010 0011 & 0101 1101

   B. 0101 1101 | 1111 1011

   C. 0101 0110 ^ 1010 1011
4. Sometimes, the high-order bit of a byte is used as a *parity bit* by modem programs. It is used to verify the integrity of each byte transferred. There are two types of parity: even and odd. If even parity is used, the parity bit is used to ensure that each byte has an even number of 1 bits. If odd parity is used, the parity bit is used to ensure that each byte has an odd number of 1 bits. Since the parity bit is not part of the information being transferred, show how you can clear the high-order bit of a character value.

MASTER THE SHIFT OPERATORS

C includes two operators not commonly found in other computer languages: the left and right bit-shift operators. The left shift operator is <<, and the right shift operator is >>. These operators may be applied only to character or integer operands. They take these general forms:

value << *number-of-bits*

value >> *number-of-bits*

The integer expression specified by *number-of-bits* determines how many places to the left or right the bits within *value* are shifted. Each left-shift causes all bits within the specified value to be shifted left one position and a zero is brought in on the right. A right-shift shifts all bits to the right one position and brings a zero in on the left. (Unless the number is negative, in which case a one is brought in.) When bits are shifted off an end, they are lost.

A right shift is equivalent to dividing a number by 2, and a left shift is the same as multiplying the number by 2. Because of the internal operation of virtually all CPUs, shift operations are usually faster than their equivalent arithmetic operations.

EXAMPLES

1. This program demonstrates the right and left shift operators:

```
#include <stdio.h>

void show_binary(unsigned u);

int main(void)
{
  unsigned short u;

  u = 45678;

  show_binary(u);
  u = u << 1;
  show_binary(u);
  u = u >> 1;
  show_binary(u);
  return 0;
}

void show_binary(unsigned u)
{
  unsigned n;

  for(n=32768; n>0; n=n/2)
    if(u & n) printf("1 ");
    else printf("0 ");

  printf("\n");
}
```

The output from this program is

```
1 0 1 1 0 0 1 0 0 1 1 0 1 1 1 0
0 1 1 0 0 1 0 0 1 1 0 1 1 1 0 0
0 0 1 1 0 0 1 0 0 1 1 0 1 1 1 0
```

Notice that after the left shift, a bit of information has been lost. When the right shift occurs, a zero is brought in. As stated earlier, bits that are shifted off one end are lost.

2. Since a right shift is the same as a division by two, but faster, the **show_binary( )** function can be made more efficient as shown here:

```
void show_binary(unsigned u)
{
  unsigned n;

  for(n=32768; n; n=n>>1)
    if(u & n) printf("1 ");
    else printf("0 ");

  printf("\n");
}
```

EXERCISES

1. Write a program that uses the shift operators to multiply and divide an integer. Have the user enter the initial value. Display the result of each operation.
2. C does not have a rotate operator. A *rotate* is similar to a shift, except that the bit shifted off one end is inserted onto the other. For example, 1010 0000 rotated left one place is 0100 0001. Write a function called **rotate( )** that rotates a byte left one position each time it is called. (Hint, you will need to use a union so that you can have access to the bit shifted off the end of the byte.) Demonstrate the function in a program.

UNDERSTAND THE ? OPERATOR

C contains one ternary operator: the ?. A *ternary operator* requires three operands. The ? operator is used to replace statements such as:

```
if(condition) var = exp1;
else var = exp2;
```

The general form of the ? operator is

var = condition ? exp1: exp2 ;

Here, *condition* is an expression that evaluates to true or false. If it is true, *var* is assigned the value of *exp1*. If it is false, *var* is assigned the value of *exp2*. The reason for the ? operator is that a C compiler can produce more efficient code using it instead of the equivalent **if/else** statement.

EXAMPLES

1. The following program illustrates the ? operator. It inputs a number and then converts the number into 1 if the number is positive and -1 if it is negative.

```
#include <stdio.h>

int main(void)
{
  int i;

  printf("Enter a number: ");
  scanf("%d", &i);

  i = i>0 ? 1: -1;

  printf("Outcome: %d", i);

  return 0;
}
```

2. The next program is a computerized coin toss. It waits for you to press a key and then prints either **Heads** or **Tails**.

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

int main(void)
{
  int i;

  while(!kbhit()) rand();

  i = rand() %2 ? 1: 0;

  if(i) printf("Heads");
  else printf("Tails");

  return 0;
}
```

The coin-toss program can be written in a more efficient way. There is no technical reason that the **?** operator need assign its value to any variable. Therefore, the coin toss program can be written as:

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

```
int main(void)
{
  while(!kbhit()) rand();

  rand()%2 ? printf("Heads") : printf("Tails");

  return 0;
}
```

Remember, since a call to a function is a valid C expression, it is perfectly valid to call **printf( )** in the **?** statement.

EXERCISES

1. One particularly good use for the **?** operator is to provide a means of preventing a division-by-zero error. Write a program that inputs two integers from the user and displays the result of dividing the first by the second. Use **?** to avoid division by zero.
2. Convert the following statement into its equivalent **?** statement.

```
if(a>b) count = 100;
else count = 0;
```

DO MORE WITH THE ASSIGNMENT OPERATOR

The assignment operator is more powerful in C than in most other computer languages. In this section, you will learn some new things about it.

You can assign several variables the same value using the general form

var1 = var2 = var3 = ... = varN = value;

For example, this statement

```
i = j = k = 100;
```

assigns **i**, **j**, and **k** the value 100. In professionally written C code, it is common to see such multiple-variable assignments.

Another variation on the assignment statement is sometimes called C *shorthand*. In C, you can transform a statement like

```
a = a + 3;
```

into a statement like

```
a += 3;
```

In general, any time you have a statement of the form

var = var *op* expression;

you can write it in shorthand form as

var *op*= expression;

Here, *op* is one of the following operators.

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

There must be no space between the operator and the equal sign. The reason you will want to use the shorthand form is not that it saves you a little typing effort, but because the C compiler can create more efficient executable code.

EXAMPLES

1. The following program illustrates the multiple-assignment statement:

```
#include <stdio.h>

int main(void)
{
  int i, j, k;

  i = j = k = 99;

  printf("%d %d %d", i, j, k);

  return 0;
}
```

2. The next program counts to 98 by twos. Notice that it uses C shorthand to increment the loop-control variable by two each iteration.

```
#include <stdio.h>

int main(void)
{
  int i;

  /* count by 2s */
  for(i=0; i<100; i+=2)
    printf("%d ", i);

  return 0;
}
```

3. The following program uses the left-shift operator in shorthand form to multiply the value of **i** by 2, three times. (The resulting value is 8.)

```
#include <stdio.h>

int main(void)
{
  int i = 1;

  i <<= 3;  /* multiply by 2, 3 times */

  printf("%d", i);

  return 0;
```

EXERCISES

1. Compile and run the program in Example 1 to prove to yourself that the multiple-assignment statement works.
2. How is the following statement written using C shorthand?

```
x = x & y;
```

3. Write a program that displays all the even multiples of 17 from 17 to 1000. Use C shorthand.

11.9 UNDERSTAND THE COMMA OPERATOR

The last operator we will examine is the comma. It has a very unique function: it tells the compiler to "do this and this and this." That is, the comma is used to string together several operations. The most common use of the comma is in the **for** loop. In the following loop, the comma is used in the initialization portion to initialize two loop-control variables, and in the increment portion to increment **i** and **j**.

```
for(i=0, j=0; i+j<count; i++, j++) . . .
```

The value of a comma-separated list of expressions is the rightmost expression. For example, the following statement assigns 100 to **value**:

```
value = (count, 99, 33, 100);
```

The parentheses are necessary because the comma operator is lower in precedence than the assignment operator.

EXAMPLES

1. This program displays the numbers 0 through 49. It uses the comma operator to maintain two loop-control variables.

```
#include <stdio.h>

int main(void)
{
  int i, j;

  /* count to 49 */
  for(i=0, j=100; i<j; i++, j--)
    printf("%d ", i);

  return 0;
```

2. In many places in C, it is actually syntactically correct to use the comma in place of the semicolon. For example, examine the following short program:

```
#include <stdio.h>

int main(void)
{
  char ch;

  ch = getchar(), /* notice the comma here */
  putchar(ch+1);

  return 0;
}
```

   Because the comma tells the compiler to "do this and this," the program runs the same with the comma after **getchar( )** as it would had a semicolon been used. Using a comma in this way is considered extremely bad form, however. It is possible that an unwanted side effect could occur. (This use of the comma operator *does* make interesting coffee-break conversation, however! Many C programmers are not aware of this interesting twist in the C syntax.)

EXERCISES

1. Write a program that uses the comma operator to maintain three **for** loop-control variables. Have one variable run from 0 to 99, the second run from –50 to 49, and have the third set to the sum of the first two, both initially and each time the loop iterates. Have the loop stop when the first variable reaches 100. Have the program display the value of the third variable each time the loop repeats.
2. What is the value of **i** after the following statement executes?

```
i = (1, 2, 3);
```

11.10 KNOW THE PRECEDENCE SUMMARY

The following table shows the precedence of all the C operators.

Highest	`( ) [ ] -> .`
	`! ~ + - ++ -- (type cast) * & sizeof`
	`* / %`
	`+ -`
	`<< >>`
	`< <= > >=`
	`== !=`
	`&`
	`^`
	`|`
	`&&`
	`||`
	`?:`
	`= += -= *= /= etc.`
Lowest	`,`

Mastery Skills Check

At this point you should be able to answer these questions and perform these exercises:

1. What does the **register** specifier do?
2. What do the **const** and **volatile** modifiers do?
3. Write a program that sums the numbers 1 to 100. Make the program execute as fast as possible.
4. Is this statement valid? If so, what does it do?

   ```
   typedef long double bigfloat;
   ```

5. Write a program that inputs two characters and compares corresponding bits. Have the program display the number of each bit in which a match occurs. For example, if the two integers are

1001 0110
1110 1010

the program will report that bits 7, 1, and 0 match. (Use the bitwise operators to solve this problem.)

6. What do the << and >> operators do?
7. Show how this statement can be rewritten:

```
c = c + 10;
```

8. Rewrite this statement using the ? operator:

```
if(!done) count
else count = 0;
```

9. What is an enumeration? Show an example that enumerates the planets.

Cumulative Skills Check

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. Write a program that swaps the low-order four bits of a byte with the high-order four bits. Demonstrate that your routine works by displaying the contents of the byte before and after, using the **show_binary( )** function developed earlier. (Change **show_binary( )** so that it works on an eight-bit quantity, however.)
2. Earlier you wrote a program that encoded files using the 1's complement operator. Write a program that reads a text file encoded using this method and displays its decoded contents. Leave the actual file encoded, however.
3. Is this fragment correct?

```
register FILE *fp;
```

4. Using the program you developed for Chapter 10, Section 10.3, Exercise 1, optimize the program by selecting appropriate local variables to become **register** types.

12

The C Preprocessor and Some Advanced Topics

chapter objectives

12.1 Learn more about **#define** and **#include**

12.2 Understand conditional compilation

12.3 Learn about **#error**, **#undef**, **#line**, and **#pragma**

12.4 Examine C's built-in macros

12.5 Use the **#** and **##** operators

12.6 Understand function pointers

12.7 Master dynamic allocation

CONGRATULATIONS! If you have worked your way through all the preceding chapters, you can definitely call yourself a C programmer. This chapter examines three topics: the C preprocessor, pointers to functions, and C's dynamic allocation system. All of the features discussed in this chapter are important, and you need to be aware of their existence. However, you won't use many of them right away. This is not because any of the features discussed in this chapter are particularly difficult, but because some features are more applicable to large programming efforts and the management of sophisticated systems. As your proficiency in C increases, however, you will find these features quite valuable.

Review Skills Check

Before proceeding you should be able to answer these questions and perform these exercises:

1. What is the major advantage gained when a variable is declared using **register**?
2. What is wrong with this function?

```
void myfunc(const int *i)
{
  *i = *i / 2;
}
```

3. What is the outcome of these operations?
   a. 1101 1101 & 1110 0110
   b. 1101 1101 | 1110 0110
   c. 1101 1101 ^ 1110 0110
4. Write a program that uses the left and right shift operators to double and halve a number entered by the user.
5. How can these statements be written differently?

```
a = 1;
b = 1;
c = 1;
```

```
  if(a<b) max = 100;
  else max = 0;

  i = i * 2;
```

6. What is the **extern** type specifier for?

12.1 LEARN MORE ABOUT #define AND #include

Although you have been using **#define** and **#include** for some time, both have more features than you've read about so far. Each is examined here in detail.

In addition to using **#define** to define a macro name that will be substituted by the character sequence associated with that macro, you can use **#define** to create *function-like macros*. In a function-like macro, arguments can be passed to the macro when it is expanded by the preprocessor. For example, consider this program:

```
#include <stdio.h>

#define SUM(i, j) i+j

int main(void)
{
  int sum;

  sum = SUM(10, 20);
  printf("%d", sum);

  return 0;
}
```

The line

```
sum = SUM(10, 20);
```

is transformed into

```
sum = 10+20;
```

by the preprocessor. As you can see, the values 10 and 20 are automatically substituted for the parameters **i** and **j**.

A more practical example is **RANGE( )**, illustrated in the following simple program. It is used to confirm that parameter **i** is within the range specified by parameters **min** and **max**. You can imagine how useful a macro like **RANGE( )** can be in programs that must perform several range checks. This program uses it to display random numbers between 1 and 100.

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

#define RANGE(i, min, max) (i<min) || (i>max) ? 1 : 0

int main(void)
{
  int r;

  /* print random numbers between 1 and 100 */
  do {
    do {
      r = rand();
    } while(RANGE(r, 1, 100));
    printf("%d ", r);
  } while(!kbhit());

  return 0;
}
```

The advantage to using function-like macros instead of functions is that in-line code is generated by the macro, thus avoiding the time it takes to call and return from a function. Of course, only relatively simple operations can be made into function-like macros. Also, because code is duplicated, the resulting program might be longer than it would be if a function were used.

The **#include** directive has these two general forms:

#include <*filename*>
#include "*filename*"

So far, all the example programs have used the first form. The reason for this will become apparent after you read the following descriptions.

If you specify the file name between angle brackets, you are instructing the compiler to search for the file in some implementation-defined manner. For most compilers, this means searching a special directory devoted to the standard header files. This is why the sample programs have been using this form to include the header files required by the standard library functions. If you enclose the file name between quotation marks, the compiler searches for the file in another implementation-defined manner. If that search fails, the search is restarted as if you had specified the file name between angle brackets. For the majority of compilers, enclosing the name between quotation marks causes the current working directory to be searched first. Typically, you will use quotation marks to include header files that you create.

EXAMPLES

1. Here is a program that uses the function-like macro **MAX( )** to compute which argument is larger. Pay close attention to the last **printf( )** statement.

```
#include <stdio.h>

#define MAX(i, j) i>j ? i : j

int main(void)
{
  printf("%d\n", MAX(1, 2));
  printf("%d\n", MAX(1, -1));

  /* this statement does not work correctly */
  printf("%d\n", MAX(100 && -1, 0));

  return 0;
}
```

When the preprocessor expands the final **printf( )** statement, the **MAX( )** macro is transformed into this expression:

```
100 && -1 > 0 ? 100 && -1 : 0
```

Because of C's precedence rules, however, this expression is executed as if parentheses had been added like this:

```
100 && (-1 > 0) ? 100 && -1 : 0
```

As you can see, this causes the wrong answer to be computed. To fix this problem, the macro needs to be rewritten as:

```
#define MAX(i, j) ((i)>(j)) ? (i) : (j)
```

Now the macro works in all possible situations. In general, you will need to fully parenthesize all parameters to a function-like macro.

*The **RANGE( )** macro discussed earlier will need similar parenthesization as well if it is to work in all possible situations. This is left as an exercise.*

2. The next program uses quotes in the **#include** directive.

```
#include "stdio.h"

int main(void)
{
  printf("This is a test");

  return 0;
}
```

While not as efficient as using the angle brackets, the **#include** statement will still find and include the STDIO.H header file.

3. It is permissible to use both forms of the **#include** directive in the same program. For example,

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

int main(void)
{
  printf("This is a random number: %d", rand());

  return 0;
}
```

EXERCISES

1. Correct the **RANGE( )** macro by adding parentheses in the proper locations.
2. Write a program that uses a parameterized macro to compute the absolute value of an integer, and demonstrate its use in a program.
3. Compile Example 2. If your compiler does not find STDIO.H, recheck the installation instructions that came with your compiler.

12.2 UNDERSTAND CONDITIONAL COMPILATION

The C preprocessor includes several directives that allow parts of the source code of a program to be selectively compiled. This is called *conditional compilation*. These directives are

```
#if
#else
#elif
#endif
#ifdef
#ifndef
```

This section examines these directives.

The general form of **#if** is shown here:

```
#if constant-expression
  statement-sequence
#endif
```

If the value of the *constant-expression* is true, the statement or statements between **#if** and **#endif** are compiled. If the *constant-expression* is false, the compiler skips the statement or statements. Keep in mind that the preprocessing stage is the first stage of compilation, so the *constant-expression* means exactly that. No variables may be used.

You can use the **#else** to form an alternative to the **#if**. Its general form is shown here:

```
#if constant-expression
  statement-sequence
#else
  statement-sequence
#endif
```

Notice that there is only one **#endif**. The **#else** automatically terminates the **#if** block of statements. If the *constant-expression* is false, the statement or statements associated with the **#else** are compiled.

You can create an if-else-if ladder using the **#elif** directive, as shown here:

```
#if constant-expression-1
  statement-sequence
#elif constant-expression-2
  statement-sequence
#elif constant-expression-3
  statement-sequence
.
.
.
#endif
```

As soon as the first expression is true, the lines of code associated with that expression are compiled, and the rest of the code is skipped.

Another approach to conditional compilation is the **#ifdef** directive. It has this general form:

```
#ifdef macro-name
 statement-sequence
#endif
```

If the *macro-name* is currently defined, then the *statement-sequence* associated with the **#ifdef** directive will be compiled. Otherwise, it is skipped. The **#else** may also be used with **#ifdef** to provide an alternative.

The complement of **#ifdef** is **#ifndef**. It has the same general form as **#ifdef**. The only difference is that the statement sequence associated with an **#ifndef** directive is compiled only if the *macro-name* is *not* defined.

In addition to **#ifdef**, there is a second way to determine if a macro name is defined. You can use the **#if** directive in conjunction with the **defined** compile-time operator. The **defined** operator has this general form:

defined *macro-name*

If *macro-name* is defined, then the outcome is true. Otherwise, it is false. For example, the following two preprocessor directives are equivalent:

```
#ifdef WIN32
#if defined WIN32
```

You can also apply the **!** operator to **defined** to reverse the condition.

EXAMPLES

1. Sometimes you will want a program's behavior to depend on a value defined within the program. Although examples that are both short and meaningful are hard to find, the following program gives the flavor of it. This program can be compiled to display either the ASCII character set by itself, or the full extended set, depending on the value of **CHAR_SET**. As you know, the ASCII character set defines characters for the values 0 through 127. However, most computers reserve the values 128 through 255 for foreign-language characters and mathematical and other special symbols. (You might want to try this program with **CHAR_SET** set to 256. You will see some very interesting characters!)

   ```
   #include <stdio.h>

   /* define CHAR_SET as either 256 or 128 */
   ```

```
#define CHAR_SET 256

int main(void)
{
  int i;
#if CHAR_SET ==256
  printf("Displaying ASCII character set plus extensions.\n");
#else
  printf("Displaying only ASCII character set.\n");
#endif

  for(i=0; i<CHAR_SET; i++)
    printf("%c", i);

  return 0;
}
```

2. A good use of **#ifdef** is for imbedding debugging information into your programs. For example, here is a program that copies the contents of one file into another:

```
/* Copy a file. */
#include <stdio.h>
#include <stdlib.h>

#define DEBUG

int main(int argc, char *argv[])
{
  FILE *from, *to;
  char ch;

  /* see if correct number of command line arguments */
  if(argc!=3) {
    printf("Usage: copy <source> <destination>\n");
    exit(1);
  }

  /* open source file */
  if((from = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  /*open destination file */
  if((to = fopen (argv[2], "wb")) ==NULL) {
    printf("Cannot open destination file.\n");
    exit(1);
  }

  /* copy the file */
  while(!feof(from)) {
    ch = fgetc(from);
    if(ferror(from)) {
      printf("Error reading source file.\n");
      exit(1);
    }
    if(!feof(from)) {
      fputc(ch, to);
#ifdef DEBUG
      putchar(ch);
#endif
    }
    if(ferror(to)) {
      printf("Error writing destination file.\n");
      exit(1);
    }
  }
  fclose(from);
  fclose(to);

  return 0;
}
```

If **DEBUG** is defined, the program displays each byte as it is transferred. This can be helpful during the development phase. Once the program is finished, the statement defining **DEBUG** is removed, and the output is not displayed. However, if the program ever misbehaves in the future, **DEBUG** can be defined again, and output will again be shown on the screen. While this might seem like a lot of work for such a simple program, in actual practice programs may have many debugging statements, and this procedure can greatly facilitate the development and testing cycle.

Note *As shown in this program, to simply define a macro name, you do not have to associate any character sequence with it.*

3. Continuing with the debugging theme, it is possible to use the **#if** to allow several levels of debugging code to be easily managed. For example, here is one of the encryption programs from the answers to Chapter 11 that supports three debugging levels:

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

/* DEBUG levels:
          0: no debug
          1: display byte read from source file
          2. display byte written to destination file
          3: display bytes read and bytes written
*/
#define DEBUG 2

int main(int argc, char *argv[])
{
  FILE *in, *out;
  unsigned char ch;

  /* see if correct number of command line arguments */
  if(argc!=4) {
    printf("Usage: code <in> <out> <key>");
    exit(1);
  }

  /* open input file */
  if((in = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open input file.\n");
    exit(1);
  }

  /* open output file */
  if((out = fopen(argv[2], "wb"))==NULL) {
    printf("Cannot open output file.\n");
    exit(1);
  }

  while(!feof(in)) {
    ch = fgetc(in);
#if DEBUG == 1 || DEBUG == 3
    putchar(ch);
#endif
    ch = *argv[3] ^ ch;
#if DEBUG >= 2
    putchar(ch);
#endif
    if(!feof(in)) fputc(ch, out);
  }

  fclose(in);
  fclose(out);

  return 0;
}
```

4. The following fragment illustrates the **#elif**. It displays **NUM is 2** on the screen.

```
#define NUM 2
.
.
.
#if NUM == 1
  printf("NUM is 1");
#elif NUM == 2
  printf("NUM is 2");
#elif NUM == 3
  printf("NUM is 3");
#elif NUM == 4
  printf("NUM is 4");
#endif
```

5. Here, the **defined** operator is used to determine if **TESTPROJECT** is defined.

```
#include <stdio.h>

#define TESTPROJECT 29

#if defined TESTPROJECT
int main(void)
{
  printf("This is a test.\n");
  return 0;
}
#endif
```

EXERCISES

1. Write a program that defines three macros called **INT**, **FLOAT**, and **PWR_TYPE**. Define **INT** as 0, **FLOAT** as 1, and **PWR_TYPE** as either **INT** or **FLOAT**. Have the program request two numbers from the user and display the result of the first number raised to the second number. Using **#if** and depending upon the value of **PWR_TYPE**, have both numbers be integers, or allow the first number to be a **double**.
2. Is this fragment correct? If not, show one way to fix it.

```
#define MIKE

#ifdef !MIKE
.
.
.
#endif
```

LEARN ABOUT #error, #undef, #line, AND #pragma

C's preprocessor supports four special-use directives: **#error**, **#undef**, **#line**, and **#pragma**. Each will be examined in turn here.

The **#error** directive has this general form:

#error *error-message*

It causes the compiler to stop compilation and issue the *error-message* along with other implementation-specific information, which will generally include the number of the line the **#error** directive is in and the name of the file. Note that the *error-message* is not enclosed between quotes. The principal use of the **#error** directive is in debugging.

The **#undef** directive undefines a macro name. Its general form is

#undef *macro-name*

If the *macro-name* is currently undefined, **#undef** has no effect. The principal use for **#undef** is to localize macro names.

When a C compiler compiles a source file, it maintains two pieces of information: the number of the line currently being compiled and the name of the source file currently being compiled. The **#line** directive is used to change these values. Its general form is

#line *line-num* "*filename*"

Here, *line-num* becomes the number of the next line of source code, and *filename* becomes the name the compiler will associate with the source file. The value of *line-num* must be between 1 and 32,767. The *filename* may be a string consisting of any valid file name. The principal use for **#line** is for debugging and for managing large projects.

The **#pragma** directive allows a compiler's implementor to define other preprocessing instructions to be given to the compiler. It has this general form:

#pragma *instructions*

If a compiler encounters a **#pragma** statement that it does not recognize, it ignores it. Whether your compiler supports any **#pragmas** depends on how your compiler was implemented.

EXAMPLES

1. This program demonstrates the **#error** directive.

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

```
int main(void)
{
  int i;

  i = 10;
#error This is an error message.
  printf("%d", i); /* this line will not be compiled */

  return 0;
}
```

As soon as the **#error** directive is encountered, compilation stops.

2. The next program demonstrates the **#undef** directive. As the program states, only the first **printf( )** statement is compiled.

```
#include <stdio.h>

#define DOG

int main(void)
{
#ifdef DOG
  printf("DOG is defined.\n");
#endif

#undef DOG

#ifdef DOG
  printf("This line is not compiled.\n");
#endif

  return 0;
}
```

3. The following program demonstrates the **#line** directive. Since virtually all implementations of **#error** display the line number and name of the file, it is used here to verify that **#line** did, in fact, perform its function correctly. (In the next section, you will see how a C program can directly access the line number and file name).

```
#include <stdio.h>

int main(void)
```

```
{
  int i;

/* reset line number to 1000 and file name to
   myprog.c
*/
#line 1000 "myprog.c"
#error Check the line number and file name.

  return 0;
}
```

4. Although the ANSI C standard does not specify any **#pragma** directives, on your own check your compiler's user manual and learn about any supported by your system.

EXERCISE

1. Try the example programs. See how these directives work on your system.

EXAMINE C'S BUILT-IN MACROS

If your C compiler complies with the ANSI C standard, it will have at least five predefined macro names that your program may use. They are

__LINE__
__FILE__
__DATE__
__TIME__
__STDC__

Each of these is explained here.

The __**LINE**__ macro defines an integer value that is equivalent to the line number of the source line currently being compiled.

The __**FILE**__ macro defines a string that is the name of the file currently being compiled.

The __**DATE**__ macro defines a string that holds the current system date. The string has this general form:

month/day/year

The __**TIME**__ macro defines a string that contains the time the compilation of a program began. The string has this general form:

hours:minutes:seconds

The __**STDC**__ macro is defined as the value 1 if the compiler conforms to the ANSI standard.

EXAMPLES

1. This program demonstrates the macros __**LINE**__, __**FILE**__, __**DATE**__, and __**TIME**__.

```
#include <stdio.h>

int main(void)
{
  printf("Compiling %s, line: %d, on %s, at %s",
            __FILE__, __LINE__, __DATE__,
            __TIME__);

  return 0;
}
```

It is important to understand that the values of the macros are fixed at compile time. For example, if the above program is called T.C, and it is compiled on March 18, 1997, at 10 A.M., it will always display this output no matter when the program is run.

```
Compiling T.C. line: 6, on Mar 18 1997, at 10:00:00
```

The main use of these macros is to create a *time and date stamp*, which shows when the program was compiled.

2. As you learned in the previous section, you can use the **#line** directive to change the number of the current line of source code and the name of the file. When you do this, you are actually changing the values of **__LINE__** and **__FILE__**. For example, this program sets **__LINE__** to **100** and **__FILE__** to **myprog.c**:

```
#include <stdio.h>

int main(void)
{
#line 100 "myprog.c"
  printf("Compiling %s, line: %d, on %s, at %s",
            __FILE__, __LINE__, __DATE__,
            __TIME__);

  return 0;
}
```

The program displays the following output, assuming it was compiled on March 18, 1997, at 10 A.M.

```
Compiling myprog.c, line: 101, on Mar 18 1997, at 10:00:00
```

EXERCISE

1. Compile and run the example programs.

USE THE # AND ## OPERATORS

The C preprocessor contains two little-used but potentially valuable operators: **#** and **##**. The **#** operator turns the argument of a function-like macro into a quoted string. The **##** operator concatenates two identifiers.

EXAMPLES

1. This program demonstrates the **#** operator.

   ```
   #include <stdio.h>

   #define MKSTRING(str) # str

   int main(void)
   {
     int value;

     value = 10;

     printf("%s is %d", MKSTRING(value), value);

     return 0;
   }
   ```

   The program displays **value is 10**. This output occurs because **MKSTRING( )** causes the identifier **value** to be made into a quoted string.

2. The following program demonstrates the **##** operator. It creates the **output( )** macro, which translates into a call to **printf( )**. The value of two variables, which end in 1 or 2, is displayed.

   ```
   #include <stdio.h>

   #define output(i) printf("%d %d\n", i ## 1, i ## 2)

   int main(void)
   {
     int count1, count2;
     int i1, i2;

     count1 = 10;
     count2 = 20;
     i1 = 99;
     i2 = -10;
   ```

```
  output(count);
  output(i);

  return 0;
}
```

The program displays **10 20 99 –10**. In the calls to **output( )**, **count** and **i** are concatenated with 1 and 2 to form the variable names **count1**, **count2**, **i1** and **i2** in the **printf( )** statements.

EXERCISES

1. Compile and run the example programs.
2. What does this program display?

```
#include <stdio.h>

#define JOIN(a, b) a ## b

int main(void)
{
  printf(JOIN("one ", "two"));

  return 0;
}
```

3. On your own, experiment with the **#** and **##** operators. Try to think of ways they can be useful to you in your own programming projects.

UNDERSTAND FUNCTION POINTERS

This section introduces one of C's most important advanced features: the function pointer. Although it is beyond the scope of this book to discuss all the nuances and implications of function pointers, the main issues are covered here.

A *function pointer* is a variable that contains the address of the entry point to a function. When the compiler compiles your program, it creates an entry point for each function in the program. The entry point is the address to which execution transfers when a function is called. Since the entry point has an address, it is possible to have a pointer variable point to it. Once you have a pointer to a function, it is possible to actually call that function using the pointer. You will see shortly why you might want to do this.

To create a variable that can point to a function, declare the pointer as having the same type as the return type of the function, followed by any parameters. For example, the following declares **p** as a pointer to a function that returns an integer and has two integer parameters, **x** and **y**.

```
int (*p) (int x, int y);
```

The parentheses surrounding ***p** are necessary because of C's precedence rules.

To assign the address of a function to a function pointer, simply use its name without any parentheses. For example, assuming that **sum( )** has the prototype

```
int sum(int a, int b);
```

the assignment statement

```
p = sum;
```

is correct. Once this has been done, you can call **sum( )** indirectly through **p** using a statement like

```
result = (*p) (10, 20);
```

Again, because of C's precedence rules, the parentheses are necessary around ***p**. Actually, you can also just use **p** directly, like this:

```
result = p(10, 20);
```

However, the **(*p)** form tips off anyone reading your code that a function pointer is being used to indirectly call a function, instead of calling a function named **p**.

EXAMPLES

1. As a first example, this program fills in the details and demonstrates the function pointer that was just described.

```
#include <stdio.h>

int sum(int a, int b);

int main(void)
{
  int (*p) (int x, int y);
  int result;

  p = sum; /* get address of sum() */

  result = (*p) (10, 20);
  printf("%d", result);

  return 0;
}

int sum(int a, int b)
{
  return a+b;
}
```

   The program prompts the user for two numbers, calls **sum( )** indirectly using **p**, and displays the result.

2. Although the program in Example 1 illustrates the mechanics of using function pointers, it does not even hint at their power. The following example, however, will give you a taste.

   One of the most important uses of function pointers occurs when a function-pointer array is created. Each element in the array can point to a different function. To call any specific function, the array is simply indexed. A function pointer array allows very efficient code to be written when a variety of different functions need to be called under differing circumstances. Function-pointer arrays are typically used when writing systems software, such as compilers, assemblers, and interpreters. However, they are not limited to these applications. While meaningful and short examples of function-pointer arrays are difficult to find, the program shown next gives you an idea of their value. Like the program in Example 1, this program prompts the user for two numbers. Next, it asks the user to enter the number of the operation to perform. This number is then used to index the function-pointer array to execute the proper function. Finally, the result is displayed.

```
#include <stdio.h>

int sum(int a, int b);
int subtract(int a, int b);
int mul(int a, int b);
int div(int a, int b);

int (*p[4]) (int x, int y);

int main(void)
{
  int result;
  int i, j, op;

  p[0] = sum; /* get address of sum() */
  p[1] = subtract; /* get address of subtract() */
  p[2] = mul; /* get address of mul() */
  p[3] = div; /* get address of div() */

  printf("Enter two numbers: ");
  scanf("%d%d", &i, &j);
  printf("0: Add, 1: Subtract, 2: Multiply, 3: Divide\n");
  do {
    printf("Enter number of operation: ");
    scanf("%d", &op);
  } while(op<0 || op>3);

  result = (*p[op]) (i, j);
  printf("%d", result);

  return 0;
}

int sum(int a, int b)
{
  return a+b;
}

int subtract(int a, int b)
```

```
{
  return a-b;
}

int mul(int a, int b)
{
  return a*b;
}

int div(int a, int b)
{
  if(b) return a/b;
  else return 0;
}
```

When you study this code, it becomes clear that using a function-pointer array to call the appropriate function is more efficient than using a **switch( )** statement.

Before leaving this example, we can use it to illustrate one more point: function-pointer arrays can be initialized, just like any other array. The following version of the program shows this.

```
#include <stdio.h>

int sum(int a, int b);
int subtract(int a, int b);
int mul(int a, int b);
int div(int a, int b);

/* initialize the pointer array */
int (*p[4]) (int x, int y) = {
  sum, subtract, mul, div
} ;

int main(void)
{
  int result;
  int i, j, op;

  printf("Enter two numbers: ");
  scanf("%d%d", &i, &j);
  printf("0: Add, 1: Subtract, 2: Multiply, 3: Divide\n");
  do {
    printf("Enter number of operation: ");
    scanf("%d", &op);
  } while(op<0 || op>3);

  result = (*p[op]) (i, j);
  printf("%d", result);

  return 0;
}

int sum(int a, int b)
{
  return a+b;
}

int subtract(int a, int b)
{
  return a-b;
}

int mul(int a, int b)
{
  return a*b;
}

int div(int a, int b)
{
  if(b) return a/b;
  else return 0;
}
```

3. One of the most common uses of a function pointer occurs when utilizing another of C's standard library functions, **qsort( )**. The **qsort( )** function is a generic sort routine that can sort any type of singly dimensioned array, using the Quicksort algorithm. Its prototype is

   void qsort(void **array*, size_t *number*, size_t *size*,
   int (**comp*)(const void **a*, const void **b*));

   Here, *array* is a pointer to the first element in the array to be sorted. The number of elements in the array is specified by *number*, and the size of each element of the array is specified by *size*. (Remember, **size_t** is defined by the C compiler and is loosely the same as **unsigned**.) The final parameter is a pointer to a function (which you create) that compares two elements of the array and returns the following results:

*a < *b	returns a negative value
*a == *b	returns a zero
*a > *b	returns a positive value

The **qsort( )** function has no return value. It uses the STDLIB.H header file.

The following program loads a 100-element integer array with random numbers, sorts it, and displays the sorted form. Notice the necessary type casts within the **comp( )** function.

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

int comp(const void *i, const void *j);

int main(void)
{
  int sort[100], i;

  for(i=0; i<100; i++)
    sort[i] = rand();

  qsort(sort, 100, sizeof(int), comp);

  for(i=0; i<100; i++)
    printf("%d\n", sort[i]);

  return 0;
}

int comp(const void *i, const void *j)
{
  return *(int*)i - *(int*)j;
}
```

EXERCISES

1. Compile and run all of the example programs. Experiment with them, making minor changes.
2. Another of C's standard library functions is called **bsearch( )**. This function searches a sorted array, given a key. It returns a pointer to the first entry in the array that matches the key. If no match is found, a null pointer is returned. Its prototype is

   void *bsearch(const void **key*, const void **array*, size_t *number*, size_t *size*,
   int (**comp*)(const void **a*, const void **b*));

   All the parameters to **bsearch( )** are the same as for **qsort( )** except the first, which is a pointer to *key*, the object being sought. The **comp( )** function operates the same for **bsearch( )** as it does for **qsort( )**.

   Modify the program in Example 3 so that after the array is sorted, the user is prompted to enter a number. Next, using **bsearch( )**, search the sorted array and report if a match is found.
3. Add a function called **modulus( )** to the final version of the arithmetic program in Example 2. Have the function return the result of **a** % **b**. Add this option to the menu and fully integrate it into the program.

MASTER DYNAMIC ALLOCATION

This final section of the book introduces you to C's dynamic-allocation system. *Dynamic allocation* is the process by which memory is allocated as needed during runtime. This allocated memory can be used for a variety of purposes. Most commonly, memory is allocated by applications that need to take full advantage of all the memory in the computer. For example, a word processor will want to let the user edit documents that are as large as possible. However, if the word processor uses a normal character array, it must fix its size at compile time. Thus, it would have to be compiled to run in computers with the minimum amount of memory, not allowing users with more memory to edit larger documents. If memory is allocated dynamically (as needed until memory is exhausted), however, any user may make full use of the memory in the system. Other uses for dynamic allocation include linked lists and binary trees.

The core of C's dynamic-allocation functions are **malloc( )**, which allocates memory, and **free( )**, which releases previously allocated memory. Their prototypes are

void *malloc(size_t *numbytes*);

void free(void **ptr*);

Here, *numbytes* is the number of bytes of memory you wish to allocate. The **malloc( )** function returns a pointer to the start of the allocated piece of memory. If **malloc( )** cannot fulfill the memory request—for example, there may be insufficient memory available—it returns a null pointer. To free memory, call **free( )** with a pointer to the start of the block of memory (previously allocated using **malloc( )**) you wish to free. Both functions use the header file STDLIB.H.

Memory is allocated from a region called the *heap*. Although the actual physical layout of memory may differ, conceptually the heap lies between your program and the stack. Since this is a finite area, an allocation request can fail when memory is exhausted.

When a program terminates, all allocated memory is automatically released.

EXAMPLES

1. You must confirm that a call to **malloc( )** is successful before you use the pointer it returns. If you perform an operation on a null pointer, you could crash your program and maybe even the entire computer. The easiest way to check for a valid pointer is shown in this fragment:

```
p = malloc(SIZE);

if(!p) {
  printf("Allocation Error");
```

```
    exit(1);
  }
```

2. The following program allocates 80 bytes and assigns a character pointer to it. This creates a dynamic character array. It then uses the allocated memory to input a string using **gets( )**. Finally, the string is redisplayed and the pointer is freed. (As stated earlier, all memory is freed when the program ends, so the call to **free( )** is included in this program simply to demonstrate its use.)

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

int main(void)
{
  char *p;

  p = malloc(80);

  if(!p) {
    printf("Allocation Failed");
    exit(1);
  }

  printf("Enter a string: ");
  gets(p);
  printf(p);
  free(p);

  return 0;
}
```

3. The next program tells you approximately how much free memory is available to your program.

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

int main(void)
{
  char *p;
  long l;

  l = 0;
  do {
    p = malloc(1000);
    if(p) l += 1000;
  } while(p);

  printf("Approximately %ld bytes of free memory.", l);

  return 0;
}
```

The program works by allocating 1000-byte-long chunks of memory until an allocation request fails. When **malloc( )** returns null, the heap is exhausted. Hence, the value of **l** represents (within 1000 bytes) the amount of free memory available to the program.

4. One good use for dynamic allocation is to create buffers for file I/O when you are using **fread( )** and/or **fwrite( )**. Often, you only need a buffer for a short period of time, so it makes sense to allocate it when needed and free it when done. The following program shows how dynamic allocation can be used to create a buffer The program allocates enough space to hold ten floating-point values. It then assigns ten random numbers to the allocated memory, indexing the pointer as an array. Next, it writes the values to disk and frees the memory. Finally, it reallocates memory, reads the file and displays the random numbers. Although there is no need to free and then reallocate the memory that serves as a file buffer in this short example, it illustrates the basic idea.

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

int main(void)
{
  int i;
  double *p;
  FILE *fp;

  /* get memory */
  p = malloc(10 * sizeof(double));
  if(!p) {
    printf("Allocation Error");
    exit(1);
  }

  /* generate 10 random numbers */
  for(i=0; i<10; i++)
    p[i] = (double) rand();

  if((fp = fopen("myfile", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  /* write the entire array in one step */
  if(fwrite(p, 10*sizeof(double), 1, fp) != 1) {
    printf("Write Error.\n");
    exit(1);
  }
  fclose(fp);

  free(p); /* memory not needed now */

  /*
    imagine something transpires here
    .
    .
    .
  */

  /* get memory again */
  p = malloc(10 * sizeof(double));
  if(!p) {
    printf("Allocation Error");
    exit(1);
  }

  if((fp = fopen("myfile", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }
  /* read the entire array in one step */
  if(fread(p, 10*sizeof(double), 1, fp) != 1) {
    printf("Read Error.\n");
```

```
      exit(1);
    }
    fclose(fp);

    /* display the array */
    for(i=0; i<10; i++) printf("%f ", p[i]);
    free(p);

    return 0;
  }
```

5. Just as array boundaries can be overrun, so can the boundaries of allocated memory. For example, this fragment is syntactically valid, but wrong.

```
p = malloc(10);

for(i=0; i<100; i) p[i] = i;
```

EXERCISES

1. Compile and run the example programs.
2. Write a program that creates a ten-element dynamic integer array. Next, using pointer arithmetic or array indexing, assign the values 1 through 10 to the integers that comprise the array. Finally, display the values and free the memory.
3. What's wrong with this fragment?

```
char *p;

*p = malloc(10);

gets(p);
```

Mastery Skills Check

At this point you should be able to answer these questions and perform these exercises:

1. What is the difference between using quotes and angle brackets with the **#include** directive?
2. Using an **#ifdef**, show how to conditionally compile this fragment of code based upon whether **DEBUG** is defined or not.

   ```
   if(!(j%2)) {
     printf("j = %d\n", j);
     j = 0;
   }
   ```

3. Using the fragment from Exercise 2, show how you can conditionally compile the code when **DEBUG** is defined as 1. (Hint: Use **#if**).
4. How do you undefine a macro?
5. What is _ _**FILE**_ _ and what does it represent?
6. What do the **#** and **##** preprocessor operators do?
7. Write a program that sorts the string "this is a test of qsort". Display the sorted output.
8. Write a program that dynamically allocates memory for one **double**. Have the program assign that location the value 99.01, display the value, and then free the memory.

Cumulative Skills Check

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. Section 10.1, Example 3, presents a computerized card-catalog program that uses an array of structures to hold information on books. Change this program so that only an array of structure

pointers is created, and use dynamically allocated memory to actually hold the information for each book as it is entered. This way, less memory is used when information on only a few books is stored.

2. Show the macro equivalent of this function:

```
char code_it(char c)
{
  return ~c;
}
```

   Demonstrate that your macro version works in a program.

3. On your own, look over the programs that you have written in the course of working through this book. Try to find places where you can:
   - ▼ Use conditional compilation.
   - ▼ Replace a short function with a function-like macro.
   - ▼ Replace statically allocated arrays with dynamic arrays.
   - ▼ Use function pointers.

4. On your own, study the user's manual or online documentation for your C compiler, paying special attention to the description of its standard library functions. The C standard library contains several hundred library functions that can make your programming tasks easier. Also, Appendix A in this book discusses some of the most common library functions.

5. Now that you have finished this book, go back and skim through each chapter, thinking about how each aspect of C relates to the rest of it. As you will see, C is a highly integrated language, in which one feature complements another. The connection between pointers and arrays, for example, is pure elegance.

6. C is a language best learned by doing! Continue to write programs in C and to study other programmers' programs. You will be surprised at how quickly C will become second nature!

7. Finally, you now have the necessary foundation in C to allow you to move on to C++, C's object-oriented extension. If C++ programming is in your future, proceed to *Teach Yourself C++*, (Berkeley, CA, Osborne/McGraw-Hill). It picks up where this book leaves off.

A

Some Common C Library Functions

THIS appendix discusses a number of the more frequen
ANSI C library functions. If you have looked through
library section in your C/C++ compiler's documentat
are no doubt aware that there are a great many librar
functions. It is far beyond the scope of this book to cover
one. However, the ones you will most commonly need are discuss
The library functions can be grouped into the following cate

- I/O functions
- String and character functions
- Mathematics functions
- Time and date functions
- Dynamic allocation functions
- Miscellaneous functions

The I/O functions were thoroughly covered in Chapters 8 ar
will not be expanded upon here.

Each function's description begins with the header file requi
the function followed by its prototype. The prototype provides
with a quick way of knowing what types of arguments and how
of them the function takes and what type of value it returns.

Keep in mind that ANSI C specifies many data types, which
defined in the header files used by the functions. New type nar
be discussed as they are introduced.

STRING AND CHARACTER FUNCTIO

The C standard library has a rich and varied set of string- and
character-handling functions. In C, a string is a null-terminated
of characters. The declarations for the string functions are foun
header file STRING.H. The character functions use CTYPE.H as
header file.

Because C has no bounds-checking on array operations, it is
programmer's responsibility to prevent an array overflow.

The character functions are declared with an integer parameter. While this is true, only the low-order byte is used by the function. Generally, you are free to use a character argument because it will automatically be elevated to **int** at the time of the call.

#include <ctype.h>
int isalnum(int ch);

Description The **isalnum( )** function returns nonzero if its argument is either a letter or a digit. If the character is not alphanumeric, then 0 is returned.

Example This program checks each character read from **stdin** and reports all alphanumeric ones:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(isalnum(ch)) printf("%c is alphanumeric\n", ch);
  }

  return 0;
}
```

#include <ctype.h>
int isalpha(int ch);

Description The **isalpha( )** function returns nonzero if *ch* is a letter of the alphabet; otherwise 0 is returned.

Example This program checks each character read from **stdin** and reports all those that are letters of the alphabet:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(isalpha(ch)) printf("%c is a letter\n", ch);
  }

  return 0;
}
```

#include <ctype.h>
int iscntrl(int ch);

Description The **iscntrl( )** function returns nonzero if *ch* is between 0 and 0x1F or is equal to 0x7F (DEL); otherwise 0 is returned.

Example This program checks each character read from **stdin** and reports all control characters:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(iscntrl(ch))
      printf("%c is a control character\n", ch);
  }

  return 0;
}
```

#include <ctype.h> int isdigit(int ch);

Description The **isdigit( )** function returns nonzero if *ch* is a digit (0 through 9); otherwise 0 is returned.

Example This program checks each character read from **stdin** and reports all those that are digits:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(isdigit(ch)) printf("%c is a digit\n", ch);
  }

  return 0;
}
```

#include <ctype.h> int isgraph(int ch);

Description The **isgraph( )** function returns nonzero if *ch* is any printable character other than a space; otherwise 0 is returned. Printable characters are in the range 0x21 through 0x7E.

Example This program checks each character read from **stdin** and reports all printing characters:

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

int main(void)
{
  char ch;
  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(isgraph(ch))
      printf("%c is a printing character\n", ch);
  }

  return 0;
}
```

#include <ctype.h>
int islower(int ch);

Description The **islower( )** function returns nonzero if *ch* is a lowercase letter (a through z); otherwise 0 is returned.

Example This program checks each character read from **stdin** and reports all those that are lowercase letters:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(islower(ch)) printf("%c is lowercase\n", ch);
  }

  return 0;
}
```

#include <ctype.h>
int isprint(int ch);

Description The **isprint( )** function returns nonzero if *ch* is a printable character, including a space; otherwise 0 is returned. Printable characters are often in the range 0x20 through 0x7E.

Example This program checks each character read from **stdin** and reports all those that are printable:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch=='Q') break;
    if(isprint(ch)) printf("%c is printable\n", ch);
  }

  return 0;
}
```

#include <ctype.h>
int ispunct(int ch);

Description The **ispunct( )** function returns nonzero if *ch* is a punctuation character, excluding the space; otherwise 0 is returned. The term "punctuation," as defined by this function, includes all printing characters that are neither alphanumeric nor a space.

Example This program checks each character read from **stdin** and reports all those that are punctuation:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(ispunct(ch)) printf("%c is punctuation\n", ch);
  }
  return 0;
}
```

#include <ctype.h>
int isspace(int ch);

Description The **isspace( )** function returns nonzero if *ch* is either a space, tab, vertical tab, form feed, carriage return, or newline character; otherwise 0 is returned.

Example This program checks each character read from **stdin** and reports all those that are whitespace characters:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(isspace(ch)) printf("%c is whitespace\n", ch);
    if(ch==' ') break;
  }

  return 0;
}
```

#include <ctype.h>
int isupper(int ch);

Description The **isupper( )** function returns nonzero if *ch* is an uppercase letter (A through Z); otherwise 0 is returned.

Example This program checks each character read from **stdin** and reports all those that are uppercase letters:

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

```
int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(isupper(ch)) printf("%c is uppercase\n", ch);
  }

  return 0;
}
```

#include <ctype.h>
int isxdigit(int ch);

Description The **isxdigit( )** function returns nonzero if *ch* is a hexadecimal digit; otherwise 0 is returned. A hexadecimal digit will be in one of these ranges: **A** through **F**, **a** through **f**, or **0** through **9**.

Example This program checks each character read from **stdin** and reports all those that are hexadecimal digits:

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

int main(void)
{
  char ch;

  for(;;) {
    ch = getchar();
    if(ch==' ') break;
    if(isxdigit(ch)) printf("%c is hexadecimal \n", ch);
  }

  return 0;
}
```

#include <string.h>
char *strcat(char *str1, const char *str2);

Description The **strcat( )** function concatenates a copy of *str2* to *str1* and terminates *str1* with a null. The null terminator originally ending *str1* is overwritten by the first character of *str2*. The string *str2* is untouched by the operation. The **strcat( )** function returns *str1*.

No bounds-checking takes place, so it is the programmer's responsibility to ensure that str1 *is large enough to hold both its original contents and those of* str2.

Example This program appends the first string read from **stdin** to the second. For example, assuming the user enters **hello** and **there**, the program will print **therehello**.

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

int main(void)
{
  char s1[80], s2[80];

  printf("Enter two strings: ");
  gets(s1);
  gets(s2);

  strcat(s2, s1);
  printf(s2);

  return 0;
}
```

#include <string.h>
char *strchr(const char *str, int ch);

Description The **strchr( )** function returns a pointer to the first occurrence of the low-order byte of *ch* in the string pointed to by *str*. If no match is found, a null pointer is returned.

Example This prints the string **is a test**:

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

int main(void)
{
  char *p;

  p = strchr("this is a test", ' ');
  printf(p);

  return 0;
}
```

#include <string.h>
int strcmp(const char *str1, const char *str2);

Description A **strcmp( )** function lexicographically compares two null-terminated strings and returns an integer based on the outcome, as shown here:

Result	**Meaning**
less than 0	*str1* is less than *str2*
0	*str1* is equal to *str2*
greater than 0	*str1* is greater than *str2*

Example The following function can be used as a password verification routine. It will return 0 on failure and 1 on success.

```
#include <string.h>

int password(void)
{
  char s[80];

  printf("Enter password: ");
  gets(s);

  if(strcmp(s,"pass")) {
    printf("Invalid Password\n");
    return 0;
  }
  return 1;
}
```

#include <string.h>
char *strcpy(char *str1, const char *str2);

Description The **strcpy( )** function is used to copy the contents of *str2* into *str1*; *str2* must be a pointer to a null-terminated string. The **strcpy( )** function returns a pointer to *str1*.

If *str1* and *str2* overlap, the behavior of **strcpy( )** is undefined.

Example The following code fragment will copy "hello" into string **str**:

```
char str[80];
strcpy(str, "hello");
```

#include <string.h>
size_t strlen(const char *str);

Description The **strlen( )** function returns the length of the null-terminated string pointed to by *str*. The null is not counted. The **size_t** type is defined in STRING.H.

Example The following code fragment will print **5** on the screen:

```
strcpy(s, "hello");
printf("%d", strlen(s));
```

#include <stdio.h>
char *strstr(const char *str1, const char *str2);

Description The **strstr( )** function returns a pointer to the first occurrence of the string pointed to by *str2* in the string pointed to by *str1* (except *str2*'s null terminator). It returns a null pointer if no match is found.

Example This program displays the message **is a test**:

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

int main(void)
{
  char *p;

  p = strstr("this is a test", "is");
  printf(p);

  return 0;
}
```

#include <string.h>
char *strtok(char *str1, const char *str2);

Description The **strtok( )** function returns a pointer to the next token in the string pointed to by *str1*. The characters making up the string pointed to by *str2* are the delimiters that separate each token. A null pointer is returned when there are no more tokens.

The first time **strtok( )** is called, *str1* is actually used in the call. Subsequent calls use a null pointer for the first argument. In this way the entire string can be reduced to its tokens.

It is possible to use a different set of delimiters for each call to **strtok( )**.

Example This program tokenizes the string "The summer soldier, the sunshine patriot" with spaces and commas as the delimiters. The output will be **The | summer | soldier | the | sunshine | patriot**.

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

int main(void)
{
  char *p;

  p = strtok("The summer soldier, the sunshine patriot", " ,");

  printf(p);
  do {
   p = strtok('\0', ", ");
   if(p) printf("|%s", p);
  } while(p);

  return 0;
}
```

#include <ctype.h>
int tolower(int ch);

Description The **tolower( )** function returns the lowercase equivalent of *ch* if *ch* is a letter; otherwise *ch* is returned unchanged.

Example This fragment displays **q**:

```
putchar(tolower('Q'));
```

#include <ctype.h>
int toupper(int ch);

Description The **toupper( )** function returns the uppercase equivalent of *ch* if *ch* is a letter; otherwise *ch* is returned unchanged.

Example This displays **A**:

```
putchar(toupper('a'));
```

THE MATHEMATICS FUNCTIONS

ANSI C defines several mathematics functions that take **double** arguments and return **double** values. These functions fall into the following categories:

- Trigonometric functions
- Hyperbolic functions
- Exponential and logarithmic functions
- Miscellaneous functions

All the math functions require that the header MATH.H be included in any program that uses them. In addition to declaring the math functions, this header defines a macro called **HUGE_VAL**. If an operation produces a result that is too large to be represented by a **double**, an overflow occurs, which causes the routine to return **HUGE_VAL**. This is called a *range error*. For all the mathematics functions, if the input value is not in the domain for which the function is defined, a *domain error* occurs.

All angles are specified in radians.

#include <math.h>
double acos(double arg);

Description The **acos( )** function returns the arc cosine of *arg*. The argument to **acos( )** must be in the range -1 through 1; otherwise a domain error will occur.

Example This program prints the arc cosines, in one-tenth increments, of the values -1 through 1:

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

int main(void)
{
  double val = -1.0;

  do {
    printf("arc cosine of %f is %f\n", val, acos(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

#include <math.h>
double asin(double arg);

Description The **asin( )** function returns the arc sine of *arg*. The argument to **asin( )** must be in the range -1 through 1; otherwise a domain error will occur.

Example This program prints the arc sines, in one-tenth increments, of the values -1 through 1:

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

int main(void)
{
  double val=-1.0;

  do {
    printf("arc sine of %f is %f\n", val, asin(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

#include <math.h>
double atan(double arg);

Description The **atan( )** function returns the arc tangent of *arg*.

Example This program prints the arc tangents, in one-tenth increments, of the values -1 through 1:

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

int main(void)
{
  double val=-1.0;

  do {
    printf("arc tangent of %f is %f\n", val, atan(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

#include <math.h>
double atan2(double y, double x);

Description The **atan2( )** function returns the arc tangent of *y*/*x*. It uses the signs of its arguments to compute the quadrant of the return value.

Example This program prints the arc tangents, in one-tenth increments of *y*, from −1 through 1:

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

int main(void)
{
  double y=-1.0;

  do {
    printf("atan2 of %f is %f\n", y, atan2(y, 1.0));
    y += 0.1;
  } while(y<=1.0);

  return 0;
}
```

#include <math.h>
double ceil(double num);

Description The **ceil( )** function returns the smallest integer (represented as a **double**) that is not less than *num*. For example, given 1.02, **ceil( )** would return 2.0; given −1.02, **ceil( )** would return −1.

Example This fragment prints **10.0** on the screen:

```
printf("%f", ceil(9.9));
```

#include <math.h> double cos(double arg);

Description The **cos( )** function returns the cosine of *arg*. The value of *arg* must be in radians.

Example This program prints the cosines, in one-tenth increments, of the values –1 through 1:

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

int main(void)
{
  double val=-1.0;

  do {
    printf("cosine of %f is %f\n", val, cos(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

#include <math.h> double cosh(double arg);

Description The **cosh( )** function returns the hyperbolic cosine of *arg*.

Example This program prints the hyperbolic cosines, in one-tenth increments, of the values –1 through 1:

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

int main(void)
{
  double val=-1.0;

  do {
    printf("hyperbolic cosine of %f is %f\n", val, cosh(val))
    val += 0.1;
  :
  } while(val<=1.0);

  return 0;
}
```

#include <math.h>
double exp(double arg);

Description The **exp( )** function returns the natural logarithm *e* raised to the *arg* power.

Example This fragment displays the value of *e* (rounded to 2.718282):

```
printf("Value of e to the first: %f", exp(1.0));
```

#include <math.h>
double fabs(double num);

Description The **fabs( )** function returns the absolute value of *num*.

Example This program prints the numbers **1.0 1.0** on the screen:

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

int main(void)
{
  printf("%1.1f %1.1f", fabs(1.0), fabs(-1.0));

  return 0;
}
```

#include <math.h>
double floor(double num);

Description The **floor( )** function returns the largest integer (represented as a **double**) not greater than *num*. For example, given 1.02, **floor( )** would return 1.0; given −1.02, **floor( )** would return −2.0.

Example This fragment prints **10.0** on the screen:

```
printf("%f", floor(10.9));
```

#include <math.h>
double log(double num);

Description The **log( )** function returns the natural logarithm for *num*. A domain error occurs if *num* is negative and a range error occurs if the argument is 0.

Example This program prints the natural logarithms for the numbers 1 through 10:

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

int main(void)
{
  double val=1.0;

  do {
    printf("%f %f\n", val, log(val));
    val++;
  } while(val<11.0);

  return 0;
}
```

#include <math.h>
double log10(double num);

Description The **log10( )** function returns the base 10 logarithm for the variable *num*. A domain error occurs if *num* is negative and a range error occurs if the argument is 0.

Example This program prints the base 10 logarithms for the numbers 1 through 10:

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

```
int main(void)
{
  double val=1.0;

  do {
    printf("%f %f\n", val, log10(val));
    val++;
  } while(val<11.0);

  return 0;
}
```

#include <math.h>
double pow(double base, double exp);

Description The **pow( )** function returns *base* raised to the *exp* power (*base*^{exp}). A domain error may occur if *base* is 0 and *exp* is less than or equal to 0. A domain error will occur if *base* is negative and *exp* is not an integer. An overflow produces a range error.

Example This program prints the first ten powers of 10:

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

int main(void)
{
  double x=10.0, y=0.0;

  do {
    printf("%f ", pow(x, y));
    y++;
  } while(y<11);

  return 0;
}
```

#include <math.h>
double sin(double arg);

Description The **sin( )** function returns the sine of *arg*. The value of *arg* must be in radians.

Example This program prints the sines, in one-tenth increments, of the values -1 through 1:

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

int main(void)
{
  double val=-1.0;

  do {
    printf("sine of %f is %f\n", val, sin(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

#include <math.h>
double sinh(double arg);

Description The **sinh( )** function returns the hyperbolic sine of *arg*.

Example The following program prints the hyperbolic sines, in one-tenth increments, of the values -1 through 1:

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

int main(void)
{
  double val=-1.0;

  do {
    printf("hyperbolic sine of %f is %f\n", val, sinh(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

#include <math.h>
double sqrt(double num);

Description The **sqrt( )** function returns the square root of *num*. If called with a negative argument, a domain error will occur.

Example This fragment prints **4.0** on the screen:

```
printf("%f", sqrt(16.0));
```

#include <math.h>
double tan(double arg);

Description The **tan( )** function returns the tangent of *arg*. The value of *arg* must be in radians.

Example This program prints the tangents, in one-tenth increments, of the values –1 through 1:

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

int main(void)
{
  double val=-1.0;

  do {
    printf("tangent of %f is %f\n", val, tan(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

#include <math.h>
double tanh(double arg);

Description The **tanh( )** function returns the hyperbolic tangent of *arg*.

Example This program prints the hyperbolic tangents, in one-tenth increments, of the values −1 through 1:

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

int main(void)
{
  double val=-1.0;

  do {
    printf("tanh of %f is %f\n", val, tanh(val));
    val += 0.1;
  } while(val<=1.0);

  return 0;
}
```

TIME AND DATE FUNCTIONS

The time and date functions require the header TIME.H for their prototypes. This header file also defines four types and two macros. The type **time_t** is able to represent the system time and date as a **long** integer. This is called the *calendar time*. The structure type **tm** holds date and time broken down into its elements. The **tm** structure is defined as shown here:

```
struct tm {
  int tm_sec;    /* seconds, 0-61 */
  int tm_min;    /* minutes, 0-59 */
  int tm_hour;   /* hours, 0-23 */
  int tm_mday;   /* day of the month, 1-31*/
  int tm_mon;    /* months since Jan, 0-11 */
  int tm_year;   /* years from 1900 */
  int tm_wday;   /* days since Sunday, 0-6 */
  int tm_yday;   /* days since Jan 1, 0-365 *;
  int tm_isdst; /* Daylight Saving Time indicator */
};
```

The value of **tm_isdst** will be positive if Daylight Saving Time is in effect, 0 if it is not in effect, and negative if there is no information available. When the date and time are represented in this way, they are referred to as *broken-down time*.

The type **clock_t** is defined the same as **time_t**. The header file also defines **size_t**.

The macros defined are **NULL** and **CLOCKS_PER_SEC**.

#include <time.h>
char *asctime(const struct tm *ptr);

Description The **asctime( )** function returns a pointer to a string that contains the time and date stored in the structure pointed to by *ptr* after it has been converted into the following form:

```
day month date hours:minutes:seconds year\n\0
```

For example:

```
Wed Jun 19 12:05:34 1999
```

The structure pointer passed to **asctime( )** is generally obtained from either **localtime( )** or **gmtime( )**.

The buffer used by **asctime( )** to hold the formatted output string is a statically allocated character array and is overwritten each time the function is called. If you want to save the contents of the string, you need to copy it elsewhere.

Example This program displays the local time defined by the system:

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

int main(void)
{
  struct tm *ptr;
  time_t lt;

  lt = time(NULL);
  ptr = localtime(&lt);
  printf(asctime(ptr));

  return 0;
}
```

#include <time.h>
clock_t clock(void);

Description The **clock( )** function returns the number of system clock cycles that have occurred since the program began execution. To compute the number of seconds, divide this value by the **CLOCKS_PER_SEC** macro.

Example The following program displays the number of system clocl cycles occurring since it began:

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

int main(void)
{
  int i;

  for(i=0; i<10000; i++);

  printf("%u", clock());

  return 0;
}
```

#include <time.h>
char *ctime(const time_t *time);

Description The **ctime( )** function returns a pointer to a string of the form

day month date hours:minutes:seconds year\n\0

given a pointer to the calendar time. The calendar time is generally obtained through a call to **time( )**. The **ctime( )** function is equivalent to:

```
asctime(localtime(time))
```

The buffer used by **ctime( )** to hold the formatted output string is a statically allocated character array and is overwritten each time the function is called. If you wish to save the contents of the string, you need to copy it elsewhere.

Example This program displays the local time defined by the system:

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

int main(void)
{
  time_t lt;

  lt = time(NULL);
  printf(ctime(&lt));

  return 0;
}
```

#include <time.h>
double difftime(time_t time2, time_t time1);

Description The **difftime( )** function returns the difference, in seconds, between *time1* and *time2*. That is, *time2* - *time1*.

Example This program times the number of seconds that it takes for the empty **for** loop to go from 0 to 500000.

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

int main(void)
{
  time_t start, end;
  long unsigned int t;

  start = time(NULL);
  for(t=0; t<500000L; t++);
  end = time(NULL);
  printf("Loop required %f seconds.\n", difftime(end, start));

  return 0;
}
```

#include <time.h>
strut tm *gmtime(const time_t *time);

Description The **gmtime( )** function returns a pointer to the broken-down form of *time* in the form of a **tm** structure. The time is represented in Coordinated Universal Time (i.e., Greenwich Mean Time). The *time* value is generally obtained through a call to **time( )**.

The structure used by **gmtime( )** to hold the broken-down time is statically allocated and is overwritten each time the function is called. If you wish to save the contents of the structure, you need to copy it elsewhere.

Example This program prints both the local time and the Coordinated Universal Time of the system:

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

/* print local and Coordinated Universal time */
int main(void)
{
  struct tm *local, *coordinated;
  time_t t;

  t = time(NULL);
  local = localtime(&t);
  printf("Local time and date: %s", asctime(local));
  coordinated = gmtime(&t);
  printf("Coordinated Universal time and date: %s",
         asctime(coordinated));

  return 0;
}
```

#include <time.h>
struct tm *localtime(const time_t *time);

Description The **localtime( )** function returns a pointer to the broken-down form of *time* in the form of a **tm** structure. The time is represented in local time. The *time* value is generally obtained through a call to the **time( )** function.

The structure used by **localtime( )** to hold the broken-down time is statically allocated and is overwritten each time the function is called. If you wish to save the contents of the structure, you need to copy it elsewhere.

Example This program prints both the local time and the Coordinated Universal time of the system:

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

/* print local and Coordinated Universal time */
int main(void)
{
  struct tm *local;
  time_t t;

  t = time(NULL);
  local = localtime(&t);
  printf("Local time and date: %s", asctime(local));
  local = gmtime(&t);
  printf("Coordinated Universal time and date: %s",
          asctime(local));

  return 0;
}
```

#include <time.h>
time_t time(time_t *systime);

Description The **time( )** function returns the current calendar time of the system. If the system has no time-keeping mechanism, then −1 is returned.

The **time( )** function can be called either with a null pointer or with a pointer to a variable of type **time_t**. If the latter is used, then the argument will also be assigned the calendar time.

Example This program displays the local time defined by the system:

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

```
int main(void)
{
  struct tm *ptr;
  time_t lt;

  lt = time(NULL);
  ptr = localtime(&lt);
  printf(asctime(ptr));

  return 0;
}
```

DYNAMIC ALLOCATION

There are two primary ways a C program can store information in the main memory of the computer. The first uses global and local variables—including arrays and structures. In the case of global and static local variables, the storage is fixed throughout the runtime of your program. For dynamic local variables, storage is allocated on the stack. Although these variables are efficiently implemented in C, they require the programmer to know in advance the amount of storage needed for every situation. The second way information can be stored is with C's dynamic allocation system. In this method, storage for information is allocated from the free memory area (called the *heap*) as it is needed.

The ANSI C standard specifies that the header information necessary to the dynamic allocation system is in STDLIB.H. In this file, the type **size_t** is defined. This type is used extensively by the allocation functions and is essentially the equivalent of **unsigned**.

#include <stdlib.h>
void *calloc(size_t num, size_t size);

Description The **calloc( )** function returns a pointer to the allocated memory. The amount of memory allocated is equal to *num* * *size*. That is, **calloc( )** allocates sufficient memory for an array of *num* objects of size *size*.

The **calloc( )** function returns a pointer to the first byte of the allocated region. If there is not enough memory to satisfy the request, a null pointer is returned.

It is always important to verify that the return value is not a null pointer before attempting to use it.

Example This function returns a pointer to a dynamically allocated array of 100 **floats**:

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

float *get_mem(void)
{
  float *p;

  p = calloc(100, sizeof(float));
  if(!p) {
    printf("Allocation error - aborting.\n");
    exit(1);
  }
  return p;
}
```

#include <stdlib.h>
void free(void *ptr);

Description The **free( )** function deallocates the memory pointed to by *ptr*. This makes the memory available for future allocation.

It is imperative that the **free( )** function be called only with a pointer that was previously allocated using one of the dynamic allocation system's functions, such as **malloc( )** or **calloc( )**. Using an invalid pointer in the call will probably destroy the memory management mechanism and cause a system crash.

Example This program first allocates room for 100 user-entered strings and then frees them:

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

int main(void)
```

```
{
  char *str[100];
  int i;

  for(i=0; i<100; i++) {
    if((str[i] = malloc(128))==NULL) {
      printf("Allocation error - aborting.\n");
      exit(0);
    }
    gets(str[i]);
  }

  /* now free the memory */
  for(i=0; i<100; i++) free(str[i]);

  return 0;
}
```

#include <stdlib.h>
void *malloc(size_t size);

Description The **malloc( )** function returns a pointer to the first byte of a region of memory of size *size* that has been allocated from the heap. (Remember, the heap is a region of free memory managed by C's dynamic allocation subsystem.) If there is insufficient memory in the heap to satisfy the request, **malloc( )** returns a null pointer. It is always important to verify that the return value is not a null pointer before attempting to use it. Attempting to use a null pointer will usually result in a system crash.

Example This function allocates sufficient memory to hold structures of type **addr**:

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

struct addr {
  char name[40];
  char street[40];
  char city[40];
  char state[3];
  char zip[10];
};
.
.
.
struct addr *get_struct(void)
{
  struct addr *p;

  if((p = malloc(sizeof(struct addr)))==NULL)
  {
    printf("Allocation error - aborting.\n");
    exit(0);
  }
  return p;
}
```

#include <stdlib.h>
void *realloc(void *ptr, size_t size);

Description The **realloc( )** function changes the size of the allocated memory pointed to by *ptr* to that specified by *size*. The value of *size* may be greater or less than the original. A pointer to the memory block is returned since it may be necessary for **realloc( )** to move the block to increase its size. If this occurs, the contents of the old block are copied into the new block—no information is lost.

If there is not enough free memory in the heap to allocate *size* bytes, a null pointer is returned. This means it is important to verify the success of a call to **realloc( )**.

Example This program first allocates 17 characters, copies the string "this is 16 chars" into the space, and then uses **realloc( )** to increase the size to 18 in order to place a period at the end.

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

int main(void)
{
  char *p;

  p = malloc(17);
  if(!p) {
    printf("Allocation error - aborting.\n");
    exit(1);
  }

  strcpy(p, "this is 16 chars");

  p = realloc(p,18);
  if(!p) {
    printf("Allocation error - aborting.\n");
    exit(1);
  }

  strcat(p, ".");

  printf(p);

  free(p);

  return 0;
}
```

MISCELLANEOUS FUNCTIONS

The functions discussed in this section are all standard functions that don't easily fit in any other category.

#include <stdlib.h>
void abort(void);

Description The **abort( )** function causes immediate termination of a program. Whether it closes any open files is defined by the implementation, but generally it won't.

Example In this program, if the user enters **A**, the program will terminate:

```
#include <stdlib.h>
#include <conio.h>
```

```
int main(void)
{
  for(;;)
    if(getche()=='A') abort();

  return 0;
}
```

#include <stdlib.h> int abs(int num);

Description The **abs( )** function returns the absolute value of the integer *num*.

Example This function converts the user-entered numbers into their absolute values:

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

int get_abs(void)
{
  char num[80];

  gets(num);

  return abs(atoi(num));
}
```

#include <stdlib.h> double atof(const char * str);

Description The **atof( )** function converts the string pointed to by *str* into a **double** value. The string must contain a valid floating-point number. If this is not the case, the returned value is 0.

The number may be terminated by any character that cannot be part of a valid floating-point number. This includes whitespace characters, punctuation (other than periods), and characters other than 'E' or 'e'. Thus, if **atof( )** is called with "100.00HELLO", the value 100.00 will be returned.

Example This program reads two floating-point numbers and displays their sum:

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

int main(void)
{
  char num1[80], num2[80];

  printf("Enter first: ");
  gets(num1);
  printf("Enter second: ");
  gets(num2);
  printf("The sum is: %f", atof(num1) + atof(num2));

  return 0;
}
```

#include <stdlib.h>
int atoi(const char *str);

Description The **atoi( )** function converts the string pointed to by *str* into an **int** value. The string must contain a valid integer number. If this is not the case, the returned value is 0.

The number may be terminated by any character that cannot be part of a integer number. This includes whitespace characters, punctuation, and other characters. Thus, if **atoi( )** is called with 123.23, the integer value 123 will be returned, anc the 0.23 ignored.

Example This program reads two integer numbers and displays their sum:

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

int main(void)
{
  char num1[80], num2[80];

  printf("Enter first: ");
  gets(num1);
  printf("Enter second: ");
  gets(num2);
  printf("The sum is: %d", atoi(num1) + atoi(num2));

  return 0;
}
```

#include <stdlib.h>
long atol(const char *str);

Description The **atol( )** function converts the string pointed to by *str* into a **long int** value. The string must contain a valid long integer number. If this is not the case, the returned value is 0.

The number may be terminated by any character that cannot be part of an integer number. This includes whitespace characters, punctuation, and other characters. Thus, if **atol( )** is called with 123.23, the integer value 123 will be returned, and the 0.23 ignored.

Example This program reads two long integer numbers and displays their sum:

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

int main(void)
{
  char num1[80], num2[80];

  printf("Enter first: ");
  gets(num1);
  printf("Enter second: ");
  gets(num2);
  printf("The sum is: %ld", atol(num1) + atol(num2));

  return 0;
}
```

#include <stdlib.h>
void *bsearch(const void *key, const void *base,
size_t num, size_t size,
int(*compare)(const void *, const void *));

Description The **bsearch( )** function performs a binary search on the sorted array pointed to by *base* and returns a pointer to the first member that matches the key pointed to by *key*. The number of elements in the array is specified by *num* and the size (in bytes) of each element is described by *size*. (The **size_t** type is defined in STDLIB.H and is essentially the equivalent of **unsigned**.)

The function pointed to by *compare* is used to compare an element of the array with the key. The form of *compare* must be

int *function_name*(const void **arg1*, const void **arg2*)

It must return the following values:

Less than 0	If *arg1* is less than *arg2*
0	If *arg1* is equal to *arg2*
Greater than 0	If *arg1* is greater than *arg2*

The array must be sorted in ascending order, with the lowest address containing the lowest element.

If the array does not contain the key, then a null pointer is returned.

Example This program reads characters entered at the keyboard and determines whether they belong to the alphabet.

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

char *alpha = "abcdefghijklmnopqrstuvwxyz";

int comp(const void *ch, const void *s);

int main(void)
{
  char ch;
  char *p;
  do {
    printf("Enter a character: ");
    scanf("%c%*c",&ch);
    ch = tolower(ch);
    p = bsearch(&ch, alpha, 26, 1, comp);
    if(p) printf("is in alphabet.\n");
    else printf("is not in alphabet.\n");
  } while(p);

  return 0;
}

/* compare two characters */
int comp(const void *ch, const void *s)
{
  return *(char *)ch - *(char *)s;
};
```

#include <stdlib.h>
void exit(int status);

Description The **exit( )** function causes immediate normal termination of a program.

The value of *status* is passed to the calling process, usually the operating system, if the environment supports it. By convention, if the value of *status* is 0, normal program termination is assumed. A nonzero value may be used to indicate an error.

You may also use the predefined macros **EXIT_SUCCESS** and **EXIT_FAILURE** as arguments to **exit( )**.

Example This function performs menu selection for a mailing list program. If **Q** is selected, the program is terminated.

```
char menu(void)
{
  char ch;

  do {
    printf("Enter names (E)\n");
    printf("Delete name (D)\n");
    printf("Print (P)\n");
    printf("Quit (Q)\n");
  } while(!strchr("EDPQ",toupper(ch)));
  if(ch=='Q') exit(0);
  return ch;
}
```

#include <stdlib.h>
long labs(long num);

Description The **labs( )** function returns the absolute value of the **long int** *num*.

Example This function converts the user-entered numbers into their absolute values:

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

long int get_labs(void)
{
  char num[80];

  gets(num);

  return labs(atol(num));
}
```

#include <setjmp.h>
void longjmp(jmp_buf envbuf, int val);

Description The **longjmp( )** function causes program execution to resume at the point of the last call to **setjmp( )**. These two functions are the way ANSI C provides for a jump between functions. Notice that the header SETJMP.H is required.

The **longjmp( )** function operates by resetting the stack as described in *envbuf*, which must have been set by a prior call to **setjmp( )**. This causes program execution to resume at the statement following the **setjmp( )** invocation—the computer is "tricked" into thinking that it never left the function that called **setjmp( )**. (As a somewhat graphic explanation, the **longjmp( )** function "warps" across time and (memory) space to a previous point in your program, without having to perform the normal function-return process.)

The buffer *envbuf* is of type **jmp_buf**, which is defined in the header SETJMP.H. The buffer must have been set through a call to **setjmp( )** prior to calling **longjmp( )**.

The value of *val* becomes the return value of **setjmp( )** and may be interrogated to determine where the long jump came from. The only value not allowed is 0.

It is important to understand that the **longjmp( )** function must be called before the function that called **setjmp( )** returns. If not, the result is technically undefined. In actuality, a crash will almost certainly occur.

By far the most common use of **longjmp( )** is to return from a deeply nested set of routines when a catastrophic error occurs.

Example This program prints **1 2 3**:

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

void f2(void);

jmp_buf ebuf;

int main(void)
{
  char first=1;
  int i;

  printf("1 ");
  i = setjmp(ebuf);
  if(first) {
    first = !first;
    f2();
    printf("this will not be printed");
  }
  printf("%d", i);

  return 0;
}

void f2(void)
```

```
  printf("2 ");
  longjmp(ebuf, 3);
}
```

#include <stdlib.h>
void qsort(void *base, size_t num, size_t size, int(*compare)(const void*, const void*));

Description The **qsort( )** function sorts the array pointed to by *base* using a Quicksort (which was developed by C.A.R. Hoare). The Quicksort is generally considered the best general-purpose sorting algorithm. Upon termination, the array will be sorted. The number of elements in the array is specified by *num* and the size (in bytes) of each element is described by *size*. (The **size_t** type is defined in STDLIB.H and is essentially the equivalent of **unsigned**.)

The function pointed to by *compare* is used to compare two elements in the array. The form of *compare* must be

int *function_name*(const void **arg1*, const void **arg2*)

It must return the following values:

Less than 0	If *arg1* is less than *arg2*
0	If *arg1* is equal to *arg2*
Greater than 0	If *arg1* is greater than *arg2*

The array is sorted in ascending order, with the lowest address containing the lowest element.

Example This program sorts a list of integers and displays the result:

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

int comp(const void *i, const void *j);

int num[10]= {
  1,3, 6, 5, 8, 7, 9, 6, 2, 0
};

int main(void)
```

```
  int i;

  printf("Original array: ");
  for(i=0; i<10; i++) printf("%d ", num[i]);
  printf("\n");

  qsort(num, 10, sizeof(int), comp);

  printf("Sorted array: ");
  for(i=0; i<10; i++) printf("%d ", num[i]);

  return 0;
}

/* compare the integers */
int comp(const void *i, const void *j)
{
  return *(int *)i - *(int *)j;
}
```

#include <stdlib.h>
int rand(void);

Description The **rand( )** function generates a sequence of pseudo-random numbers. Each time it is called, an integer between 0 and **RAND_MAX** is returned. **RAND_MAX** is defined in STDLIB.H. The ANSI standard stipulates that the macro **RAND_MAX** will have a value of at least 32,767.

Example This program displays ten pseudo-random numbers:

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

int main(void)
{
  int i;

  for(i=0; i<10; i++)
    printf("%d ", rand());
  return 0;
}
```

#include <setjmp.h>
int setjmp(jmp_buf envbuf);

Description The **setjmp( )** function saves the contents of the system stack in the buffer *envbuf* for later use by **longjmp( )**.

The **setjmp( )** function returns 0 upon invocation. However, **longjmp( )** passes an argument to **setjmp( )** when it executes, and it is this value (always nonzero) that will appear to be the value of **setjmp( )** after a call to **longjmp( )**.

See the **longjmp( )** section for more information.

Example This program prints **1 2 3**:

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

void f2(void);

jmp_buf ebuf;

int main(void)
{
  char first=1;
  int i;

  printf("1 ");
  i = setjmp(ebuf);
  if(first) {
    first = !first;
    f2( );
    printf("this will not be printed");
  }
  printf("%d",i);

  return 0;
}

void f2(void)
{
  printf("2 ");
  longjmp(ebuf, 3);
}
```

#include <stdlib.h>
void srand(unsigned seed);

Description The **srand( )** function is used to set a starting point for the sequence generated by **rand( )**, which returns pseudo-random numbers.

Generally **srand( )** is used to allow multiple program runs to use different sequences of pseudo-random numbers.

Example This program uses the system time to randomly initialize the **rand( )** function using **srand( )**:

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

/* Seed rand with the system time
   and display the first 100 numbers.
*/
int main(void)
{
  int i, utime;
  long ltime;

  /* get the current calendar time */
  ltime = time(NULL);
  utime = (unsigned int) ltime/2;
  srand(utime);

  for(i=0; i<10; i++) printf("%d ", rand());

  return 0;
```

B

C Keyword Summary

THERE are 32 keywords that, when combined with the formal C syntax, form the C language as defined by the ANSI C standard. These keywords are shown in Table B-1.

All C keywords use lowercase letters. In C, uppercase and lowercase are different; for instance, **else** is a keyword, **ELSE** is not.

An alphabetical summary of each of the keywords follows:

auto

auto is used to create temporary variables that are created upon entry into a block and destroyed upon exit. For example:

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

int main(void)
{
  for(;;) {
    if(getche()=='a') {
      auto int t;
      for(t=0; t<'a'; t++)
        printf("%d ", t);
      break;
    }
  }

  return 0;
}
```

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

TABLE B-1 *Keyword List*

In this example, the variable **t** is created only if the user strikes an **a**. Outside the **if** block, **t** is completely unknown; and any reference to it would generate a compile-time syntax error. The use of **auto** is completely optional since all local variables are **auto** by default.

break

break is used to exit from a **do**, **for**, or **while** loop, bypassing the normal loop condition. It is also used to exit from a **switch** statement.

An example of **break** in a loop is shown here:

```
while(x<100) {
  x = get_new_x();
  if(kbhit()) break; /* key hit on keyboard */
  process(x);
}
```

Here, if a key is pressed, the loop will terminate no matter what the value of **x** is.

In a **switch** statement, **break** effectively keeps program execution from "falling through" to the next **case**. (Refer to the **switch** section for details.)

case

case is covered in conjunction with **switch**.

char

char is a data type used to declare character variables. For example, to declare **ch** to be a character type, you would write:

```
char ch;
```

In C, a character is one byte long.

const

The **const** modifier tells the compiler that the contents of a variable cannot be changed. It is also used to prevent a function from modifying the object pointed to by one of its arguments.

continue

continue is used to bypass portions of code in a loop and forces the conditional expression to be evaluated. For example, the following **while** loop will simply read characters from the keyboard until an **s** is typed:

```
while(ch=getche()) {
  if(ch != 's') continue; /* read another char */
  process(ch);
}
```

The call to **process( )** will not occur until **ch** contains the character **s**.

default

default is used in the **switch** statement to signal a default block of code to be executed if no matches are found in the **switch**. See the **switch** section.

do

The **do** loop is one of three loop constructs available in C. The general form of the **do** loop is

```
do {
  statement block
} while(condition);
```

If only one statement is repeated, the braces are not necessary, but they add clarity to the statement. The **do** loop repeats as long as the *condition* is true.

The **do** loop is the only loop in C that will always have at least one iteration because the condition is tested at the bottom of the loop.

A common use of the **do** loop is to read disk files. This code will read a file until an EOF is encountered.

```
do {
  ch = getc(fp);
  if(!feof(fp)) printf("%c", ch);
} while(!feof(fp));
```

double

double is a data type specifier used to declare double-precision floating-point variables. To declare **d** to be of type **double** you would write the following statement:

```
double d;
```

else

See the **if** section.

enum

The **enum** type specifier is used to create enumeration types. An enumeration is simply a list of named integer constants. For example, the following code declares an enumeration called **color** that consists of three constants: **red**, **green**, and **yellow**.

```
#include <stdio.h>

enum color {red, green, yellow};
enum color c;

int main(void)
{
  c = red;
  if(c==red) printf("is red\n");

  return 0;
}
```

extern

The **extern** data type modifier tells the compiler that a variable is defined elsewhere in the program. This is often used in conjunction with separately compiled files that share the same global data and are linked together. In essence, it notifies the compiler of a variable without redefining it.

As an example, if **first** were declared in another file as an integer, the following declaration would be used in subsequent files:

```
extern int first;
```

float

float is a data type specifier used to declare floating-point variables. To declare **f** to be of type **float** you would write:

```
float f;
```

for

The **for** loop allows automatic initialization and incrementation of a counter variable. The general form is

```
for(initialization; condition; increment) {
  statement block
}
```

If the *statement block* is only one statement, the braces are not necessary.

Although the **for** allows a number of variations, generally the *initialization* is used to set a counter variable to its starting value. The *condition* is generally a relational statement that checks the counter variable against a termination value, and the *increment* increments (or decrements) the counter value. The loop repeats until the condition becomes false.

The following code will print **hello** ten times.

```
for(t=0; t<10; t++) printf("Hello\n");
```

goto

The **goto** causes program execution to jump to the *label* specified in the **goto** statement. The general form of the **goto** is

```
goto label;
.
.
.
label:
```

All *labels* must end in a colon and must not conflict with keywords or function names. Furthermore, a **goto** can branch only within the current function, and not from one function to another.

The following example will print the message **right** but not the message **wrong**:

```
goto labl;
  printf("wrong");
labl:
  printf("right");
```

if

The general form of the **if** statement is

```
if(condition) {
  statement block 1
}
else {
  statement block 2
}
```

If single statements are used, the braces are not needed. The **else** is optional.

The *condition* may be any expression. If that expression evaluates to any value other than 0, then *statement block 1* will be executed; otherwise, if it exists, *statement block 2* will be executed.

The following code fragment can be used for keyboard input and to look for a 'q' which signifies "quit."

```
ch = getche();
if(ch=='q') {
  printf("Program Terminated");
  exit(0);
}
else proceed();
```

int

int is the type specifier used to declare integer variables. For example, to declare **count** as an integer you would write

```
int count;
```

long

long is a data type modifier used to declare long integer and long **double** variables. For example, to declare **count** as a long integer you would write

```
long int count;
```

register

The **register** modifier requests that a variable be stored in the way that allows the fastest possible access. In the case of characters or integers, this usually means a register of the CPU. To declare **i** to be a **register** integer, you would write

```
register int i;
```

return

The **return** statement forces a return from a function and can be used to transfer a value back to the calling routine. For example, the following function returns the product of its two integer arguments.

```
int mul(int a, int b)
{
  return a*b;
}
```

Keep in mind that as soon as a **return** is encountered, the function will return, skipping any other code in the function.

short

short is a data type modifier used to declare small integers. For example, to declare **sh** to be a short integer you would write

```
short int sh;
```

signed

The **signed** type modifier is most commonly used to specify a **signed char** data type.

sizeof

The **sizeof** keyword is a compile-time operator that returns the length of the variable or type it precedes. If it precedes a type, the type must be enclosed in parentheses. For example,

```
printf("%d", sizeof(short int));
```

will print **2** for most C implementations.

The **sizeof** statement's principal use is in helping to generate portable code when that code depends on the size of the C built-in data types.

static

The **static** keyword is a data type modifier that causes the compiler to create permanent storage for the local variable that it precedes. This enables the specified variable to maintain its value between function calls. For example, to declare **last_time** as a **static** integer, you would write

```
static int last_time;
```

static can also be used on global variables to limit their scope to the file in which they are declared.

struct

The **struct** statement is used to create aggregate data types, called structures, that are made up of one or more members. The general form of a structure is

```
struct struct-name {
  type member1 ;
  type member2 ;
  .
  .
  .
  type member N ;
} variable-list ;
```

The individual members are referenced using the dot or arrow operators.

switch

The **switch** statement is C's multi-path branch statement. It is used to route execution in one of several ways. The general form of the statement is

```
switch(int-expression) {
  case constant1: statement-set 1;
    break;
  case constant2 : statement-set 2 ;
    break;
 .
 .
 .
  case constantN: statement-set N ;
    break;
  default: default-statements;
}
```

Each *statement-set* may be one or many statements long. The **default** portion is optional. The expression controlling the **switch** and all **case** constants must be of integral or character types.

The **switch** works by checking the value of *int-expression* against the constants. As soon as a match is found, that set of statements is executed. If the **break** statement is omitted, execution will continue into the next **case**. You can think of the **cases** as labels. Execution will continue until a **break** statement is found or the **switch** ends.

The following example can be used to process a menu selection:

```
ch = getche();

switch(ch) {
  case 'e': enter();
   break;
  case 'l': list();
    break;
  case 's': sort();
    break;
  case 'q': exit(0);
    break;
  default: printf("Unknown Command\n");
    printf("Try Again\n");
```

typedef

The **typedef** statement allows you to create a new name for an existing data type. The general form of **typedef** is

typedef *type-specifier new-name ;*

For example, to use the word "balance" in place of "float," you would write

```
typedef float balance;
```

union

The **union** keyword creates an aggregate type in which two or more variables share the same memory location. The form of the declaration and the way a member is accessed are the same as for **struct**. The general form is

```
union union-name {
   type member1 ;
   type member2 ;
   .
   .
   .
   type member N ;
} variable-list ;
```

unsigned

The **unsigned** type modifier tells the compiler to create a variable that holds only unsigned (i.e., positive) values. For example, to declare **big** to be an unsigned integer you would write

```
unsigned int big;
```

void

The **void** type specifier is primarily used to declare **void** functions (functions that do not return values). It is also used to create **void** pointers (pointers to **void**) that are generic pointers capable of pointing to any type of object and to specify an empty parameter list.

volatile

The **volatile** modifier tells the compiler that a variable may have its contents altered in ways not explicitly defined by the program. Variables that are changed by the hardware, such as real-time clocks, interrupts, or other inputs are examples.

while

The **while** loop has the general form:

```
while(condition) {
  statement block
}
```

If a single statement is the object of the **while**, the braces may be omitted. The loop will repeat as long as the *condition* is true.

The **while** tests its *condition* at the top of the loop. Therefore, if the *condition* is false to begin with, the loop will not execute at all. The *condition* may be any expression.

An example of a **while** follows. It reads characters until end-of-file is encountered.

```
t = 0;

while(!feof(fp)) {
  s[t] = getc(fp);
  t++;
}
```

C

Building a Windows Skeleton

C is a popular language for Windows programming. As such, it makes sense that some coverage of this important topic be included in this book. But be forewarned: Programming for Windows requires a thorough knowledge of both C and Windows. Frankly, before you can write useful Windows programs, you will need to hone your C programming skills and then invest substantial time in learning the ins and outs of the Windows operating system. Keep in mind that just a description of the functions available within Windows requires approximately 2,000 printed pages!

The preceding notwithstanding, if you will be moving on to Windows programming, you are probably anxious to begin. The purpose of this appendix is to give you a brief overview of Windows programming and to explain a few of its most fundamental elements. In essence, the information presented here is designed to give you a "jump start" into the world of Windows programming.

This appendix discusses in a general way what Windows is, how a program must interact with it, and what rules must be followed by every Windows application. It also develops an application skeleton that you can use as a basis for your own Windows programs. As you will see, all Windows programs share several common traits. It is these shared attributes that will be contained in the application skeleton.

WHICH VERSION OF WINDOWS?

At the time of this writing, there are three versions of the Windows operating system in common use: Windows 3.1, Windows 95, and Windows NT. The skeleton developed in this appendix is designed for 32-bit versions of Windows, such as Windows 95 or Windows NT, since these are the most widely used versions. However, the basic principles apply to all versions of Windows.

WINDOWS PROGRAMMING PERSPECTIVE

The goal of Windows is to enable a person who has basic familiarity with the system to sit down and run virtually any application without prior training. To accomplish this end, Windows provides a consistent interface to the user. In theory, if you can run one Windows-based program, you can run them all. Of course, in actuality, most useful programs will still require some sort of training in order to be used effectively, but at least this instruction can be restricted to *what* the program *does*, not *how* the user must *interact* with it. In fact, much of the code in a Windows application is there just to support the user interface.

Before continuing, it must be stated that not every program that runs under Windows will necessarily present the user with a Windows-style interface. It is possible to write Windows programs that do not take advantage of the Windows interface elements. To create a Windows-style program, you must purposely do so. Only those programs written to take advantage of Windows will look and feel like Windows programs. While you can override the basic Windows design philosophy, you had better have a good reason to do so, because the users of your programs will, most likely, be very disappointed. In general, any application programs you are writing for Windows should utilize the normal Windows interface and conform to the standard Windows design practices.

Windows is graphics-oriented, which means that it provides a Graphical User Interface (GUI). While graphics hardware and video modes are quite diverse, many of the differences are handled by Windows. This means that, for the most part, your program does not need to worry about what type of graphics hardware or video mode is being used.

Let's look at a few of the more important features of Windows.

THE DESKTOP MODEL

With few exceptions, the point of a window-based user interface is to provide the equivalent of a desktop on the screen. On a desk you might find several different pieces of paper, one on top of another, often with fragments of different pages visible beneath the top page. The equivalent of the desktop in Windows is the screen. The pieces of paper are represented by windows on the screen. On a desk you may move pieces of paper about, maybe switching which piece of paper is on top, or how much of another is exposed to view. Windows allows the same type of operations on its windows. By selecting a window, you can make it current, which means putting it on top of all the other open windows. You can enlarge or shrink a window, or move it about on the screen. In short, Windows lets you control the surface of the screen the way you control the items on your desk.

While the desktop model forms the foundation of the Windows user interface, Windows is not limited by it. In fact, several Windows interface elements emulate other types of familiar devices, such as slider controls, spin controls, property sheets, and toolbars. Windows gives you, the programmer, a large array of features from which you may choose those most appropriate to your specific application.

THE MOUSE

Windows allows the use of the mouse for almost all control, selection, and drawing operations. Of course, to say that it *allows* the use of the mouse is an understatement. The fact is that the Windows interface was *designed for the mouse*—it *allows* the use of the keyboard! Although it is certainly possible for an application program to ignore the mouse, it does so only in violation of a basic Windows design principle.

ICONS AND BITMAPS

Windows encourages the use of icons and bitmaps (graphics images). The theory behind the use of icons and bitmaps is found in the old adage "a picture is worth a thousand words."

An icon is a small symbol that represents some operation or program. Generally, the operation or program can be activated by selecting the icon. A bitmap is often used to convey information quickly and simply to the user. However, bitmaps can also be used as menu elements.

MENUS AND DIALOG BOXES

Aside from standard windows, Windows also provides several special-purpose windows. The most common of these are the menu and the dialog box. A *menu* is, as you would expect, a special window that contains choices from which the user makes a selection. The thing that makes menus valuable is that they are largely automated. Instead of having to manage menu selection manually in your program, you simply create a standard menu—Windows will handle the details for you.

A *dialog box* is a special window that allows more complex interaction with the application than that allowed by a menu. For example, your application might use a dialog box to request a file name. With few exceptions, non-menu input is accomplished via a dialog box.

HOW WINDOWS AND YOUR PROGRAM INTERACT

When you write a program for many operating systems, it is your program that initiates interaction with the operating system. For example, in a DOS program, it is the program that requests such things as input and output. Put differently, programs written in the "traditional way" call the operating system. The operating system does not call your program. However, Windows generally works in the opposite way. It is Windows that calls your program. The process works like this: Your program waits until it is sent a *message* by Windows. The message is passed to your program through a special function that is called by Windows. Once a message is received, your program is expected to take an appropriate action. While your program may call Windows when responding to a message, it is still Windows that initiates the activity. More than anything else, it is the message-based interaction with Windows that dictates the general form of all Windows programs.

There are many different types of messages that Windows may send your program. For example, each time the mouse is clicked on a window belonging to your program, a mouse-clicked message will be sent to your program. Another type of message is sent each time a window belonging to your program must be redrawn. Still another message is sent each time the user presses a key when your program is the focus of input. Keep one fact firmly in mind: As far as your program is concerned, messages arrive randomly. This is why Windows programs resemble interrupt-driven programs. You can't know what message will be next.

One final point: Messages sent to your program are stored in a *message queue* associated with your program. Therefore, no message will be lost because your program is busy processing another message. The message will simply wait in the queue until your program is ready for it.

WINDOWS IS MULTITASKING

Since the start, Windows has been a multitasking operating system. This means that it can run two or more programs concurrently. All 32-bit versions of Windows (such as Windows NT and Windows 95) use *preemptive multitasking*. Using this approach, each active application receives a slice of CPU time. It is during its time slice that an application actually executes. When the application's time slice runs out, the next application begins executing. (The previously executing application enters a suspended state in which it awaits another time slice.) In this fashion, each application in the system receives a portion of CPU time. Although the application skeleton developed in this appendix is not concerned with the multitasking aspects of Windows, they will be an important part of any application you create.

Older, 16-bit versions of Windows used a form of multitasking called non-preemptive multitasking. *With this approach, an application retained the CPU until it explicitly released it. This allowed applications to monopolize the CPU and effectively "lock out" other programs. Preemptive multitasking eliminates this problem.*

THE WIN32 API

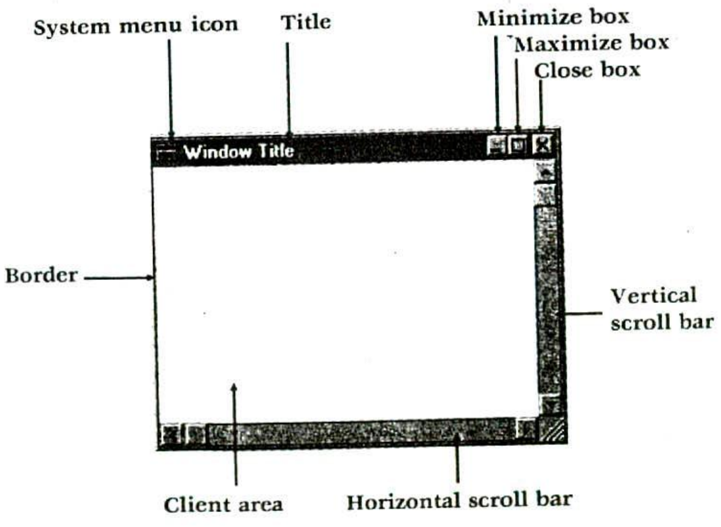
In general, the Windows environment is accessed through a call-based interface called the Application Program Interface (API). The API consists of several hundred functions that your program calls as needed. The API functions provide all the system services performed by Windows. There is a subset to the API called the Graphics Device Interface (GDI), which is the part of Windows that provides device-independent graphics support. It is the GDI functions that make it possible for a Windows application to run on a variety of hardware.

Programs designed for use by 32-bit versions of Windows, such as Windows 95 and Windows NT, use the Win32 API. For the most part, Win32 is a superset of the older Windows 3.1 API (Win16). Indeed, for the most part, the functions are called by the same name and are used in the same way. However, even though similar in spirit and purpose, the two APIs differ because Win32 supports 32-bit addressing while Win16 supports only the 16-bit, segmented-memory model. Because of this difference, several of the older API functions have been widened to accept 32-bit arguments and return 32-bit values. A few API functions have had to be altered to accommodate the 32-bit architecture. API functions have also been added to support preemptive multitasking, new interface elements, and other enhanced features.

Because modern versions of Windows support 32-bit addressing, it makes sense that integers are also 32 bits long. This means that types **int** and **unsigned** are 32 bits long, not 16 bits, as is the case for Windows 3.1. If you want to use a 16-bit integer, it must be declared as **short**. Windows provides portable **typedef** names for these types, as you will see shortly.

THE COMPONENTS OF A WINDOW

Before moving on to specific aspects of Windows programming, a few important terms need to be defined. Figure C-1 shows a standard window with each of its elements pointed out.



All windows have a border that defines the limits of the window; the borders are also used when resizing the window. At the top of the window are several items. On the far left is the system menu icon (also called the title bar icon). Clicking on this box displays the system menu. To the right of the system menu icon is the window's title. At the far right are the minimize, maximize, and close boxes. The client area is the part of the window in which your program activity takes place. Most windows also have horizontal and vertical scroll bars that are used to move information through the window.

SOME WINDOWS APPLICATION BASICS

Before developing the Windows a pplication skeleton, some basic concepts common to all Windows programs need to be discussed.

WinMain( )

All Windows programs begin execution with a call to **WinMain( )**. (Windows programs do not have a **main( )** function.) **WinMain( )** has some special properties that differentiate it from other functions in your application. First, it must be compiled using the **WINAPI** calling convention. (You will see **APIENTRY** used as well. They both mean the same thing.) By default, functions in your C programs use the C calling convention. However, it is possible to compile a function so that it uses a different calling convention; Pascal is a common alternative. For various technical reasons, the calling convention Windows uses to call **WinMain( )** is **WINAPI**. The return type of **WinMain( )** should be **int**.

THE WINDOW FUNCTION

All Windows programs must contain a special function that is *not* called by your program, but is called by Windows. This function is generally referred to as the *window function* or the *window procedure.* The window function is called by Windows when it needs to pass a message to your program. It is through this function that Windows communicates with your program. The window function receives the message in its parameters. All window functions must be declared as returning type **LRESULT CALLBACK**. The type **LRESULT** is a **typedef** that, at the time of this writing, is another name for a long integer. The **CALLBACK** calling convention is used with those functions that will be called by Windows. In Windows terminology, any function that is called by Windows is referred to as a *callback* function.

In addition to receiving the messages sent by Windows, the window function must initiate any actions indicated by a message. Typically, a window function's body consists of a **switch** statement that links a specific response to each message that the program will respond to. Your program need not respond to every message that Windows sends. For messages that your program doesn't care about, you can let Windows provide default processing. Since there are hundreds of different messages that Windows can generate, it is common for most messages simply to be processed by Windows and not by your program.

All messages are 32-bit integer values. Furthermore, all messages are linked with any additional information that the messages require.

WINDOW CLASSES

When your Windows program first begins execution, it will need to define and register a *window class*. When you register a window class, you are telling Windows about the form and function of the window. However, registering the window class does not cause a window to come into existence. To actually create a window requires additional steps.

THE MESSAGE LOOP

As explained earlier, Windows communicates with your program by sending it messages. All Windows applications must establish a *message loop* inside the **WinMain( )** function. This loop reads any pending message from the application's message queue and dispatches that message back to Windows, which then calls your program's window function with that message as a parameter. This may seem to be an overly complex way of passing messages, but it is, nevertheless, the way all Windows programs must function. (Part of the reason for this scheme is to return control to Windows so that the scheduler can allocate CPU time as it sees fit rather than waiting for your application's time slice to end.)

WINDOWS DATA TYPES

As you will soon see, Windows programs do not make extensive use of standard C data types, such as **int** or **char ***. Instead, all data types used by Windows have been **typedef**ed within the WINDOWS.H file and/or its related files. The WINDOWS.H file is supplied by your Windows-compatible compiler and must be included in all Windows programs. Some of the most common types are **HANDLE, HWND, BYTE, WORD, DWORD, UINT, LONG, BOOL**, **LPSTR**, and **LPCSTR**. **HANDLE** is a 32-bit integer that is used as a handle. As you will see, there are a number of handle types, but they are all the same size as **HANDLE**. A *handle* is simply a value that identifies some resource. Also, all handle types begin with an H. For example, **HWND** is a 32-bit integer used as a window handle. **BYTE** is an 8-bit unsigned character. **WORD** is a 16-bit unsigned short integer. **DWORD** is an unsigned long integer. **UINT** is a 32-bit unsigned integer. **LONG** is another name for **long**. **BOOL** is an integer; this type is used to indicate values that are either true or false. **LPSTR** is a pointer to a string, and **LPCSTR** is a **const** pointer to a string.

In addition to the basic types described above, Windows defines several structures. The two that are needed by the skeleton program are **MSG** and **WNDCLASSEX**. The **MSG** structure holds a Windows message, and **WNDCLASSEX** is a structure that defines a window class. These structures will be discussed later in this appendix.

A WINDOWS SKELETON

Now that the necessary background information has been covered, it's time to develop a minimal Windows application. As stated, all Windows programs have certain things in common. This section develops a Windows skeleton that provides these necessary features. In the world of Windows programming, application skeletons are commonly used because there is a substantial "price of admission" when creating a Windows program. For instance, the short example programs shown in this book are designed for a command-line interface (such as DOS), in which a minimal program is about 5 lines long. A minimal Windows program, however, is approximately 50 lines long.

A minimal Windows program contains two functions: **WinMain( )** and the window function. The **WinMain( )** function must perform the following general steps:

1. Define a window class.
2. Register that class with Windows.
3. Create a window of that class.
4. Display the window.
5. Begin running the message loop.

The window function must respond to all relevant messages. Since the skeleton program does nothing but display its window, the only message that it must respond to is the one telling the application that the user has terminated the program.

Before considering the specifics, examine the following program, which is a minimal Windows skeleton. It creates a standard window that includes a title. The window also contains the system menu and is, therefore, capable of being minimized, maximized, moved, resized, and closed. It also contains the standard minimize, maximize, and close boxes.

```
/* A minimal 32-bit Windows skeleton. */

#include <windows.h>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MyWin"; /* name of window class */

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;

  /* Define a window class. */
  wcl.cbSize = sizeof(WNDCLASSEX); /* size of WNDCLASSEX */

  wcl.hInstance = hThisInst; /* handle to this instance */
wcl.lpszClassName = szWinName; /* window class name */
wcl.lpfnWndProc = WindowFunc; /* window function */
wcl.style = 0; /* default style */

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* icon style */
wcl.hIconSm = LoadIcon(NULL, IDI_WINLOGO); /* small icon style */

wcl.hCursor = LoadCursor(NULL, IDC_ARROW); /* cursor style */
wcl.lpszMenuName = NULL; /* no menu */

wcl.cbClsExtra = 0; /* no extra */
wcl.cbWndExtra = 0; /* information needed */

/* Make the window background white. */
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

/* Register the window class. */
if(!RegisterClassEx(&wcl)) return 0;

/* Now that a window class has been registered, a window
   can be created. */
hwnd = CreateWindow(
  szWinName, /* name of window class */
  "Windows Skeleton", /* title */
  WS_OVERLAPPEDWINDOW, /* window style - normal */
  CW_USEDEFAULT, /* X coordinate - let Windows decide */
  CW_USEDEFAULT, /* Y coordinate - let Windows decide */
  CW_USEDEFAULT, /* width - let Windows decide */
  CW_USEDEFAULT, /* height - let Windows decide */
  HWND_DESKTOP, /* no parent window */
  NULL, /* no menu */
  hThisInst, /* handle of this instance of the program */
  NULL /* no additional arguments */
);

/* Display the window. */
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);

/* Create the message loop. */
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg); /* translate keyboard messages */
    DispatchMessage(&msg); /* return control to Windows */
  }
  return msg.wParam;
}

/* This function is called by Windows and is passed
   messages from the message queue.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  switch(message) {
    case WM_DESTROY: /* terminate the program */
      PostQuitMessage(0);
      break;
    default:
      /* Let Windows process any messages not specified in
         the preceding switch statement. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
  return 0;
}
```

The window produced by this program is shown in Figure C-2. Now let's go through this program step by step.

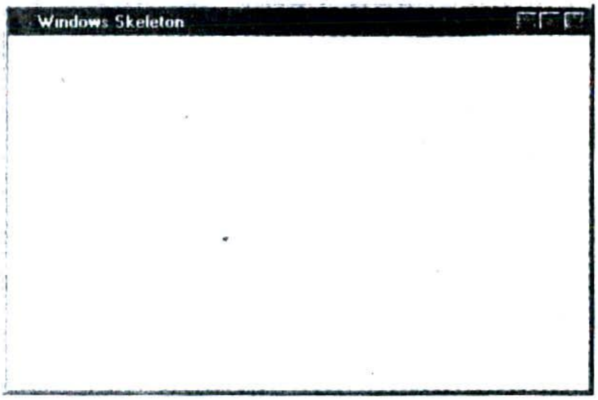
First, all Windows programs must include the header file WINDOWS.H. As stated, this file (along with its support files) contains the API function prototypes and various types, macros, and definitions used by Windows. For example, the data types **HWND** and **WNDCLASSEX** are defined in WINDOWS.H.

The window function used by the program is called **WindowFunc( )**. It is declared as a callback function, because this is the function that Windows calls to communicate with the program.

Program execution begins with **WinMain( )**, which is passed four parameters. **hThisInst** and **hPrevInst** are handles. **hThisInst** refers to the current instance of the program. Remember, Windows is a multitasking system, so more than one instance of your program may be running at the same time. **hPrevInst** will always be **NULL**. (In

FIGURE C-2

The window produced by the Windows skeleton



Windows 3.1 programs, **hPrevInst** would be non-zero if there were other instances of the program currently executing, but this doesn't apply to 32-bit versions of Windows.) The **lpszArgs** parameter is a pointer to a string that holds any command line arguments specified when the application was begun. The **nWinMode** parameter contains a value that determines how the window will be displayed when your program begins execution.

Inside the function, three variables are created. The **hwnd** variable will hold the handle to the program's window. The **msg** structure variable will hold window messages, and the **wcl** structure variable will be used to define the window class.

DEFINING THE WINDOW CLASS

The first two actions that **WinMain( )** takes are to define a window class and then register it. A window class is defined by filling in the fields defined by the **WNDCLASSEX** structure. Its fields are shown here:

```
UINT cbSize; /* size of the WNDCLASSEX structure */
UINT style; /* type of window */
WNDPROC lpfnWndProc; /* address to window func */
```

```
int cbClsExtra; /* extra class info */
int cbWndExtra; /* extra window info */
HINSTANCE hInstance; /* handle of this instance */
HICON hIcon; /* handle of standard icon */
HICON hIconSm; /* handle of small icon */
HCURSOR hCursor; /* handle of mouse cursor */
HBRUSH hbrBackground; /* background color */
LPCSTR lpszMenuName; /* name of main menu */
LPCSTR lpszClassName; /* name of window class */
```

As you can see by looking at the program, **cbSize** is assigned the size of the **WNDCLASSEX** structure. The **hInstance** field is assigned the current instance handle as specified by **hThisInst**. The name of the window class is pointed to by **lpszClassName**, which points to the string "MyWin" in this case. The address of the window function is assigned to **lpfnWndProc**. No default style is specified, and no extra information is needed.

All Windows applications need to define a default shape for the mouse cursor and for the application's icons. An application can define its own custom version of these resources or it may use one of the built-in styles, as the skeleton does. In either case, handles to these resources must be assigned to the appropriate members of the **WNDCLASSEX** structure. To see how this is done, let's begin with icons.

A modern Windows application has at least two icons associated with it: one standard size and one small. The small icon is used when the application is minimized and it is also the icon that is used for the system menu. The standard icon is displayed when you move or copy an application to the desktop. Typically, standard icons are 32 by 32 bitmaps and small icons are 16 by 16 bitmaps. The style of each icon is loaded by the API function **LoadIcon( )**, whose prototype is shown here:

HICON LoadIcon(HINSTANCE *hInst*, LPCSTR *lpszName*);

This function returns a handle to an icon. Here, *hInst* specifies the handle of the module that contains the icon and the icon's name is specified in *lpszName*. However, to use one of the built in icons, you must use **NULL** for the first parameter and specify one of the following macros for the second:

Icon Macro	Shape
IDI_APPLICATION	Default icon
IDI_ASTERISK	Information icon
IDI_EXCLAMATION	Exclamation point icon
IDI_HAND	Stop sign
IDI_QUESTION	Question mark icon
IDI_WINLOGO	Windows Logo

In the skeleton, **IDI_APPLICATION** is used for the standard icon and **IDI_WINLOGO** is used for the small icon.

To load the mouse cursor, use the **LoadCursor( )** API function. This function has the following prototype:

```
HCURSOR LoadCursor(HINSTANCE hInst, LPCSTR lpszName);
```

This function returns a handle to a cursor resource. Here, *hInst* specifies the handle of the module that contains the mouse cursor, and the name of the mouse cursor is specified in *lpszName*. However, to use one of the built-in cursors, you must use **NULL** for the first parameter and specify one of the built-in cursors, using its macro, for the second parameter. Some of the most common built-in cursors are shown here:

Cursor Macro	Shape
IDC_ARROW	Default arrow pointer
IDC_CROSS	Cross hairs
IDC_IBEAM	Vertical I-beam
IDC_WAIT	Hourglass

The background color of the window created by the skeleton is specified as white, and a handle to this *brush* is obtained using the API function **GetStockObject( )**. A brush is a resource that paints the screen using a predetermined size, color, and pattern. The function **GetStockObject( )** is used to obtain a handle to a number of standard display objects, including brushes, pens (which draw lines), and character fonts. It has this prototype:

```
HGDIOBJ GetStockObject(int object);
```

The function returns a handle to the object specified by *object*. (The type **HGDIOBJ** is a GDI handle.) Here are some of the built-in brushes available to your program:

Brush Macro	Background Type
BLACK_BRUSH	Black
DKGRAY_BRUSH	Dark gray
HOLLOW_BRUSH	See-through window
LTGRAY_BRUSH	Light gray
WHITE_BRUSH	White

You can use these macros as parameters to **GetStockObject( )** to obtain a brush.

Once the window class has been fully specified, it is registered with Windows using the API function **RegisterClassEx( )**, whose prototype is shown here:

```
ATOM RegisterClassEx(CONST WNDCLASS *lpWClass);
```

The function returns a value that identifies the window class. **ATOM** is a **typedef** that means **WORD**. Each window class is given a unique value. *lpWClass* must be the address of the **WNDCLASSEX** structure.

CREATING A WINDOW

Once a window class has been defined and registered, your application can actually create a window of that class using the API function **CreateWindow( )**, whose prototype is shown here:

```
HWND CreateWindow(
  LPCSTR lpClassName, /* name of window class */
  LPCSTR lpWinName, /* title of window */
  DWORD dwStyle, /* type of window */
  int X, int Y, /* upper-left coordinates */
```

```
    int Width, int Height, /* dimensions of window */
    HWND hParent, /* handle of parent window */
    HMENU hMenu, /* handle of main menu */
    HINSTANCE hThisInst, /* handle of creator */
    LPVOID lpszAdditional /* pointer to additional info */
  );
```

As you can see by looking at the skeleton program, many of the parameters to **CreateWindow( )** may be defaulted or specified as **NULL**. In fact, most often the *X, Y, Width*, and *Height* parameters will simply use the macro **CW_USEDEFAULT**, which tells Windows to select an appropriate size and location for the window. If the window has no parent, which is the case in the skeleton, then *hParent* must be specified as **HWND_DESKTOP**. (You may also use **NULL** for this parameter.) If the window does not contain a main menu, then *hMenu* must be **NULL**. Also, if no additional information is required, as is most often the case, then *lpszAdditional* is **NULL**. (The type **LPVOID** is **typedef**ed as **void ***. Historically, **LPVOID** stands for "long pointer to **void**.")

The remaining four parameters must be set explicitly by your program. First, *lpszClassName* must point to the name of the window class. (This is the name you gave it when it was registered.) The title of the window is a string pointed to by *lpszWinName*. This can be a null string, but usually a window will be given a title. The style (or type) of window actually created is determined by the value of *dwStyle*. The macro **WS_OVERLAPPEDWINDOW** specifies a standard window that has a system menu, a border, and minimize, maximize, and close boxes. While this style of window is the most common, you can construct one to your own specifications. To accomplish this, simply OR together the various style macros that you want. Some other common styles are shown here:

Style Macros	**Window Feature**
WS_OVERLAPPED	Overlapped window with border
WS_MAXIMIZEBOX	Maximize box
WS_MINIMIZEBOX	Minimize box
WS_SYSMENU	System menu
WS_HSCROLL	Horizontal scroll bar
WS_VSCROLL	Vertical scroll bar

The *hThisInst* parameter must contain the current instance handle of the application.

The **CreateWindow( )** function returns the handle of the window it creates or **NULL** if the window cannot be created.

Once the window has been created, it still is not displayed on the screen. To cause the window to be displayed, call the **ShowWindow( )** API function. This function has the following prototype:

```
BOOL ShowWindow(HWND hwnd, int nHow);
```

The handle of the window to display is specified in *hwnd*. The display mode is specified in *nHow*. The first time the window is displayed, you will want to pass **WinMain( )**'s **nWinMode** as the *nHow* parameter. Remember, the value of **nWinMode** determines how the window will be displayed when the program begins execution. Subsequent calls can display (or remove) the window as necessary. Some common values for *nHow* are shown here:

Display Macros	Effect
SW_HIDE	Removes the window
SW_MINIMIZE	Minimizes the window into an icon
SW_MAXIMIZE	Maximizes the window
SW_RESTORE	Returns a window to normal size

The **ShowWindow( )** function returns the previous display status of the window. If the window was displayed, then nonzero is returned. If the window was not displayed, zero is returned.

Although not technically necessary for the skeleton, a call to **UpdateWindow( )** is included because it is needed by virtually every Windows application that you will create. It essentially tells Windows to send a message to your application that the main window needs to be updated.

THE MESSAGE LOOP

The final part of the skeletal **WinMain( )** is the *message loop*. The message loop is a part of all Windows applications. Its purpose is to receive and process messages sent by Windows. When an application is running, it is continually being sent messages. These messages are stored in the application's message queue until they can be read and processed. Each time your application is ready to read another message, it must call the API function **GetMessage( )**, which has this prototype

BOOL GetMessage(LPMSG *msg*, HWND *hwnd*, UINT *min*, UINT *max*);

The message will be received by the structure pointed to by *msg*. All Windows messages are contained in a structure of type **MSG**, shown here:

```
/* Message structure */
typedef struct tagMSG
{
  HWND hwnd;       /* window that message is for */
  UINT message;    /* message */
  WPARAM wParam;   /* message-dependent info */
  LPARAM lParam;   /* more message-dependent info */
  DWORD time;      /* time message posted */
  POINT pt;        /* X,Y location of mouse */
} MSG;
```

In **MSG**, the handle of the window for which the message is intended is contained in **hwnd**. All Win32 messages are 32-bit integers, and the message is contained in **message**. Additional information relating to each message is passed in **wParam** and **lParam.** The type **WPARAM** is a **typedef** for **UINT,** and **LPARAM** is a **typedef** for **LONG**.

The time the message was sent (posted) is specified in milliseconds in the **time** field.

The **pt** member will contain the coordinates of the mouse when the message was sent. The coordinates are held in a **POINT** structure, which is defined like this:

```
typedef struct tagPOINT {
  LONG x, y;
} POINT;
```

If there are no messages in the application's message queue, then a call to **GetMessage( )** will pass control back to Windows.

The *hwnd* parameter to **GetMessage( )** specifies the window for which messages will be obtained. It is possible, and even likely, that an application will contain several windows, but you only want to receive messages for a specific window. If you want to receive all messages directed at your application, this parameter must be **NULL**.

The remaining two parameters to **GetMessage( )** specify a range of messages that will be received. Generally, you want your application to receive all messages. To accomplish this, specify both *min* and *max* as 0, as the skeleton does.

GetMessage( ) returns zero when the user terminates the program, causing the message loop to terminate. Otherwise it returns nonzero.

Inside the message loop, two functions are called. The first is the API function **TranslateMessage( )**. This function translates raw keyboard input into character messages. Although it is not necessary for all applications, most applications call **TranslateMessage( )** because it is needed to allow full integration of the keyboard into your application program.

Once the message has been read and translated, it is dispatched back to Windows using the **DispatchMessage( )** API function. Windows then holds this message until it can be passed to the program's window function.

Once the message loop terminates, the **WinMain( )** function ends by returning the value of **msg.wParam** to Windows. This value contains the return code generated when your program terminates.

THE WINDOW FUNCTION

The second function in the application skeleton is its window function. In this case, the function is called **WindowFunc( )**, but it could have any name you like. The window function is passed the first four members of the **MSG** structure as parameters. For the skeleton, the only parameter used is the message itself. However, actual applications will use the other parameters to this function.

The skeleton's window function responds to only one message explicitly: **WM_DESTROY**. This message is sent when the user terminates the program. When this message is received, your program must execute a call to the API function **PostQuitMessage( )**. The argument to this function is an exit code that is returned in **msg.wParam** inside **WinMain( )**. Calling **PostQuitMessage( )** causes a **WM_QUIT** message to be sent to your application, which causes **GetMessage( )** to return false, thus stopping your program.

Any other messages received by **WindowFunc( )** are passed to Windows, via a call to **DefWindowProc( )**, for default processing. This step is necessary because all messages must be dealt with in one fashfon or another.

A SHORT WORD ABOUT DEFINITION FILES

You may have heard or read about *definition files*. For 16-bit versions of Windows, such as 3.1, programs need to have a definition file associated with them. A definition file is simply a text file that specifies certain information and settings required by a Windows 3.1 program. However, because of the 32-bit architecture (and other improvements) of modern versions of Windows, definition files are no longer needed.

NAMING CONVENTIONS

Before concluding this appendix, a short comment on the naming of functions and variables needs to be made. Several of the variable and parameter names in the skeleton program and its description probably seemed rather unusual. This is because they follow a set of naming conventions that was invented for Windows programming by Microsoft. For functions, the name consists of a verb followed by a noun. The first character of the verb and noun is capitalized.

For variable names, Microsoft chose to use a rather complex system of embedding the data type into the name. To accomplish this, a lowercase type prefix is added to the start of the variable's name. The name itself begins with a capital letter. The type prefixes are shown in Table C-1. Frankly, the use of type prefixes is controversial and is not universally supported. Many Windows programmers use this method, but many do not. You are free to use any naming convention you like.

TO LEARN MORE

The foregoing overview of Windows programming just scratches the surface. In order to write Windows programs that are useful, you must learn much more about Windows programming. To learn more about Windows 95 programs you will want to read the following books:

Schildt's Windows 95 Programming in C and C++

Schildt's Advanced Windows 95 Programming in C and C++

Prefix	Data Type
b	Boolean (one byte)
c	Character (one byte)
dw	Long unsigned integer
f	16-bit bit-field (flags)
fn	Function
h	Handle
l	Long integer
lp	Long pointer
n	Short integer
p	Pointer
pt	Long integer holding screen coordinates
w	Short unsigned integer
sz	Pointer to null-terminated string
lpsz	Long pointer to null-terminated string
rgb	Long integer holding RGB color values

TABLE C-1 *Variable Type Prefix Characters* ▼

To learn more about Windows NT programming, you will find

Windows NT 4 Programming From the Ground Up

especially useful. These books are written by Herbert Schildt and published by Osborne/McGraw-Hill.

D

Answers

CHAPTER 1

1.3 EXERCISES

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

   int main(void)
   {
     int num;

     num = 1000;
     printf("%d is the value of num", num);

     return 0;
   }
```

1.4 EXERCISES

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

   int main(void)
   {
     float a, b;

     printf("Enter two numbers: ");
     scanf("%f", &a);
     scanf("%f", &b);
     printf("Their sum is %f.", a+b);

     return 0;
   }
```

1.5 EXERCISES

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

   int main(void)
   {
     int len, width, height;

     printf("En er length: ");
```

```
    scanf("%d", &len);

    printf("Enter width: ");
    scanf("%d", &width);

    printf("Enter height: ");
    scanf("%d", &height);

    printf("Volume is %d.", len * width * height);

    return 0;
  }
```

2.
```
  #include <stdio.h>

  int main(void)
  {
    printf("Number of seconds in a year: ");
    printf("%f", 60.0 * 60.0 * 24.0 * 365.0);

    return 0;
```

1.6 EXERCISES

2. Yes, a comment can contain nothing.
3. Yes, you can temporarily remove a line of code from your program by making it into a comment. This is sometimes called "commenting out" a line of code.

1.7 EXERCISES

2.
```
  #include <stdio.h>

  void one(void);
  void two(void);

  int main(void)
  {
    one();
    two();
    return 0;
  }

  void one(void)
  {
    printf("The summer soldier, ");
  }

  void two(void)
  {
    printf("the sunshine patriot.");
  }
```

3. The compiler will report an error. The prototype is needed in order for the compiler to properly call **func1( )**.

1.8 EXERCISES

2.
```
#include <stdio.h>

int convert(void);

int main(void)
{
  printf("%d", convert());

  return 0;
}

int convert(void)
{
  int dollars;

  printf("Enter number of dollars: ");
  scanf("%d", &dollars);
  return dollars / 2;
}
```

3. There is nothing technically wrong with the program. However, function **f1( )** returns an integer value, but it is being assigned to a variable of type **double**. This would lead one to suspect that perhaps the programmer has misunderstood the purpose of the **f1( )** function.

4. A function declared with a **void** return type cannot return a value.

EXERCISES

1.
```
#include <stdio.h>

void outnum(int num);

int main(void)
{
  outnum(10);

  return 0;
}

void outnum(int num)
{
  printf("%d", num);
}
```

2. The **sqr_it( )** function requires an integer argument, but it is called with a floating-point value.

MASTERY SKILLS CHECK

1.
```
#include <stdio.h>

int main(void)
{
  float weight;

  printf("Enter your weight: ");
  scanf("%f", &weight);
  printf("Effective moon weight: %f", weight * 0.17);

  return 0;
}
```

2. The comment is not terminated with a */.

3.
```
#include <stdio.h>
```

```
int o_to_c(int o);

int main(void)
{
  int ounces;
  int cups;

  printf("Enter ounces: ");
  scanf("%d", &ounces);

  cups = o_to_c(ounces);
  printf("%d cups", cups);

  return 0;
}

int o_to_c(int o)
{
  return o / 8;
}
```

4. **char**, **int**, **float**, **double**, and **void**.
5. The variable names are wrong because
   a. A dash may not be used in a variable name.
   b. A dollar sign may not be used in a variable name.
   c. A + sign may not be used in a variable name.
   d. A digit may not begin a variable name.

CHAPTER 2

REVIEW SKILLS CHECK

1. All programs must have a **main( )** function. This is the first function called when your program begins executing.
2.
```
#include <stdio.h>

int main(void)
{
  printf("This is the number %d", 100);
  return 0;
}
```

3. To include a header file, use the **#include** compiler directive. For example,

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

   includes the STDIO.H header.

4. The five basic data types are **char**, **int**, **float**, **double**, and **void**.
5. The invalid variable names are b, c, and e.
6. The **scanf( )** function is used to input information from the keyboard.

7.
   ```
   #include <stdio.h>

   int main(void)
   {
     int i;

     printf("Enter a number: ");
     scanf("%d", &i);
     printf("%d", i*i);

     return 0;
   }
   ```

8. Comments must be surrounded by the /* and */ comment symbols. For example, this is a valid C comment.

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

9. A function returns a value to the calling routine using **return**.
10. `void Myfunc(int count, float balance, char ch);`

EXERCISES

1. b, d, and e are true.

2.
   ```
   #include <stdio.h>

   int main(void)
   {
     int i;
   ```

```
  printf("Enter a number: ");
  scanf("%d", &i);
  if((i%2)==0) printf("Even");
  if((i%2)==1) printf("Odd");

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int a, b, op;

  printf("Enter first number: ");
  scanf("%d", &a);

  printf("Enter second number: ");
  scanf("%d", &b);

  printf("Enter 0 to add, 1 to multiply: ");
  scanf("%d", &op);

  if(op==0) printf("%d", a+b);
  else printf("%d", a*b);

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int i;

  printf("Enter a number: ");
  scanf("%d", &i);
  if((i%2)==0) printf("Even");
  else printf("Odd");
  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int a, b, op;

  printf("Enter 0 to add, 1 to subtract: ");
  scanf("%d", &op);

  if(op==0) { /* add */
    printf("Enter first number: ");
    scanf("%d", &a);
    printf("Enter second number: ");
    scanf("%d", &b);
    printf("%d", a+b);
  }
  else { /* subtract */
    printf("Enter first number: ");
    scanf("%d", &a);
    printf("Enter second number: ");
    scanf("%d", &b);
    printf("%d", a-b);
  }

  return 0;
}
```

2. No, the opening curly brace is missing.

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=1; i<101; i=i+1) printf("%d ", i);
  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=17; i<101; i=i+1)
    if((i%17)==0) printf("%d ", i);

  return 0;
}
```

3.
```
#include <stdio.h>

int main(void)
{
  int num, i;

  printf("Enter the number to test: ");
  scanf("%d", &num);

  for(i=2; i<(num/2)+1; i=i+1)
    if((num%i)==0) printf("%d ", i);

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=1; i<101; i++) printf("%d ", i);

  return 0;
}
```

```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=17; i<101; i++)
    if((i%17)==0) printf("%d ", i);

  return 0;
}
```

```
#include <stdio.h>

int main(void)
{
  int num, i;

  printf("Enter the number to test: ");
  scanf("%d", &num);

  for(i=2; i<(num/2)+1; i++)
    if((num%i)==0) printf("%d ", i);

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int a, b;

  a = 1;
  a++;
  b = a;
  b--;
  printf("%d %d", a, b);

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=1; i<11; i++)
    printf("%d %d %d\n", i, i*i, i*i*i);

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int i, j;

  printf("Enter a number: ");
  scanf("%d", &i);

  for(j=i; j>0; j--) printf("%d\n", j);
  printf("\a");

  return 0;
}
```

EXERCISES

1. The loop prints the numbers **0** through **99**.
2. Yes.
3. No, the first is true, the second is false.

MASTERY SKILLS CHECK

1.
```
#include <stdio.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */
  int i;

  magic = 1325;
  guess = 0;

  for(i=0; i<10 && guess!=magic; i++) {
    printf("Enter your guess: ");
    scanf("%d", &guess);

    if(guess == magic) {
      printf("RIGHT!");
      printf(" %d is the magic number.\n", magic);
    }
    else {
      printf("...Sorry, you're wrong...");
      if(guess > magic)
        printf(" Your guess is too high.\n");
      else printf(" Your guess is too low.\n");
    }
  }
  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int rooms, len, width, total;
  int i;

  printf("Number of rooms? ");
  scanf("%d", &rooms);

  total = 0;
  for(i=rooms; i>0; i--) {
    printf("Enter length: ");
    scanf("%d", &len);

    printf("Enter width: ");
    scanf("%d", &width);

    total = total + len * width;
  }
  printf("Total square footage: %d", total);
  return 0;
}
```

3. The increment operator increases a variable by one and the decrement operator decreases a variable by one.

4.
```
#include <stdio.h>

int main(void)
{
  int answer, count;
  int right, wrong;

  right = 0;
  wrong = 0;

  for(count=1; count < 11; count=count+1) {
    printf("What is %d + %d? ", count, count);
    scanf("%d", &answer);

    if(answer == count+count) {
      printf("Right!  ");
      right++;
    }
    else {
      printf("Sorry, you're wrong. ");
      printf("The answer is %d.  ", count+count);
      wrong++;
    }
  }
  printf("You got %d right and %d wrong.", right, wrong);

  return 0;
}
```

5.
```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=1; i<=100; i++) {
    printf("%d\t", i);
```

```
      if((i%5)==0) printf("\n");
    }

    return 0;
  }
```

CHAPTER 3

REVIEW SKILLS CHECK

1. C's relational and logical operators are <, >, <=, >=, !=, ==, !, &&, and ||.
2. A block of code is a group of logically connected statements. To make a block, surround the statements with curly braces.
3. To output a newline, use the **\n** backslash character code.

4.
   ```
   #include <stdio.h>

   int main(void)
   {
     int i;

     for(i=-100; i<101; i++) printf("%d ", i);

     return 0;
   }
   ```

5.
   ```
   #include <stdio.h>

   int main(void)
   {
     int i;

     printf("Enter proverb number: ");
     scanf("%d", &i);

     if(i==1) printf("A bird in the hand...");
     if(i==2) printf("A rolling stone...");
     if(i==3) printf("Once burned, twice shy.");
     if(i==4) printf("Early to bed, early to rise...");
     if(i==5) printf("A penny saved is a penny earned.");
   ```

```
  return 0;
}
```

6.
```
count++;
/* or */
++count;
```

7. In C, true is any nonzero value. False is zero.

EXERCISES

1.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  int i;
  char ch, smallest;

  printf("Enter 10 letters.\n");

  smallest = 'z' ; /* make largest to begin with */

  for(i=0; i<10; i++) {
    ch = getche();
    if(ch < smallest) smallest = ch;
  }
  printf("\nThe smallest character is %c.", smallest);

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  char ch;

  for(ch='A'; ch<='Z'; ch++)
    printf("%d ", ch);

  printf("\n");

  for(ch='a'; ch<='z'; ch++)
```

```
    printf("%d ", ch);

  return 0;
}
```

The codes differ by 32.

EXERCISES

1. The **else** relates to the first **if**; it is not in the same block as the second.

2.
```
#include <stdio.h>

int main(void)
{
  char ch;
  int s1, s2;
  float radius;

  printf("Compute area of Circle, Square, or Triangle? ");
  ch = getchar();
  printf("\n");

  if(ch=='C') {
    printf("Enter radius of circle: ");
    scanf("%f", &radius);
    printf("Area is: %f", 3.1416*radius*radius);
  }
  else if(ch=='S') {
    printf("Enter length of first side: ");
    scanf("%d", &s1);
    printf("Enter length of second side: ");
    scanf("%d", &s2);
    printf("Area is: %d", s1*s2);
  }
  else if(ch=='T') {
    printf("Enter length of base: ");
    scanf("%d", &s1);
    printf("Enter height: ");
    scanf("%d", &s2);
    printf("Area is: %d", (s1*s2)/2);
  }
```

```
    return 0;
  }
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  float dist, speed;
  int num;

  printf("Enter number of drive time computations: ");
  scanf("%d", &num);

  for(; num; num-- ) {
    printf("\nEnter distance: ");
    scanf("%f", &dist);

    printf("Enter average speed: ");
    scanf("%f", &speed);

    printf("Drive time is %f\n", dist/speed);
  }

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int i;

  printf("Enter a number: ");

  scanf("%d", &i);

  for( ; i; i--) ;

  printf("\a");
  return 0;
}
```

3.
```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=1; i<1001; i=i+i) printf("%d ", i);

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  float dist, speed;
  int num;

  printf("Enter number of drive time computations: ");
  scanf("%d", &num);

  while(num) {
    printf("\nEnter distance: ");
    scanf("%f", &dist);

    printf("Enter average speed: ");
    scanf("%f", &speed);

    printf("Drive time is %f\n", dist/speed);

    num--;
  }

  return 0;
}
```

2.
```
#include <stdio.h>
#include <conio.h>
```

```
int main(void)
{
  char ch;

  printf("Enter your encoded message.\n");

  ch = getche();
  while(ch!='\r') {
    printf("%c", ch-1);
    ch = getche();
  }

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  float gallons;

  printf("\nEnter gallons: ");
  scanf("%f", &gallons);

  do {
    printf("Liters: %f\n", gallons*3.7854);

    printf("Enter gallons or 0 to quit. ");
    scanf("%f", &gallons);

  } while(gallons!=0);

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int choice;

  printf("Mailing list menu:\n\n");
  printf("  1. Enter addresses\n");
  printf("  2. Delete addresses\n");
  printf("  3. Search the list\n");
  printf("  4. Print the list\n");
  printf("  5. Quit\n");

  do {
    printf("Enter the number of the choice (1-5): ");
    scanf("%d", &choice);
  } while(choice<1 || choice>5);

  return 0;
```

EXERCISES

1.
```
/* This program finds the prime numbers from
   2 to 1000
*/

#include <stdio.h>

int main(void)
{
  int i, j, prime;

  for(i=2; i<1000; i++)
    prime = 1;
    for(j=2; j <= i/2; j++)
      if(!(i%j)) prime=0;
    if(prime) printf("%d is prime.\n", i);
  }

  return 0;
}
```

2.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  int i;
  char ch;
  for(i=0; i<10; i++) {
    printf("\nEnter a letter: ");
    ch = getche();
    printf("\n");
    for( ; ch; ch--) printf("%c", '.');
  }

  return 0;
}
```

EXERCISES

2.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  float i;
  char ch;

  printf("Tip Computer\n");

  for(i=1.0; i<101.0; i=i+1.0) {
    printf("%f %f %f %f\n", i, i+i*.1, i+i*.15, i+i*.2);
    printf("More? (Y/N) ");
    ch = getche();
    printf("\n");
    if(ch=='N') break;
  }

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=1; i<101; i++) {
    if(!(i%2)) continue;
    printf("%d ", i);
  }

  return 0;
}
```

EXERCISES

1. Floating point values may not be used to control **switch**.

2.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  char ch;
  int digit, punc, letter;

  printf("Enter characters, ENTER to stop.\n");

  digit = 0;
  punc = 0;
  letter = 0;

  do {
    ch = getche();
    switch(ch) {
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
      case '0':
        digit++;
        break;
      case '.':
      case ',':
      case '?':
      case '!':
      case ':':
        case ';':
          punc++;
          break;
        default:
          letter++;
      }
    } while(ch!='\r');
    printf("\nDigits: %d\n", digit);
    printf("Punctuation: %d\n", punc);
    printf("Letters: %d\n", letter);

    return 0;
  }
```

3.10 EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i;

  i = 1;

  jump_label:
    if(i>=11) goto done_label;
    printf("%d ", i);
    i++;
    goto jump_label;
  done_label: printf("Done");

  return 0;
}
```

MASTERY SKILLS CHECK

1.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  char ch;
  printf("Enter lowercase letters. ");
  printf("(Press ENTER to Quit.)\n");
  do {
    ch = getche();
    if(ch!='\r') printf("%c", ch-32);
  } while(ch!='\r');

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int i;

  printf("Enter a number: ");
  scanf("%d", &i);

  if(!i) printf("zero");
  else if(i<0) printf("negative");
  else printf("positive");

  return 0;
}
```

3. The **for** loop is valid. C allows any of its expressions to be empty.

4. `for( ; ; ) ...`

5.
```
/* for */
for(i=1; i<11; i++) printf("%d ", i);

/* do */
i = 1;
do {
  printf("%d ",i);
  i++;
} while(i<11);

/* while */
i=1;
while (i<11) {
  printf("%d ", i);
```

```
    i++;
  }
```

6. The **break** statement causes immediate termination of the loop

7. Yes.

8. No, the label is missing the colon.

CUMULATIVE SKILLS CHECK

1.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  char ch;

  printf("Enter characters (q to quit): \n");
  do {
    ch = getche();
    switch(ch) {
      case '\t': printf("tab\n");
        break;
      case '\b': printf("backspace\n");
        break;
      case '\r': printf("Enter\n");
    }
  } while(ch!='q');

  return 0;
}
```

2.
```
include <stdio.h>

int main(void)
{
  int i, j, k;

  for(k=0; k<10; k++) { /* use increment operator */
    printf("Enter first number: ");
    scanf("%d", &i);

    printf("Enter second number: ");
    scanf("%d", &j);
```

```
      if(j) printf("%d\n", i/j); /* simplify condition */
      else printf("Cannot divide by zero.\n"); /* use else */
    }

    return 0;
  }
```

CHAPTER 4

REVIEW SKILLS CHECK

1.
```
int i;
for(i=1; i<11; i++) printf("%d ", i);

i = 1;
do {
  printf("%d ", i);
  i++;
} while(i<11);

i = 1;
while(i<11){
  printf("%d ", i);
  i++;
}
```

2.
```
switch(ch) {
    case 'L': load();
      break;
    case 'S': save();
      break;
    case 'E': enter();
      break;
    case 'D': display();
      break;
    case 'Q': quit();
      break;
  }
```

3.
```
#include <stdio.h>
#include <conio.h>
```

```
int main(void)
{
  char ch;

  do {
    ch = getche();
  } while(ch!='\r');

  return 0;
}
```

4. The **break** statement causes immediate termination of the loop that contains it. It also terminates a statement sequence in a **switch**.
5. The **continue** statement causes the next iteration of a loop to occur.
6.
```
#include <stdio.h>

int main(void)
{
  int i;
  float feet, meters, ounces, pounds;

  do {
    printf("Convert\n\n");
    printf("1. feet to meters\n");
    printf("2. meters to feet\n");
    printf("3. ounces to pounds\n");
    printf("4. pounds to ounces\n");
    printf("5. Quit\n\n");
    do {
      printf("Enter the number of your choice: ");
      scanf("%d", &i);
    } while(i<0 || i>5);

    switch(i) {
      case 1:
        printf("Enter feet: ");
        scanf("%f", &feet);
        printf("Meters: %f\n", feet / 3.28);
        break;
      case 2:
        printf("Enter meters: ");
        scanf("%f", &meters);
        printf("Feet: %f\n", meters * 3.28);
        break;
      case 3:
        printf("Enter ounces: ");
        scanf("%f", &ounces);
        printf("Pounds: %f\n", ounces / 16);
        break;
      case 4:
        printf("Enter pounds: ");
        scanf("%f", &pounds);
        printf("ounces: %f\n", pounds * 16);
        break;
    }
  } while(i!=5)

  return 0;
}
```

EXERCISES

1. `unsigned short int loc_counter;`

2.
```
#include <stdio.h>

int main(void)
{
  unsigned long int distance;

  printf("Enter distance: ");

  scanf("%lu", &distance);

  printf("%ld seconds", distance / 186000);

  return 0;
}
```

3. The statement can be recoded using C's shorthand as follows

```
short i;
```

EXERCISES

1. Local variables are known only to the function in which they are declared. Global variables are known to and accessible by all functions. Further, local variables are created when the function is entered and destroyed when the function is exited. Thus they cannot maintain their values between function calls. However, global variables stay in existence during the entire lifetime of the program and maintain their values.

2. Here is the non-generalized version.

```
#include <stdio.h>

void soundspeed(void);

double distance;

int main(void)
{
  printf("Enter distance in feet: ");
  scanf("%lf", &distance);
  soundspeed();

  return 0;
}

void soundspeed(void)
{
  printf("Travel time: %f", distance / 1129);
}
```

Here is the parameterized version.

```
#include <stdio.h>

void soundspeed(double distance);

int main(void)
{
  double distance;

  printf("Enter distance in feet: ");
  scanf("%lf", &distance);
  soundspeed(distance);
  return 0;
}

void soundspeed(double distance)
{
  printf("Travel time: %f", distance / 1129);
}
```

EXERCISES

1. To cause a constant to be recognized by the compiler explicitly as a **float**, follow the value with an **F**.

2.
```
#include <stdio.h>

int main(void)
{
  long int i;

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

  return 0;
}
```

3.
```
#include <stdio.h>

int main(void)
{
  printf("%s %s %s", "I", "like", "C");

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i=100;
```

```
    for( ; i>0; i--) printf("%d ", i);

    return 0;
  }
```

2. No. You cannot initialize a global variable using another variable
3. Yes. A local variable can be initialized using any expression valid at the time of the initialization.

EXERCISES

1. The entire expression is **float**.
2. The subexpression is **unsigned long**.

EXERCISES

1. The program displays **10**.
2. The program displays **3.0**.

EXERCISE

1.
```
#include <stdio.h>

int main(void)
{
  float f;

  for(f=1.0; (int) f<=9; f=f + 0.1)
  printf("%f ", f);

  return 0;
}
```

2. Here is the corrected statement.

```
x = (int)123.23 % 3; /* now fixed */
```

MASTERY SKILLS CHECK

1. The data-type modifiers are

   unsigned
   long
   short
   signed

   They are used to modify the base type so that you can obtain variables that best fit the needs of your program.

2. To define an **unsigned** constant, follow the value with a **U**. To define a **long** constant, follow the value with an **L**. To specify a **long double**, follow the value with an **L**.

3. `float balance = 0.0;`

4. When the C compiler evaluates an expression, it automatically converts all **chars** and **shorts** to **int**.

5. A **signed** integer uses the high-order bit as a sign flag. When the bit is set, the number is negative, when it is cleared, the number is positive. An **unsigned** integer uses all bits as part of the number and can represent only positive values.

6. Global variables maintain their values throughout the lifetime of the program. They are also accessible by all functions in the program.

7.
```
#include <stdio.h>

int series(void);

int num = [illegible]

int ma[illegible]
{
  int i;

  for(i=0; i<10; i++)
    printf("%d ", series());

  return 0;
```

```
int series(void)
{
  num = (num*1468) % 467;
  return num;
}
```

8. A type cast temporarily changes the type of a variable. For example, here the **int i** is temporarily changed into a **double**.

   ```
   (double) i
   ```

CUMULATIVE SKILLS CHECK

1. The fragment is not valid because to C, both 'A' and 65 are the same thing, and no two **case** constants can be the same.
2. The reason that the return value of **getchar( )** or **getche( )** can be assigned to a **char** is because C automatically removes the high-order byte.
3. No. Because **i** is a signed integer, its maximum value is 32,767. Therefore, it will never exceed 33,000.

CHAPTER 5

REVIEW SKILLS CHECK

1. A local variable is known only to the function in which it is declared. Further, it is created when the function is entered and destroyed when the function returns. A global variable is known throughout the entire program and remains in existence the entire time the program is executing.
2. C compiler will assign the following types:
   a. **int**
   b. **int**
   c. **double**
   d. **long**
   e. **long**

3.
```
#include <stdio.h>

int main(void)
{
  long l;
  short s;
  double d;

  printf("Enter a long value: ");
  scanf("%ld", &l);

  printf("Enter a short value: ");
  scanf("%hd", &s);

  printf("Enter a double value: ");
  scanf("%lf", &d);

  printf("%ld\n", l);
  printf("%hd\n", s);
  printf("%f\n", d);

  return 0;
}
```

4. A type cast temporarily changes the type of a value.
5. The **else** is associated with the **if(j)** statement, contrary to what the (incorrect) indentation would have you believe.
6. When **i** is 1, **a** is 2. When **i** is 4, **a** is 5.

EXERCISES

1. The array **count** is being overrun. It is only 10 elements long. but the program requires one that is 100 elements long.
2.
```
#include <stdio.h>

int main(void)
{
  int i[10], j, k, match;

  printf("Enter 10 numbers:\n");
  for(j=0; j<10; j++) scanf("%d", &i[j]);
  /* see if any match */
  for(j=0; j<10; j++) {
    match = i[j];
    for(k=j+1; k<10; k++)
      if(match==i[k])
        printf("%d is duplicated\n", match);
  }

  return 0;
}
```

3.
```
#include <stdio.h>

int main(void)
{
  float item[100], t;
  int a, b;
  int count;

  /* read in numbers */
  printf("How many numbers? ");
  scanf("%d", &count);
  for(a=0; a<count; a++) scanf("%f", &item[a]);

  /* now sort them using a bubble sort */
  for(a=1; a<count; ++a)
    for(b=count-1; b>=a; --b) {
      /* compare adjacent elements */
      if(item[b-1] > item[b]) {
        /* exchange elements */
        t = item[b-1];
        item[b-1] = item[b];
        item[b] = t;
      }
    }

  /* display sorted list */
  for(a=0; a<count; a++) printf("%f ", item[a]);

  return 0;
}
```

EXERCISES

1.
```
/* Reverse a string. */
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];
  int i;

  printf("Enter a string: ");
  gets(str);

  for(i=strlen(str)-1; i>=0; i--)
    printf("%c", str[i]);

  return 0;
}
```

2. The string **str** is not long enough to hold the string "this is a test".

3.
```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char bigstr[1000] = "", str[80];

  for( ; ; ) {
    printf("Enter a string: ");
    gets(str);
    if(!strcmp(str, "quit")) break;
    strcat(str, "\n");
    /* prevent an array overrun */
    if(strlen(bigstr)+strlen(str) >= 1000) break;
    strcat(bigstr, str);
  }

  printf(bigstr);

  return 0;
}
```

5.3 EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int three_d[3][3][3];
  int i, j, k, x;

  x = 1;
  for(i=0; i<3; i++)
    for(j=0; j<3; j++)
      for(k=0; k<3; k++) {
        three_d[i][j][k] = x;
        x++;
        printf("%d ", three_d[i][j][k]);
     }

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  int three_d[3][3][3];
  int i, j, k, sum;

  for(i=0; i<3; i++)
    for(j=0; j<3; j++)
      for(k=0; k<3; k++) {
        three_d[i][j][k] = (i+1) * (j+1) * (k+1);
        printf("%d ", three_d[i][j][k]);
      }

  /* sum all elements */
  sum = 0;
  for(i=0; i<3; i++)
    for(j=0; j<3; j++)
      for(k=0; k<3; k++)
        sum = sum + three_d[i][j][k];

  printf("\n%d", sum);
  return 0;
}
```

EXERCISES

1. No. The list must be enclosed between curly braces.
2. No. The array **name** is only 4 characters long. The attempted call to **strcpy( )** will cause the array to be overrun.
3.

```
#include <stdio.h>

int main(void)
{
  int cube[][3] = {
    1, 1, 1,
    2, 4, 8,
    3, 9, 27,
    4, 16, 64,
    5, 25, 125,
    6, 36, 216,
    7, 49, 343,
    8, 64, 512,
    9, 81, 729,
    10, 100, 1000
  };
  int num, i;

  printf("Enter cube: ");
  scanf("%d", &num);

  for(i=0; i<10; i++)
    if(cube[i][2]==num) {
      printf("Root: %d\n", cube[i][0]);
      printf("Square: %d", cube[i][1]);
      break;
    }

  if(i==10) printf("Cube not found.\n");

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  char digits[10][10] = {
    "zero", "one", "two", "three",
    "four", "five", "six", "seven",
    "eight", "nine"
  };
  char num;

  printf("Enter number: ");
  num = getche();
  printf("\n");

  num = num - '0';
  if(num>=0 && num<10) printf("%s", digits[num]);

  return 0;
}
```

MASTERY SKILLS CHECK

1. An array is a list of like-type variables.
2. The statement will not generate an error message because C provides no bounds checking on array operations, but it is wrong because it causes **count** to be overrun.
3.
```
#include <stdio.h>

int main(void)
{
  int stats[20], i, j;
  int mode, count, oldcount, oldmode;

  printf("Enter 20 numbers: \n");
  for(i=0; i<20; i++) scanf("%d", &stats[i]);

  oldcount = 0;
  /* find the mode */
  for(i=0; i<20; i++) {
    mode = stats[i];
    count = 1;

    /* count the occurrences of this value */
    for(j=i+1; j<20; j++)
      if(mode==stats[j]) count++;

    /* if count is greater than old count, use new mode */
    if(count>oldcount) {
      oldmode = mode;
      oldcount = count;
    }
  }
  printf("The mode is %d\n", oldmode);

  return 0;
}
```

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

5.
```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];

  do {
    printf("Enter a string: ");
    gets(str);
  } while(strcmp("quit", str));

  return 0;
}
```

6.
```
/* Computerized dictionary program. */

#include <stdio.h>
#include <string.h>

int main(void)
{
  char dict[][2][40] = {
      "house", "a place of dwelling",
      "car", "a vehicle",
      "computer", "a thinking machine",
      "program", "a sequence of instructions",
      "", ""
    };
    char word[80];
    int i;

    printf("Enter word: ");
    gets(word);

    /* look up the word */
    i = 0;
    /* search while null string not yet encountered */
    while(strcmp(dict[i][0], "" )) {
      if(!strcmp(word, dict[i][0])) {
        printf("meaning: %s", dict[i][1]);
        break;
      }
      i++;
    }
    if(!strcmp(dict[i][0], ""))
    printf("Not in dictionary\n");

    return 0;
  }
```

CUMULATIVE SKILLS CHECK

1.
```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];
  int i;

  printf("Enter a string: ");
  gets(str);

  /* pad the string if necessary */
  for(i=strlen(str); i<79; i++)
    strcat(str, ".");
  printf(str);

  return 0;
}
```

2.
```
/* A simple coding program. */

#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];
  int i, j;

  printf("Enter message: ");
  gets(str);

  /* code it */
  i=0; j = strlen(str) - 1;
  while(i<=j) {
    if(i<j) printf("%c%c", str[i], str[j]);
    else printf("%c", str[i]);
    i++; j--;
  }

  return 0;
}
```

3.
```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];
  int spaces, periods, commas;
  int i;

  printf("Enter a string: ");
  gets(str);

  spaces = 0;
  commas = 0;
  periods = 0;
```

```
    for(i=0; i<strlen(str); i++)
      switch(str[i]) {
        case '.': periods++;
          break;
        case ',': commas++;
          break;
        case ' ': spaces++;
      }

    printf("spaces: %d\n", spaces);
    printf("commas: %d\n", commas);
    printf("periods: %d", periods);

    return 0;
  }
```

4. The **getchar( )** function returns a character, not a string. Hence, it cannot be used as shown. You must use **gets( )** t read a string from the keyboard.

5.
```
/* A simple game of Hangman */

#include <stdio.h>
#include <string.h>

int main(void)
{
  char word[] = "concatenation";
  char temp[] = "-------------";
  char ch;
  int i, count;

  count = 0; /* count number of guesses */

  do {
    printf("%s\n", temp);
    printf("Enter your guess: ");
    ch = getchar();
    printf("\n");

    /* see if letter matches any in word */
    for(i=0; i<strlen(word); i++)
      if(ch==word[i]) temp[i] = ch;
      count++;
  } while(strcmp(temp, word));
  printf("%s\n", temp);
  printf("You guessed the word and used %d guesses", count);

  return 0;
}
```

CHAPTER 6

REVIEW SKILLS CHECK

1.
```
#include <stdio.h>

int main(void)
{
  int num[10], i, even, odd;

  printf("Enter 10 integers: ");

  for(i=0; i<10; i++) scanf("%d", &num[i]);

  even = 0; odd = 0;
  for(i=0;  i< 10; i++) {
    if(num[i]%2) odd = odd + num[i];
    else even = even + num[i];
  }

  printf("Sum of even numbers: %d\n", even);
  printf("Sum of odd numbers: %d", odd);

  return 0;
}
```

2.
```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char pw[80];
  int i;

  for(i=0; i<3; i++) {
    printf("Password: ");
```

```
      gets(pw);
      if(!strcmp("Tristan", pw)) break;
    }

    if(i==3) printf("Access Denied");
    else printf("Log-on Successful");

    return 0;
  }
```

3. The array, **name**, is not big enough to hold the string being assigned to it.
4. A null string is a string that contains only the null character.
5. The **strcpy( )** function copies the contents of one string into another. The **strcmp( )** function compares two strings and returns less than zero if the first string is less than the second, zero if the strings match, or greater than zero if the first string is greater than the second.
6.

```
/* A Simple computerized telephone book. */

#include <stdio.h>
#include <string.h>

char phone[][2][40] = {
  "Fred", "555-1010",
  "Barney", "555-1234",
  "Ralph", "555-2347",
  "Tom", "555-8396",
  "", ""
};

int main(void)
{
  char name[80];
  int i;

  printf("Name? ");
  gets(name);

  for(i=0; phone[i][0][0]; i++)
    if(!strcmp(name, phone[i][0]))
      printf("number: %s", phone[i][1]);
  return 0;
}
```

EXERCISES

1. A pointer is a variable that contains the address of another variable.
2. The pointer operators are the * and the &. The * operator returns the value of the object pointed to by the pointer it precedes. The & operator returns the address of the variable it precedes.
3. The base type of a pointer is important because all pointer arithmetic is done relative to it.
4.
```
#include <stdio.h>

int main(void)
{
  int i, *p;

  p = &i;

  for(i=0; i<10; i++) printf("%d ", *p);

  return 0;
}
```

EXERCISES

1. You cannot multiply a pointer.
2. No, you can only add or subtract integer values.
3. 108

EXERCISES

1. No, you cannot change the value of a pointer that is generated by using an array name without an index.
2. 8

3.
```
#include <stdio.h>

int main(void)
{
  char str[80], *p;

  printf("Enter a string: ");
  gets(str);

  p = str;

  /* While not at the end of the string and no
     space has been encountered, increment p to
     point to next character.
  */
  while(*p && *p!=' ') p++;

  printf(p);

  return 0;
}
```

EXERCISE

1.
```
#include <stdio.h>

int main(void)
{
  char *one = "one";
  char *two = "two";
  char *three = "three";

  printf("%s %s %s\n", one, two, three);
  printf("%s %s %s\n", one, three, two);
  printf("%s %s %s\n", two, one, three);
  printf("%s %s %s\n", two, three, one);
  printf("%s %s %s\n", three, one, two);
  printf("%s %s %s\n", three, two, one);

  return 0;
}
```

EXERCISE

1.
```
#include <stdio.h>
#include <string.h>

 int main(void)
{
  char *p[3] = {
    "yes", "no",
    "maybe - rephrase the question"
  } ;
  char str[80];

  printf("Enter your question: \n");
  gets(str);

  printf(p[strlen(str) % 3]);

  return 0;
}
```

EXERCISE

1.
```
#include <stdio.h>

int main(void)
{
  int i, *p, **mp;

  p = &i;
  mp = &p;

  **mp = 10;

  printf("%p %p %p", &i, p, mp);

  return 0;
}
```

6.7 EXERCISES

1.
```
#include <stdio.h>
#include <string.h>

void mystrcat(char *to, char *from);

int main(void)
{
  char str[80];

  strcpy(str, "first part");
  mystrcat(str, " second part");
  printf(str);

  return 0;
}

void mystrcat(char *to, char *from)
{
  /* find the end of to */
  while(*to) to++;

  /* concatenate the string */
  while(*from) *to++ = *from++;

  /* add the null terminator */
  *to = '\0';
}
```

2.
```
#include <stdio.h>

void f(int *p);

int main(void)
{
  int i;

  f(&i);

  printf("%d", i);

  return 0;
}
```

```
void f(int *p)
{
  *p = -1;
}
```

MASTERY SKILLS CHECK

1. `double *p;`

2.
```
#include <stdio.h>

int main(void)
{
  int i, *p;

  p = &i;

  *p = 100;

  printf("%d", i);

  return 0;
}
```

3. No. The pointer **p** has never been initialized to point to a valid piece of memory that can hold a string.

4. Pointers and arrays are basically two ways of looking at the same thing. They are virtually interchangeable.

5.
```
str[2]

*(str+2)

*(p+2)
```

6. 108

CUMULATIVE SKILLS CHECK

1. Pointers are often more convenient than array indexing and may be faster in some cases.

2.
```
#include <stdio.h>

int main(void)
{
  char str[80], *p;
  int spaces;

  printf("Enter a string: ");
  gets(str);

  spaces = 0;
  p = str;
  while(*p) {
    if(*p==' ') spaces++;
    p++;
  }

  printf("Number of spaces: %d", spaces);

  return 0;
}
```

3.
```
*((int *)count + (44 * 10) + 8) = 99;
```

CHAPTER 7

REVIEW SKILLS CHECK

1. The fragment assigns to **i** the value 19 indirectly using a pointer.
2. An array name with no index generates a pointer to the start of the array.
3. Yes, the fragment is correct. It works because the compiler creates a string table entry for the string "this is a string" and assigns **p** a pointer to the start of it.
4.
```
#include <stdio.h>

int main(void)
{
  double d, *p;

  p = &d;

  *p = 100.99;

  printf("%f", d);

  return 0;
}
```

5.
```
#include <stdio.h>

int mystrlen(char *p);

int main(void)
{
  char str[80];

  printf("Enter a string: ");
  gets(str);

  printf("Length is %d", mystrlen(str));

  return 0;
}

int mystrlen(char *p)
{
  int i;

  i = 0;
  while(*p) {
    i++;
    p++;
  }
  return i;
}
```

6. The fragment is correct. It displays **C**.

7.1 EXERCISES

1.
```
#include <stdio.h>

double avg();

int main(void)
{
  printf("%f", avg());

  return 0;
}

double avg()
{
  int i;
  double sum, num;

  sum = 0.0;
  for(i=0; i<10; i++) {
    printf("Enter next number: ");
    scanf("%lf", &num);
    sum = sum + num;
  }
  return sum / 10.0;
}
```

2.
```
#include <stdio.h>

double avg(void);

int main(void)
{
  printf("%f", avg());

  return 0;
}

double avg(void)
{
  int i;
  double sum, num;

  sum = 0.0;
  for(i=0; i<10; i++) {
    printf("Enter next number: ");
    scanf("%lf", &num);
    sum = sum + num;
  }
  return sum / 10.0;
}
```

3. The program is correct. However, the program would be better if a full function prototype were used when declaring **myfunc( )**.
4. `double *Purge(void);`

EXERCISES

1.
```
#include <stdio.h>

int fact(int i);

int main(void)
{
  printf("5 factorial is %d", fact(5));

  return 0;
}

int fact(int i)
{
  if(i==1) return 1;
  else return i * fact(i-1);
}
```

2. The function will call itself repeatedly, until it crashes the program, because there is no condition that prevents a recursive call from occurring.
3.
```
#include <stdio.h>

void display(char *p);

int main(void)
{
  display("this is a test");

  return 0;
}

void display(char *p)
{
  if(*p) {
    printf("%c", *p);
    display(p+1);
  }
}
```

EXERCISES

1. No. The function **myfunc( )** is being called with a pointer tc the first parameter instead of the parameter itself.

2.
```
#include <stdio.h>

void prompt(char *msg, char *str);

int main(void)
{
  char str[80];

  prompt("Enter a string: ", str);
  printf("Your string is: %s", str);

  return 0;
}

void prompt(char *msg, char *p)
{
  printf(msg);
  gets(p);
}
```

3. In call by value, the value of an argument is passed to a function. In call by reference, the address of an argument is passed to a function.

EXERCISES

1.
```
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int i;

  if(argc!=3) {
    printf("You must specify two arguments.");
    exit(1);
  }

  i = strcmp(argv[1], argv[2]);
  if(i < 0) printf("%s > %s", argv[2], argv[1]);
  else if(i > 0) printf("%s > %s", argv[1], argv[2]);
  else printf("They are the same");

  return 0;
}
```

2.
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  if(argc!=3) {
    printf("You must specify two numbers.");
    exit(1);
  }

  printf("%f", atof(argv[1]) + atof(argv[2]));

  return 0;
}
```

3.
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  if(argc!=4) {

    printf("You must specify the operation ");
    printf("followed by two numbers.");
    exit(1);
  }

  if(!strcmp("add", argv[1]))
    printf("%f", atof(argv[2]) + atof(argv[3]));
  else if(!strcmp("subtract", argv[1]))
    printf("%f", atof(argv[2]) - atof(argv[3]));
  else if(!strcmp("multiply", argv[1]))
    printf("%f", atof(argv[2]) * atof(argv[3]));
  if(!strcmp("divide", argv[1]))
    printf("%f", atof(argv[2]) / atof(argv[3]));

  return 0;
}
```

EXERCISE

1.
```
#include <stdio.h>

double f_to_m(double f);

int main(void)
{
  double feet;

  printf("Enter feet: ");
  scanf("%lf", &feet);
  printf("Meters: %f", f_to_m(feet));

  return 0;
}

/* use old-style declaration. */
double f_to_m(f)
double f;
{
  return f / 3.28;
}
```

MASTERY SKILLS CHECK

1. A function that does not have parameters specifies **void** in the parameter list of its prototype.

2. A function prototype tells the compiler these three things: the return type of the function, the type of its parameters, and the number of its parameters. It is useful because it allows the compiler to find errors if the function is called incorrectly.
3. Command-line arguments are passed to a C program through the **argc** and **argv** parameters to **main( )**.
4.
```
#include <stdio.h>

void alpha(char ch);

int main(void)
{
  alpha('A');

  return 0;
}

void alpha(char ch)
{
  printf("%c", ch);
  if(ch < 'Z') alpha(ch+1);
}
```

5.
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *p;

  if(argc!=2) {
    printf("You need to specify a string");
    exit(1);
  }

  p = argv[1];

  while(*p) {
    printf("%c", (*p)+1);
    p++;
  }
  return 0;
}
```

6. The prototype is shown here.

   ```
   double myfunc(int x, int y, char ch);
   ```

7. Using the old-style function declaration, the function from Exercise 6 looks like this.

   ```
   double myfunc(x, y, ch)
   int x, y;
   char ch;
   {
     .
     .
     .
   }
   ```

8. The **exit( )** function causes immediate program termination also returns a value to the operating system.

9. The **atoi( )** function converts its string argument into its equivalent integer form. The string must represent (in string form) a valid integer.

CUMULATIVE SKILLS CHECK

1.
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  if(argc!=2) {
    printf("Specify a password");
    exit(1);
  }
  if(!strcmp(argv[1], "password"))
    printf("Access Permitted");
  else printf("Access Denied");

  return 0;
}
```

2.
```
#include <stdio.h>
#include <ctype.h>

void string_up(char *p);

int main(void)
{
  char str[] = "this is a test";

  string_up(str);
  printf(str);

  return 0;
}

void string_up(char *p)
{
  while(*p) {
    *p = toupper(*p);
    p++;
  }
}
```

3.
```
#include <stdio.h>

void avg(double *d, int num);

int main(void)
{
  double nums[] = {1.0, 2.0, 3.0, 4.0, 5.0,
                   6.0, 7.0, 8.0, 9.0, 10.0};

  avg(nums, 10);

  return 0;
}

void avg(double *d, int num)
{
  double sum;
  int temp;

  temp = num-1;

  for(sum=0; temp>=0; temp--)
```

```
        sum = sum + d[temp];

      printf("Average is %f", sum / (double) num);
    }
```

4. A pointer contains the address of another variable. When a pointer is passed to a function, the function may alter the contents of the object pointed to by the pointer. This is the equivalent of call by reference.

CHAPTER 8

REVIEW SKILLS CHECK

1. To allow the compiler to verify that a function is being called correctly, you must include its prototype.
2. Function prototypes enable the compiler to provide stronger type checking between the arguments used to call a function and the parameters of the function. Also, it lets the compiler confirm that the function is called with the proper number of arguments.
3.
```
#include <stdio.h>
#include <math.h>

double hypot(double s1, double s2);

int main(void)
{
  printf("%f", hypot(12.2, 19.2));

  return 0;
}

double hypot(double s1, double s2)
{
  double h;

  h = s1*s1 + s2*s2;
  return sqrt(h);
}
```

4. When a function does not return a value, its return type should be specified as **void**.

5.
```
#include <stdio.h>

int rstrlen(char *p);

int main(void)
{
  printf("%d", rstrlen("hello there"));

  return 0;
}

int rstrlen(char *p)
{
  if(*p) {
    p++;
    return 1+rstrlen(p);
  }
  else return 0;
}
```

6.
```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("There were %d arguments.\n", argc);
  printf("The last one is %s.", argv[argc-1]);

  return 0;
}
```

7.
```
func(a, ch, d)
int a;
char ch;
double d;
{
```

EXERCISES

1.
```
#include <stdio.h>

#define MAX 100
```

```
#define COUNTBY 3

int main(void)
{
  int i;

  for(i=0; i<MAX; i++)
    if(!(i%COUNTBY)) printf("%d ", i);

  return 0;
}
```

2. No, the fragment is wrong because a macro cannot be defined terms of another before the second macro is defined. Stated differently, **MIN** is not defined when **MAX** is being defined.
3. As the macro is used, the fragment is wrong. The string needs be within double quotes.
4. Yes.

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i;

  do {
    i = getchar();
    if(i==EOF) {
      printf("Error on input.");
      break;
    }
    if(putchar('.')==EOF) {
      printf("Error on output.");
      break;
    }
  } while((char) i != '\n');

  return 0;
}
```

2. The **putchar( )** function outputs a character. It cannot output a string.

EXERCISES

1.
```
#include <conio.h>
#include <stdio.h>

int main(void)
{
  char ch;

  ch = getch();
  printf("%d", ch);

  return 0;
}
```

2.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  do {
    printf("%c", '.');
  } while(!kbhit());

  return 0;
}
```

EXERCISES

2. No. The program is incorrect because **gets( )** must be called with a pointer to an actual array.

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  unsigned long i;

  for(i=2; i<=100; i++)
    printf("%-10lu %-10lu %-10lu\n", i, i*i, i*i*i);

  return 0;
}
```

2. `printf("Clearance price: 40% off as marked");`

3. `printf("%.2f", 1023.03);`

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  char first[21], middle[21], last[21];

  printf("Enter your entire name: ");
  scanf("%20s%20s%20s", first, middle, last);
  printf("%s %s %s", first, middle, last);

  return 0;
}
```

2.
```
#include <stdio.h>

int main(void)
{
  char num[80];

  printf("Enter a floating point number: ");
  scanf("%[0-9.]", num);
  printf(num);

  return 0;
}
```

3. No, a character can only have a maximum field length of 1.

4.
```
#include <stdio.h>

int main(void)
```

```
{
  char str[80];
  double d;
  int i, num;

  printf("Enter a string, a double, and an integer: ");
  scanf("%s%lf%d%n", str, &d, &i, &num);
  printf("Number of characters read: %d", num);

  return 0;
}
```

5.
```
#include <stdio.h>

int main(void)
{
  unsigned u;

  printf("Enter hexadecimal number: ");
  scanf("%x", &u);
  printf("Decimal equivalent: %u", u);

  return 0;
}
```

MASTERY SKILLS CHECK

1. All these functions input a character from the keyboard. The **getchar( )** function is often implemented using line-buffered I/O which makes its use in interactive environments undesirable. The **getche( )** is an interactive equivalent to **getchar( )**. The **getch( )** function is the same as **getche( )** except that it does not echo the character typed.
2. The **%e** specifier outputs a number in scientific notation using a lowercase 'e'. The **%E** specifier outputs a number in scientific notation using an 'E'.
3. A scanset is a set of characters that **scanf( )** matches with input. As long as the characters being read are part of the scanset, **scanf( )** continues to input them into the array pointed to by the scanset's corresponding argument.

4.
```
#include <stdio.h>

int main(void)
{
  char name[80], date[80], phone[80];

  printf("Enter first name, birthdate ");
  printf("and phone number:\n");
  scanf("%s%8s%8s", name, date, phone);
  printf("%s %s %s", name, date, phone);

  return 0;
}
```

5. The **puts( )** function is much smaller and faster than **printf( )** But, it can only output strings.

6.
```
#include <stdio.h>

#define COUNT 100

int main(void)
{
  int i;

  for(i=0; i<COUNT;i++)
    printf("%d ", i);

  return 0;
}
```

7. **EOF** is a macro that stands for end-of-file. It is defined in STDIO.H.

CUMULATIVE SKILLS CHECK

1.
```
#include <stdio.h>

int main(void)
{
  char name[9][80];
  double b_avg[9];
  int i, h, l;
  double high, low, team_avg;

  for(i=0; i<9; i++) {
    printf("Enter name %d: ", i+1);
    scanf("%s", name[i]);
    printf("Enter batting average: ");
    scanf("%lf", &b_avg[i]);
    printf("\n");
  }

  high = 0.0;
  low = 1000.0;
  team_avg = 0.0;
  for(i=0; i<9; i++) {
    if(b_avg[i]>high) {
      h = i;
      high = b_avg[i];
    }
    if(b_avg[i]<low) {
      l = i;
      low = b_avg[i];
    }
    team_avg = team_avg+b_avg[i];
  }
  printf("The high is %s %f\n", name[h], b_avg[h]);
  printf("The low is %s %f\n", name[l], b_avg[l]);
  printf("The team average is %f", team_avg/9.0);

  return 0;
}
```

2. Note: There are many ways you could have written this program. This one is simply representative.

```
/* An electronic card catalog. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100

int menu(void);
void display(int i);
void author_search(void);
void title_search(void);
void enter(void);

char names[MAX][80]; /* author names */
char titles[MAX][80]; /* titles */
char pubs[MAX][80]; /* publisher */

int top = 0; /* last location used */

int main(void)
{
  int choice;

  do {
    choice = menu();
    switch(choice) {
      case 1: enter(); /* enter books */
        break;
      case 2: author_search(); /* search by author *,
        break;
      case 3: title_search(); /* search by title */
        break;
    }
  } while(choice!=4);

  return 0;
}

/* Return a menu selection. */
menu(void)
{
  char str[80];
  int i;

  printf("Card Catalog:\n");
  printf("  1. Enter\n");
  printf("  2. Search by Author\n");
  printf("  3. Search by Title\n");
  printf("  4. Quit\n");

  do {
    printf("Choose your selection: ");
    gets(str);
    i = atoi(str);
    printf("\n");
  } while(i<1 || i>4);
  return i;
}

/* Enter books into database. */
void enter(void)
{
  int i;

  for(i=top; i<MAX; i++) {
    printf("Enter author name (ENTER to quit): ");
    gets(names[i]);
    if(!*names[i]) break;
    printf("Enter title: ");
    gets(titles[i]);
    printf("Enter publisher: ");
    gets(pubs[i]);
  }
  top = i;
}

/* Search by author. */
void author_search(void)
{
  char name[80];
  int i, found;

  printf("Name: ");
  gets(name);

  found = 0;
  for(i=0; i<top; i++)
    if(!strcmp(name, names[i])) {
      display(i);
      found = 1;
      printf("\n");
  }

  if(!found) printf("Not Found\n");
}

/* Search by title.*/
void title_search(void)
{
  char title[80];
```

```
    int i, found;

    printf("Title: ");
    gets(title);

    found = 0;
    for(i=0; i<top; i++)
      if(!strcmp(title, titles[i])) {
        display(i);
        found = 1;
        printf("\n");
      }
    if(!found) printf("Not Found\n");
}

/* Display catalog entry. */
void display(int i)
{
  printf("%s\n", titles[i]);
  printf("by %s\n", names[i]);
  printf("Published by %s\n", pubs[i]);
}
```

CHAPTER 9

REVIEW SKILLS CHECK

1. The **getchar( )** function is defined by the ANSI standard and is used to input characters from the keyboard. However, in most implementations, it uses line-buffered I/O, which makes it impractical for interactive use. The **getche( )** function is not defined by the ANSI standard, but it is quite common and is essentially an interactive version of **getchar( )**.
2. When **scanf( )** is reading a string, it stops when it encounters the first whitespace character.
3.
```
#include <stdio.h>

int isprime(int i);

int main(void)
{
  int i, count;

  count = 0;
  for(i=2; i<1001; i++)
    if(isprime(i)) {
      printf("%10d", i);
      count++;
      if(count==4) {
        printf("\n");
       count = 0;
      }
    }

  return 0;
}

int isprime(int i)
{
  int j;

  for(j=2; j<=(i/2); j++)
     if(!(i%j)) return  0;
  return 1;
}
```

4.
```
#include <stdio.h>

int main(void)
{
  double d;
  char ch;
  char str[80];

  printf("Enter a double, a character, and a string\n");
  scanf("%lf%c%20s", &d, &ch, str);
  printf("%f %c %s", d, ch, str);

  return 0;
}
```

5.
```
#include <stdio.h>

int main(void)
{
  char str[80];
```

```
    printf("Enter leading digits followed by a string\n");
    scanf("%*[0-9]%s", str);
    printf("%s", str);

    return 0;
  }
```

EXERCISES

1.
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  /* see if filename is specified */
  if(argc!=2) {
    printf("File name missing.\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  while((ch=fgetc(fp)) != EOF) putchar(ch);

  fclose(fp);

  return 0;
}
```

2.
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int count[26];

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;
  int i;

  /* see if file name is specified */
  if(argc!=2) {
    printf("File name missing.\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  while((ch=fgetc(fp))!=EOF) {
    ch = toupper(ch);
    if(ch>='A' && ch<='Z') count[ch-'A']++;
  }

  for(i=0; i<26; i++)
    printf("%c occurred %d times\n", i+'A', count[i]);

  fclose(fp);

  return 0;
}
```

3.
```
/* Copy a file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  FILE *from, *to;
  char ch, watch;

  /* see if correct number of command line arguments */
  if(argc<3) {
    printf("Usage: copy <source> <destination>\n");
    exit(1);
  }

  /* open source file */
```

```
    if(( from = fopen(argv[1], "r"))==NULL){
      printf("Cannot open source file.\n");
      exit(1);
    }

    /* open destination file */
    if((to = fopen(argv[2], "w"))==NULL) {
      printf("Cannot open destination file.\n");
      exit(1);
    }

    if(argc==4 && !strcmp(argv[3], "watch")) watch = 1;
    else watch = 0;

    /* copy the file */
    while((ch=fgetc(from))!=EOF) {
      fputc(ch, to);
      if(watch) putchar(ch);
    }
    fclose(from);
    fclose(to);

    return 0;
  }
```

EXERCISES

1.
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;

  unsigned count;

  /* see if file name is specified */
  if(argc!=2) {
    printf("File name missing.\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  count = 0;
  while(!feof(fp)) {
    fgetc(fp);
    if(ferror(fp)) {
      printf("File error.\n");
      exit(1);
    }
    count++;
  }

  printf("File has %u bytes", count-1);
  fclose(fp);

  return 0;
}
```

2.
```
/* Exchange two files. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  FILE *f1, *f2, *temp;
  char ch;

  /* see if correct number of command line arguments */
  if(argc!=3) {
    printf("Usage: exchange<f1> <f2>\n");
    exit(1);
  }

  /* open first file */
  if((f1 = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open first file.\n");
    exit(1);
  }

  /* open second file */
  if((f2 = fopen(argv[2],"rb"))==NULL) {
    printf("Cannot open second file.\n");
    exit(1);
  }

  /* open temporary file */
  if((temp = fopen("temp.tmp", "wb"))==NULL) {
    printf("Cannot open temporary file.\n");
    exit(1);
  }

  /* copy f1 to temp */
  while(!feof(f1)) {
    ch = fgetc(f1);
    if(!feof(f1)) fputc(ch, temp);
  }

  fclose(f1);
  /* open first file for output */
  if((f1 = fopen(argv[1], "wb"))==NULL) {
    printf("Cannot open first file.\n");
    exit(1);
  }

  /* copy f2 to f1 */
  while(!feof(f2)) {
    ch = fgetc(f2);
    if(!feof(f2)) fputc(ch, f1);
  }
  fclose(f2);
  fclose(temp);

  /* open second file for output */
  if((f2 = fopen(argv[2], "wb"))==NULL) {
    printf("Cannot open second file.\n");
    exit(1);
  }
  /* open temp file for input */
  if((temp = fopen("temp.tmp", "rb"))==NULL) {
    printf("Cannot open temporary file.\n");
    exit(1);
  }

  /* copy temp to f2 */
  while(!feof(temp)) {
    ch = fgetc(temp);
    if(!feof(temp)) fputc(ch, f2);
  }
  fclose(f1);
  fclose(f2);
  fclose(temp);

  return 0;
}
```

EXERCISES

1.
```
/* A simple computerized telephone book. */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char names[100][40];
char numbers[100][40];

int loc=0;

int menu(void);
void enter(void);
void load(void);
void save(void);
void find(void);

int main(void)
{
  int choice;

  do {
    choice = menu();
    switch(choice) {
      case 1: enter();
        break;
      case 2: find();
        break;
      case 3: save();
        break;
      case 4: load();
    }
  } while(choice!=5);

  return 0;
```

```
}

/* Get menu choice. */
int menu(void)
{
  int i;
  char str[80];

  printf("1. Enter names and numbers\n");
  printf("2. Find numbers\n");
  printf("3. Save directory to disk\n");
  printf("4. Load directory from disk\n");
  printf("5. Quit\n");

  do {
    printf("Enter your choice: ");
    gets(str);
    i = atoi(str);
    printf("\n");
  } while(i<1 || i>5);
  return i;
}

void enter(void)
{

  for(;loc<100; loc++) {
    if(loc<100) {
      printf("Enter name and phone number:\n");
      gets(names[loc]);
      if(!*names[loc]) break;
      gets(numbers[loc]);
    }
  }
}

void find(void)
{
  char name[80];
  int i;

  printf("Enter name: ");
  gets(name);

  for(i=0; i<100; i++)
```

```
      if(!strcmp(name, names[i]))
        printf("%s %s\n", names[i], numbers[i]);
}

void load(void)
{
  FILE *fp;

  if((fp = fopen("phone", "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  loc = 0;
  while(!feof(fp)) {
    fscanf(fp, "%s%s", names[loc], numbers[loc]);
    loc++;
  }
  fclose(fp);
}

void save(void)
{
  FILE *fp;
  int i;

  if((fp = fopen("phone", "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  for(i=0; i<loc; i++) {
    fprintf(fp, "%s %s ", names[i], numbers[i]);
  }
  fclose(fp);
}
```

2.
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;
  char str[80];
  int count;

  /* see if correct number of common line arguments */
  if(argc!=2) {
    printf("Usage: display <file>\n");
    exit(1);
  }

  /* open the file */
  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open the file.\n");
    exit(1);
  }

  count = 0;
  while(!feof(fp)) {
    fgets(str, 79, fp);
    printf("%s", str);
    count++;

    if(count==23) {
      printf("More? (y/n) ");
      gets(str);
      if(toupper(*str)=='N') break;
      count = 0;
    }
  }

  fclose(fp);

  return 0;
}
```

3.
```
/* Copy a file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  FILE *from, *to;
  char str[128];

  /* see if correct number of command line arguments */
  if(argc<3) {
    printf("Usage: copy <source> <destination>\n");
    exit(1);
  }

  /* open source file */
  if((from = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  }

  /* open destination file */
  if((to = fopen(argv[2], "w"))==NULL) {
    printf("Cannot open destination file.\n");
    exit(1);
  }

  /* copy the file */
  while(!feof(from)) {
    fgets(str, 127, from);
    if(ferror(from)) {
      printf("Error on input.\n");
      break;
    }
    if(!feof(from)) fputs(str, to);
    if(ferror(to)) {
      printf("Error on output.\n");
      break;
    }
  }

  if(fclose(from)==EOF) {
    printf("Error closing source file.\n");
    exit(1);
  }

  if(fclose(to)==EOF) {
    printf("Error closing destination file.\n");
    exit(1);
  }

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp1, *fp2;
  double d;
  int i;

  if((fp1 = fopen("values", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  if((fp2 = fopen("count", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  d = 1.0;
  for(i=0; d!=0.0 && i<32766; i++) {
    printf("Enter a number (0 to quit): ");
    scanf("%lf", &d);
    fwrite(&d, sizeof d, 1, fp1);
  }

  fwrite(&i, sizeof i, 1, fp2);

  fclose(fp1);
  fclose(fp2);

  return 0;
}
```

2.
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp1, *fp2;
  double d;
  int i;
  if((fp1 = fopen("values", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  if((fp2 = fopen("count", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  fread(&i, sizeof i, 1, fp2); /* get count */

  for(; i>0; i--) {
    fread(&d, sizeof d, 1, fp1);
    printf("%f\n", d);
  }

  fclose(fp1);
  fclose(fp2);

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;
  long l;

  if(argc!=2) {
    printf("You must specify the file.\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "rb"))== NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }
```

```
    fseek(fp, 0, SEEK_END); /* find end of file */
    l = ftell(fp);

    /* go back to the start of the file */
    fseek(fp, 0, SEEK_SET);
    for( ; l>=0; l = l - 2L) {
      ch = fgetc(fp);
      putchar(ch);
      fseek(fp, 1L, SEEK_CUR);
    }

    fclose(fp);

    return 0;
  }
```

2.
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  unsigned char ch, val;

  if(argc!=3) {
    printf("Usage: find <filename> <value>");
    exit(1);
  }

  if((fp = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  val = atoi(argv[2]);

  while(!feof(fp)) {
    ch = fgetc(fp);
    if(ch == val)
      printf("Found value at %ld\n", ftell(fp));
  }

  fclose(fp);
  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(void)
{
  char fname[80];

  printf("Enter name of file to erase: ");
  gets(fname);
  printf("Are you sure? (Y/N) ");
  if(toupper(getchar())=='Y')
    if(remove(fname))
      printf("\nFile not found or write protected.\n");

  return 0;
}
```

EXERCISE

1.
```
/* Copy using redirection.

   Execute like this:

   C>NAME < in > out

*/

#include <stdio.h>

int main(void)
{
  char ch;

  while(!feof(stdin)) {
    scanf("%c", &ch);
    if(!feof(stdin)) printf("%c", ch);
  }
```

```
    return 0;
  }
```

MASTERY SKILLS CHECK

1.
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char str[80];

  /* see if file name is specified */
  if(argc!=2) {
    printf("File name missing.\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  while (!feof(fp)) {
    fgets(str, 79, fp);
    if(!feof(fp)) printf("%s", str);
    printf("...More? (y/n) ");
    if(toupper(getchar())=='N') break;
    printf("\n");
  }

  fclose(fp);

  return 0;
}
```

2.
```
/* Copy a file and convert to uppercase. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```
int main(int argc, char *argv[])
{
  FILE *from, *to;
  char ch;

  /* see if correct number of command line arguments */
  if(argc!=3) {
    printf("Usage: copy <source> <destination>\n");
    exit(1);
  }

  /* open source file */
  if((from = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  }

  /* open destination file */
  if((to = fopen(argv[2], "w"))==NULL) {
    printf("Cannot open destination file.\n");
    exit(1);
  }

  /* copy the file */
  while(!feof(from)) {
    ch = fgetc(from);
    if(!feof(from)) fputc(toupper(ch), to);
  }
  fclose(from);
  fclose(to);

  return 0;
}
```

3. The **fprintf( )** and **fscanf( )** functions operate exactly like **printf( )** and **scanf( )**, except that they work with files.

4.
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;
  int i, num;
```

```
    if((fp = fopen("rand", "wb"))==NULL) {
      printf("Cannot open file.\n");
      exit(1);
    }

    for(i=0; i<100; i++) {
      num = rand();
      fwrite(&num, sizeof num, 1, fp);
    }

    fclose(fp);

    return 0;
  }
```

5.
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;
  int i, num;

  if((fp = fopen("rand", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  for(i=0; i<100; i++) {
    fread(&num, sizeof num, 1, fp);
    printf("%d\n", num);
  }

  fclose(fp);

  return 0;
}
```

6.
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;
  long i;
  int num;

  if((fp = fopen("rand", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  printf("Which number (0-99)? ");
  scanf("%ld", &i);
  fseek(fp, i * sizeof(int), SEEK_SET);
  fread(&num, sizeof num, 1, fp);
  printf("%d\n", num);

  fclose(fp);

  return 0;
}
```

7. The "console" I/O functions are simply special cases of the general file system.

CUMULATIVE SKILLS CHECK

1.
```
/* An electronic card catalog. */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

int menu(void);
void display(int i);
void author_search(void);
void title_search(void);
void enter(void);
void save(void);
void load(void);

char names[MAX][80]; /* author names */
char titles[MAX][80]; /* titles */
char pubs[MAX][80]; /* publisher */

int top = 0; /* last location used */
int main(void)
{
  int choice;

  load(); /* read in catalog */

  do {
    choice = menu();
    switch(choice) {
      case 1: enter(); /* enter books */
        break;
      case 2: author_search(); /* search by author */
        break;
      case 3: title_search(); /* search by title */
        break;
      case 4: save();
    }
  } while(choice!=5);

  return 0;
}

/* Return a menu selection. */
menu(void)
{
  int i;
  char str[80];

  printf("Card Catalog:\n");
  printf("  1. Enter\n");
  printf("  2. Search by author\n");
  printf("  3. Search by Title\n");
  printf("  4. Save catalog\n");
  printf("  5. Quit\n");

  do {
    printf("Choose your selection: ");
    gets(str);
    i = atoi(str);
    printf("\n");
  } while(i<1 || i>5);

  return i;
}
```

```
/* Enter books into database. */
void enter(void)
{
  int i;

  for(i=top; i<MAX; i++) {
    printf("Enter author name (ENTER to quit) : ");
    gets(names[i]);
    if(!*names[i]) break;
    printf("Enter title: ");
    gets(titles[i]);
    printf("Enter publisher: ");
    gets(pubs[i]);
  }
  top = i;
}

/* Search by author. */
void author_search(void)
{
  char name[80];
  int i, found;

  printf("Name: ");
  gets(name);

  found = 0;
  for(i=0; i<top; i++)
    if(!strcmp(name, names[i])) {
      display(i);
      found = 1;
      printf("\n");
    }

  if(!found) printf("Not Found\n");
}

/* Search by title. */
void title_search(void)
{
  char title[80];
  int i, found;

  printf("Title: ");
  gets(title);
  found = 0;
  for(i=0; i<top; i++)
    if(!strcmp(title, titles[i])) {
      display(i);
      found = 1;
      printf("\n");
    }
  if(!found) printf("Not Found\n");
}

/* Display catalog entry. */
void display(int i)
{
  printf("%s\n", titles[i]);
  printf("by %s\n", names[i]);
  printf("Published by %s\n", pubs[i]);
}

/* Load the catalog file. */
void load(void)
{
  FILE *fp;

  if((fp = fopen("catalog", "r"))==NULL) {
    printf("Catalog file not on disk.\n");
    return;
  }

  fread(&top, sizeof top, 1, fp); /* read count */
  fread(names, sizeof names, 1, fp);
  fread(titles, sizeof titles, 1, fp);
  fread(pubs, sizeof pubs, 1, fp);

  fclose(fp);
}

/* save the catalog file. */
void save(void)
{
  FILE *fp;

  if((fp = fopen("catalog", "w"))==NULL) {
    printf("Cannot open catalog file.\n");
    exit(1);
    }

    fwrite(&top, sizeof top, 1, fp);
    fwrite(names, sizeof names, 1, fp);

    fwrite(titles, sizeof titles, 1, fp);
    fwrite(pubs, sizeof pubs, 1, fp);

    fclose(fp);
  }
```

2.
```
/* Copy a file and remove tabs. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  FILE *from, *to;
  char ch;
  int tab, count;

  /* see if correct number of command line arguments */
  if(argc!=3) {
    printf("Usage: copy <source> <destination>\n");
    exit(1);
  }

  /* open source file */
  if((from = fopen(argv[1], "r"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  }

  /* open destination file */
  if((to = fopen(argv[2], "w"))==NULL) {
    printf("Cannot open destination file.\n");
    exit(1);
  }
  /* copy the file */
  count = 0;
  while(!feof(from)) {
    ch = fgetc(from);
    if(ch=='\t') {
      for(tab = count; tab<8; tab++)
```

```
          fputc(' ', to);
        count = 0;
      }
      else {
        if(!feof(from)) fputc(ch, to);
        count++;
        if(count==8 || ch=='\n') count = 0;
      }
    }
    fclose(from);
    fclose(to);

    return 0;
  }
```

CHAPTER 10

REVIEW SKILLS CHECK

1.
```
/* Copy a file. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *from, *to;
  char ch;

  /* see if correct number of command line arguments
  if(argc!=3) {
    printf("Usage: copy <source> <destination>\n");
    exit(1);
  }

  /* open source file */
  if((from = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  }

  /* open destination file */
  if((to = fopen(argv[2], "wb"))==NULL) {
    printf("Cannot open destination file.\n");
```

```
      exit(1);
    }

    /* copy the file */
    while(!feof(from)) {
      ch = fgetc(from);
      if(ferror(from)) {
        printf("Error on input.\n");
        break;
      }
      if(!feof(from)) fputc(ch, to);
      if(ferror(to)) {
        printf("Error on output.\n");
        break;
      }
    }

    if(fclose(from)==EOF) {
      printf("Error closing source file.\n");
      exit(1);
    }

    if(fclose(to)==EOF) {
      printf("Error closing destination file.\n");
      exit(1);
    }

    return 0;
  }
```

2.
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;

  /* open file */
  if((fp = fopen("myfile", "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  fprintf(fp, "%s %.2f %X %c", "this is a string",
          1230.23, 0x1FFF, 'A');
    fclose(fp);

    return 0;
  }
```

3.
```
  #include <stdio.h>
  #include <stdlib.h>

  int main(void)
  {
    FILE *fp;
    int count[20], i;

    /* open file */
    if((fp = fopen("TEMP", "wb"))==NULL) {
      printf("Cannot open file.\n");
      exit(1);
    }

    for(i=0; i<20; i++) count[i] = i+1;

    fwrite(count, sizeof count, 1, fp);

    fclose(fp);

    return 0;
  }
```

4.
```
  #include <stdio.h>
  #include <stdlib.h>

  int main(void)
  {
    FILE *fp;
    int count[20], i;

    /* open file */
    if((fp = fopen("TEMP", "rb"))==NULL) {
      printf("Cannot open file.\n");
      exit(1);
    }

    fread(count, sizeof count, 1, fp);
  for(i=0; i<20; i++) printf("%d ", count[i]);

  fclose(fp);

  return 0;
}
```

5. **stdin**, **stdout**, and **stderr** are three streams that are opened automatically when your C program begins executing. By default they refer to the console, but in operating systems that support I/O redirection, they can be redirected to other devices.
6. The **printf( )** and **scanf( )** functions are part of the C file system. They are simply special case functions that automatically use **stdin** and **stdout**.

EXERCISES

1.
```
/* A simple computerized telephone book. */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

struct phone_type {
  char name[40];
  int areacode;
  char number[9];
} phone[MAX];

int loc=0;

int menu(void);
void enter(void);
void load(void);
void save(void);
void find(void);

int main(void)
{
  int choice;
  do {
    choice = menu();
    switch(choice) {
      case 1: enter();
        break;
      case 2: find();
        break;
      case 3: save();
        break;
      case 4: load();
    }
  } while(choice!=5);

  return 0;
}

/* Get menu choice. */
menu(void)
{
  int i;
  char str[80];

  printf("1. Enter names and numbers\n");
  printf("2. Find numbers\n");
  printf("3. Save directory to disk\n");
  printf("4. Load directory from disk\n");
  printf("5. Quit\n");

  do {
    printf("Enter your choice: ");
    gets(str);
    i = atoi(str);
    printf("\n");
  } while (i<1 || i>5);
  return i;
}

void enter(void)
{
  char temp[80];

  for(;loc<100; loc++) {
    if(loc<100) {
      printf("Enter name: ");
      gets(phone[loc].name);
      if(!*phone[loc].name) break;
      printf("Enter area code: ");
      gets(temp);
      phone[loc].areacode = atoi(temp);
      printf("Enter number: ");
      gets(phone[loc].number);
    }
  }
}

void find(void)
{
  char name[80];
  int i;

  printf("Enter name: ");
  gets(name);
  if(!*name) return;

  for(i=0; i<100; i++)
    if(!strcmp(name, phone[i].name))
      printf("%s (%d) %s\n", phone[i].name,
             phone[i].areacode, phone[i].number);
}

void load(void)
{
  FILE *fp;

  if((fp = fopen("phone", "r"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  loc = 0;
  while(!feof(fp)) {
    fscanf(fp, "%s%d%s", phone[loc].name,
           &phone[loc].areacode, phone[loc].number);
    loc++;
  }
  fclose(fp);
}

void save(void)
{
  FILE *fp;
  int i;

  if((fp = fopen("phone", "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  for(i=0; i<loc; i++) {
    fprintf(fp, "%s %d %s ", phone[i].name,
            phone[i].areacode, phone[i].number);

  }
  fclose(fp);
}
```

2. The variable **i** is a member of structure **s_type**. Therefore, it cannot be used by itself. Instead, it must be accessed using **s** and the dot operator, as shown here.

   ```
   s.i = 10;
   ```

EXERCISES

1. No. Since **p** is a pointer to a structure, you must use the arrow operator, not the dot operator, to access a member.

2.
   ```
   #include <stdio.h>
   #include <time.h>

   int main(void)
   {
     struct tm *systime, *gmt;
     time_t t;

     t = time(NULL);
     systime = localtime(&t);

     printf("Time is %.2d:%.2d:%.2d\n", systime->tm_hour,
            systime->tm_min, systime->tm_sec);
     gmt = gmtime(&t);
     printf("Coordinated Universal Time is %.2d:%.2d:%.2d\n",
            gmt->tm_hour,
            gmt->tm_min, gmt->tm_sec);
   ```

```
  printf("Date: %.2d/%.2d/%.2d", systime->tm_mon+1,
         systime->tm_mday, systime->tm_year);

  return 0;
}
```

EXERCISES

1.
```
/* A simple computerized telephone book. */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

struct address {
  char street[40];
  char city[40];
  char state[3];
  char zip[12];
};

struct phone_type {
  char name[40];
  int areacode;
  char number[9];
  struct address addr;
} phone[MAX];

int loc=0;

int menu(void);
void enter(void);
void load(void);
void save(void);
void find(void);

int main(void)
{
  int choice;

  do {
    choice = menu( );
    switch(choice) {
      case 1: enter( );
        break;
      case 2: find( );
        break;
      case 3: save( );
        break;
      case 4: load( );
    }
  } while(choice!=5);

  return 0;
}

/* Get menu choice. */
menu(void)
{
  int i;
  char str[80];

  printf("1. Enter names and numbers\n");
  printf("2. Find numbers\n");
  printf("3. Save directory to disk\n");
  printf("4. Load directory from disk\n");
  printf("5. Quit\n");

  do {
    printf("Enter your choice: ");
    gets(str);
    i = atoi(str);
    printf("\n");
  } while(i<1 || i>5);
  return i;
}

void enter(void)
{
  char temp[80];

  for(;loc<100; loc++) {
    if(loc<100) {
      printf("Enter name: ");
      gets(phone[loc].name);
      if(!*phone[loc].name) break;
      printf("Enter area code: ");
      gets(temp);
      phone[loc].areacode = atoi(temp);
      printf("Enter number: ");
      gets(phone[loc].number);

      /* input address info */
      printf("Enter street address: ");
      gets(phone[loc].addr.street);
      printf("Enter city: ");
      gets(phone[loc].addr.city);
      printf("Enter State: ");
      gets(phone[loc].addr.state);
      printf("Enter zip code: ");
      gets(phone[loc].addr.zip);
    }
  }
}

void find(void)
{
  char name[80];
  int i;

  printf("Enter name: ");
  gets(name);
  if(!*name) return;

  for(i=0; i<100; i++)
    if(!strcmp(name, phone[i].name)) {
      printf("%s (%d) %s\n", phone[i].name,
             phone[i].areacode, phone[i].number);
      printf("%s\n%s %s %s\n", phone[i].addr.street,
             phone[i].addr.city, phone[i].addr.state,
             phone[i].addr.zip);
  }
}

void load(void)
{
  FILE *fp;

  if((fp = fopen("phone", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }
```

```
    loc = 0;
    while(!feof(fp)) {
      fread(&phone[loc], sizeof phone[loc], 1, fp);
      loc++;
    }
    fclose(fp);
  }

  void save(void)
  {
    FILE *fp;
    int i;

    if((fp = fopen("phone", "wb"))==NULL) {
      printf("Cannot open file.\n");
      exit(1);
    }

    for(i=0; i<loc; i++) {
      fwrite(&phone[i], sizeof phone[i], 1, fp);
    }
    fclose(fp);
  }
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  struct b_type {
    int a: 3;
    int b: 3;
    int c: 2;
  } bvar;

  bvar.a = -1;
  bvar.b = 3;
  bvar.c = 1;
  printf("%d %d %d", bvar.a, bvar.b, bvar.c);

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>
#include <stdlib.h>

union u_type {
  double d;
  unsigned char c[8];
} ;

double uread(FILE *fp);
void  uwrite(double num, FILE *fp);

int main(void)
{
  FILE *fp;
  double d;

  if((fp = fopen("myfile", "wb+"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  uwrite(100.23, fp);
  d = uread(fp);
  printf("%lf", d);

  return 0;
}

void uwrite (double num, FILE *fp)
{
  int i;
  union u_type var;

  var.d = num;
  for(i=0; i<8; i++) fputc(var.c[i], fp);
}

double uread(FILE *fp)
{
  int i;
  union u_type var;
```

```
      rewind(fp);
      for(i=0; i<8; i++) var.c[i] = fgetc(fp);

      return var.d;
    }
```

2.
```
#include <stdio.h>

int main(void)
{
  union t_type {
    long l;
    int i;
  } uvar;

  uvar.l = 0L; /* clear l */
  uvar.i = 100;

  printf("%ld", uvar.l);

  return 0;
}
```

MASTERY SKILLS CHECK

1. A structure is a named group of related variables. A union defines a memory location shared by two or more variables of different types.
2.
```
struct s_type {
  char ch;
  float d;
  int i;
  char str[80];
  double balance;
} s_var;
```
3. Because **p** is a pointer to a structure, you must use the arrow operator to reference an element, not the dot operator.
4.
```
#include <stdio.h>
#include <stdlib.h>

struct s_type {
  char name[40];
```

```
    char phone[14];
    int hours;
    double wage;
  } emp[10];

  int main(void)
  {
    FILE *fp;
    int i;
    char temp[80];

    if((fp = fopen("emp", "wb"))==NULL) {
      printf("Cannot open EMP file.\n");
      exit(1);
    }

    for(i=0; i<10; i++) {
      printf("Enter name: ");
      gets(emp[i].name);
      printf("Enter telephone number: ");
      gets(emp[i].phone);
      printf("Enter hours worked: ");
      gets(temp);
      emp[i].hours = atoi(temp);
      printf("Enter hourly wage: ");
      gets(temp);
      emp[i].wage = atof(temp);
    }

    fwrite(emp, sizeof emp, 1, fp);
    fclose(fp);

    return 0;
  }
```

5.
```
#include <stdio.h>
#include <stdlib.h>

struct s_type {
  char name[40];
  char phone[14];
  int hours;
  double wage;
} emp[10];
```

```
int main(void)
{
  FILE *fp;
  int i;

  if((fp = fopen("emp", "rb"))==NULL) {
    printf("Cannot open EMP file.\n");
    exit(1);
  }

  fread(emp, sizeof emp, 1, fp);
  for(i=0; i<10; i++) {
    printf("%s %s\n", emp[i].name, emp[i].phone);
    printf("%d %f\n\n", emp[i].hours, emp[i].wage);
  }

  fclose(fp);

  return 0;
}
```

6. A bit-field is a structure member that specifies its length in bits.

7.
```
#include <stdio.h>

int main(void)
{
  union u_type {
    short int i;
    unsigned char c[2];
  } uvar;

  uvar.i = 99;

  printf("High order byte: %u\n", uvar.c[1]);
  printf("Low order byte: %u\n", uvar.c[0]);

  return 0;
}
```

CUMULATIVE SKILLS CHECK

1.
```
#include <stdio.h>

struct s_type {
  int i;
  char ch;
  double d;
} var1, var2;

void struct_swap(struct s_type *i, struct s_type *j);

int main(void)
{
  var1.i = 100;
  var2.i = 99;
  var1.ch = 'a';
  var2.ch = 'b';
  var1.d = 1.0;
  var2.d = 2.0;

  printf("var1: %d %c %f\n", var1.i, var1.ch, var1.d);
  printf("var2: %d %c %f\n", var2.i, var2.ch, var2.d);

  struct_swap(&var1, &var2);

  printf("After swap:\n");
  printf("var1: %d %c %f\n", var1.i, var1.ch, var1.d);
  printf("var2: %d %c %f", var2.i, var2.ch, var2.d);

  return 0;
}

void struct_swap(struct s_type *i, struct s_type *j)
{
  struct s_type temp;

  temp = *i;
  *i = *j;
  *j = temp;
}
```

2.
```
/* Copy a file. */
#include <stdio.h>
```

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *from, *to;
  union u_type {
    int i;
    char ch;
  } uvar;

  /* see if correct number of command line arguments */
  if(argc!=3) {
    printf("Usage: copy <source> <destination>\n");
    exit(1);
  }

  /* open source file */
  if((from = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  }

  /* open destination file */
  if((to = fopen(argv[2], "wb"))==NULL) {
    printf("Cannot open destination file.\n");
    exit(1);
  }

  /* copy the file */
  for(;;) {
    uvar.i = fgetc(from);
    if(uvar.i==EOF) break;
    fputc(uvar.ch, to);
  }
  fclose(from);
  fclose(to);

  return 0;
}
```

3. You cannot use a structure as an argument to **scanf( )**. However, you can use a structure element as an argument, as shown here.

```
scanf("%d", &var.a);
```

4.
```
#include <string.h>
#include <stdio.h>

struct s_type {
  char str[80];
} var;

void f(struct s_type i);

int main(void)
{
  strcpy(var.str, "this is original string");
  f(var);
  printf("%s", var.str);

  return 0;
}

void f(struct s_type i)
{
  strcpy(i.str, "new string");
  printf("%s\n", i.str);
}
```

CHAPTER 11

REVIEW SKILLS CHECK

1.
```
#include <stdio.h>

struct num_type {
  int i;
  int sqr;
  int cube;
} nums[10];

int main(void)
{
  int i;

  for(i=1; i<11; i++) {
    nums[i-1].i = i;
```

```
      nums[i-1].sqr = i*i;
      nums[i-1].cube = i*i*i;
    }

    for(i=0; i<10; i++) {
      printf("%d ", nums[i].i);
      printf("%d ", nums[i].sqr);
      printf("%d\n", nums[i].cube);
    }

    return 0;
  }
```

2.
```
#include <stdio.h>

union i_to_c {
  char c[2];
  short int i;
} ic;

int main(void)
{
  printf("Enter an integer ");
  scanf("%hd", &ic.i);
  printf("Character representation of each byte: %c %c",
         ic.c[0], ic.c[1]);

  return 0;
}
```

3. The fragment displays **8**, the size of the largest element of the union.
4. To access a structure member when actually using a structure variable, you must use the dot operator. The arrow operator is used when accessing a member using a pointer to a structure.
5. A bit-field is a structure element whose size is specified in bits.

EXERCISES

1. The best variables to make into **register** types are **k** and **m**, because they are accessed most frequently.

2.
```
#include <stdio.h>

void sum_it(int value);

int main(void)
{
  sum_it(10);
  sum_it(20);
  sum_it(30);
  sum_it(40);

  return 0;
}

void sum_it(int value)
{
  static int sum=0;

  sum = sum + value;
  printf("Current value: %d\n", sum);
}
```

4. You cannot obtain the address of a **register** variable.

EXERCISES

1.
```
#include <stdio.h>

const double version = 6.01;

int main(void)
{
  printf("Version %.2f", version);

  return 0;
}
```

2.
```
#include <stdio.h>

char *mystrcpy(char *to, const char *from);

int main(void)
{
  char *p, str[80];

  p = mystrcpy(str, "testing");

  printf("%s %s", p, str);

  return 0;
}

char *mystrcpy(char *to, const char *from)
{
  char *temp;

  temp = to;

  while(*from) *to++ = *from++;
  *to = '\0' ; /* null terminator */

  return temp;
}
```

EXERCISES

2. `enum money {penny, nickel, quarter, half_dollar, dollar};`

3. No, you cannot output an enumeration constant as a string as is attempted in the **printf( )** statement.

EXERCISES

1.
```
#include <stdio.h>

typedef unsigned long UL;

int main(void)
{
  UL count;

  count = 312323;

  printf("%lu", count);
  return 0;
}
```

2. The **typedef** statement is out of order. The correct form of **typedef** is

   typedef *oldname newname*;

EXERCISES

1.
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *in, *out;
  unsigned char ch;

  if(argc!=3) {
    printf("Usage: code <in> <out>\n");
    exit(1);
  }

  if((in = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open input file.\n");
    exit(1);
  }

  if((out = fopen(argv[2], "wb"))==NULL) {
    printf("Cannot open output file.\n");
    exit(1);
  }

  while(!feof(in)) {
    ch = fgetc(in);
    if(!feof(in)) fputc(~ch, out);
  }

  fclose(in);
  fclose(out);

  return 0;
}
```

2.
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *in, *out;
  unsigned char ch;

  if(argc!=4) {
    printf("Usage: code <in> <out> <key>\n");
    exit(1);
  }

  if(( in = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open input file.\n");
    exit(1);
  }

  if((out = fopen(argv[2], "wb"))==NULL) {
    printf("Cannot open output file.\n");
    exit(1);
  }

  while(!feof(in)) {
    ch = fgetc(in);
    ch = *argv[3] ^ ch;
    if(!feof(in)) fputc(ch, out);
  }

  fclose(in);
  fclose(out);

  return 0;
}
```

3. a. 0000 0001

   b. 1111 1111

   c. 1111 1101

4.
```
char ch;

.
.
.
```

```
/* To zero high order bit, AND with 127, which
   in binary is 0111 1111. This causes the high-
   order bit to be zeroed and all other bits left
   untouched.
*/
ch = ch & 127;
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i, j, k;

  printf("Enter a number: ");
  scanf("%d", &i);

  j = i << 1;
  k = i >> 1;
  printf("%d doubled: %d\n", i, j);
  printf("%d halved: %d", i, k);

  return 0;
}
```

2.
```
#include <stdio.h>

void rotate(unsigned char *c);

int main(void)
{
  unsigned char ch;
  int i;

  ch = 1;

  for(i=0; i<16; i++) {
    rotate(&ch);
    printf("%u\n", ch);
  }
```

```
    return 0;
  }

  void rotate(unsigned char *c)
  {
    union {
      unsigned char ch[2];
      unsigned u;
    } rot;

    rot.u = 0; /* clear 16 bits */

    rot.ch[0] = *c;

    /* shift integer left */
    rot.u = rot.u << 1;

    /* See if a bit got shifted into c[1].
       If so, OR it back onto the other end. */
    if(rot.ch[1]) rot.ch[0] = rot.ch[0] | 1;

    *c = rot.ch[0];
  }
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i, j, answer;

  printf("Enter two integers: ");
  scanf("%d%d", &i, &j);

  answer = j ? i/j: 0;
  printf("%d", answer);

  return 0;
}
```

2. `count = a>b ? 100 : 0;`

EXERCISES

2. `x &= y;`

3.
```
#include <stdio.h>

int main(void)
{
  int i;

  for(i=17; i<=1000; i+=17)
    printf("%d\n", i);

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

int main(void)
{
  int i, j, k;

  for(i=0, j=-50, k=i+j; i<100; i++, j++, k=i+j)
    printf("k = %d\n", k);

  return 0;
}
```

2. 3

MASTERY SKILLS CHECK

1. The **register** specifier causes the C compiler to provide the fastest access possible for the variable it precedes.
2. The **const** specifier tells the C compiler that no statement in the program may modify a variable declared as **const**. Also, a **const** pointer parameter may not be used to modify the object pointed to by the pointer. The **volatile** specifier tells the compiler that

any variable it precedes may have its value changed in ways not explicitly specified by the program.

3.
```
#include <stdio.h>

int main(void)
{
  register int i, sum;

  sum = 0;
  for(i=1; i<101; i++)
    sum = sum + i;

  printf("%d", sum);

  return 0;
}
```

4. Yes, the statement is valid. It creates another name for the type **long double**.

5.
```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  char ch1, ch2;
  char mask, i;

  printf("Enter two characters: ");
  ch1 = getche();
  ch2 = getche();
  printf("\n");

  mask = 1;
  for(i=0; i<8; i++) {
    if((mask & ch1) && (mask & ch2))
      printf("bits %d the same\n", i);
    mask <<= 1;
  }

  return 0;
}
```

6. The << and >> are the left and right shift operators, respectively.

7. `c += 10;`

8. `count = done ? 0 : 100;`

9. An enumeration is a list of named integer constants. Here is one that enumerates the planets.

```
enum planets {Mercury, Venus, Earth, Mars, Jupiter,
              Saturn, Neptune, Uranus, Pluto} ;
```

CUMULATIVE SKILLS CHECK

1.
```
#include <stdio.h>

void show_binary(unsigned u);

int main(void)
{
  unsigned char ch, t1, t2;

  ch = 100;
  show_binary(ch);

  t1 = ch;
  t2 = ch;

  t1 <<= 4;
  t2 >>= 4;

  ch = t1 | t2;

  show_binary(ch);

  return 0;
}

void show_binary(unsigned u)
{
  unsigned n;

  for(n=128; n>0; n=n/2)
    if(u & n) printf("1 ");
    else printf("0 ");
    printf("\n");
  }
```

2.
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *in;
  unsigned char ch;

  if(argc!=2) {
    printf("Usage: code <in>\n");
    exit(1);
  }

  if((in = fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open input file.\n");
    exit(1);
  }

  while(!feof(in)) {
    ch = fgetc(in);
    if(!feof(in)) putchar(~ch);
  }

  fclose(in);

  return 0;
}
```

3. Yes, any type of variable can be specified using **register**. However, on some types, it may have no effect.

4.
```
/* A simple computerized telephone book. */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

struct address {
  char street[40];
  char city[40];
  char state[3];
  char zip[12];
};

struct phone_type {
  char name[40];
  int areacode;
  char number[9];
  struct address addr;
} phone[MAX];

int loc =0;

int menu(void);
void enter(void);
void load(void);
void save(void);
void find(void);

int main(void)
{
  register int choice;

  do {
    choice = menu();
    switch(choice) {
      case 1: enter();
        break;
      case 2: find();
        break;
      case 3: save();
        break;
      case 4: load();
    }
  } while(choice!=5);

  return 0;
}

/* Get menu choice. */
menu(void)
{
  register int i;
  char str[80];
  printf("1. Enter names and numbers \n");
  printf("2. Find numbers\n");
  printf("3. Save directory to disk\n");
  printf("4. Load directory from disk\n");
  printf("5. Quit\n");

  do {
    printf("Enter your choice: ");
    gets(str);
    i = atoi(str);
    printf("\n");
  } while(i<1 || i>5);
  return i;
}

void enter(void)
{
  char temp[80];

  for(; loc<100; loc++) {
    if(loc<100) {
      printf("Enter name: ");
      gets(phone[loc].name);
      if(!*phone[loc].name) break;
      printf("Enter area code: ");
      gets(temp);
      phone[loc].areacode = atoi(temp);
      printf("Enter number: ");
      gets(phone[loc].number);

      /* input address info */
      printf("Enter street address: ");
      gets(phone[loc].addr.street);
      printf("Enter city: ");
      gets(phone[loc].addr.city);
      printf("Enter State: ");
      gets(phone[loc].addr.state);
      printf("Enter zip code: ");
      gets(phone[loc].addr.zip);
    }
  }
}

void find(void)
{
  char name[80];
  register int i;

  printf("Enter name: ");
  gets(name);
  if(!*name) return;

  for(i=0; i<100; i++)
    if(!strcmp(name, phone[i].name)) {
      printf("%s (%d) %s\n", phone[i].name,
             phone[i].areacode, phone[i].number);
      printf("%s\n%s %s %s\n", phone[i].addr.street,
             phone[i].addr.city, phone[i].addr.state,
             phone[i].addr.zip);
  }
}

void load(void)
{
  FILE *fp;

  if((fp = fopen("phone", "rb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  loc = 0;
  while(!feof(fp)) {
    fread(&phone[loc], sizeof phone[loc], 1, fp);
    loc++;
  }
  fclose(fp);
}

void save(void)
{
  FILE *fp;
  register int i;

  if((fp = fopen("phone", "wb"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  for(i=0; i<loc; i++) {
      fwrite(&phone[i], sizeof phone[i], 1, fp);
    }
    fclose(fp);
  }
```

CHAPTER 12

REVIEW SKILLS CHECK

1. Modifying a variable with **register** causes the compiler to store the variable in such a way that access to it is as fast as possible. For integer and character types, this typically means storing it in a register of the CPU.
2. Because **i** is declared as **const** the function cannot modify any object pointed to by it.
3. a. 1100 0100

   b. 1111 1111

   c. 0011 1011
4.
```
#include <stdio.h>

int main(void)
{
  int i;

  printf("Enter a number: ");
  scanf("%d", &i);

  printf("Doubled: %d\n", i << 1);
  printf("Halved: %d\n", i >> 1);

  return 0;
}
```
5.
```
a = b = c = 1;

max = a<b ? 100 : 0;

i *= 2;
```

6. The **extern** modifier is principally used to inform the compiler about global variables defined in a different file. Placing **extern** in front of a variable's declaration tells the compiler that the variable is defined elsewhere, but allows the current file to refer to it.

EXERCISES

1.
```
#define RANGE(i, min, max) ((i)<(min)) || ((i)>(max)) ? 1 : 0
```

2.
```
#include <stdio.h>

#define ABS(i) (i)<0 ? -(i) : i

int main(void)
{
  printf("%d %d", ABS(-1), ABS(1));

  return 0;
}
```

EXERCISES

1.
```
#include <stdio.h>

#define INT 0
#define FLOAT 1
#define PWR_TYPE INT

int main(void)
{
  int e;
#if PWR_TYPE==FLOAT
  double base, result;
#elif PWR_TYPE==INT
  int base, result;
#endif

#if PWR_TYPE==FLOAT
  printf("Enter floating point base: ");
  scanf("%lf", &base);
#elif PWR_TYPE==INT
  printf("Enter integer base: ");
  scanf("%d", &base);
#endif
  printf("Enter integer exponent (greater than 0): ");
  scanf("%d", &e);

  result = 1;
  for(; e; e--)
    result = result * base;

#if PWR_TYPE==FLOAT
    printf("Result: %f", result);
#elif PWR_TYPE==INT
    printf("Result: %d", result);
#endif

  return 0;
}
```

2. No. You cannot use an expression like **!MIKE** with **#ifdef**. Here are two possible solutions.

```
#ifndef MIKE
.
.
.
#endif

/* or */

#if !defined MIKE
.
.
.
#endif
```

EXERCISES

2. The program displays **one two**.

EXERCISES

2.
```
#include <stdio.h>
#include <stdlib.h>

int comp(const void *i, const void *j);

int main(void)
{
  int sort[100], i, key;
  int *p;

  for(i=0; i<100; i++)
    sort[i] = rand();

  qsort(sort, 100, sizeof(int), comp);

  for(i=0; i<100; i++)
    printf("%d\n", sort[i]);

  printf("Enter number to find: ");
  scanf("%d", &key);
  p = bsearch(&key, sort, 100, sizeof(int), comp);
  if(p) printf("Number is in array.\n");
  else printf("Number not found.\n");

  return 0;
}

int comp(const void *i, const void *j)
{
  return *(int*)i - *(int*)j;
}
```

3.
```
#include <stdio.h>

int sum(int a, int b);
int subtract(int a, int b);
int mul(int a, int b);
int div(int a, int b);
int modulus(int a, int b);

/* initialize the pointer array */
int(*p[5]) (int x, int y) = {
  sum, subtract, mul, div, modulus
} ;

int main(void)
{
  int result;
  int i, j, op;

  printf("Enter two numbers: ");
  scanf("%d%d", &i, &j);
  printf("0: add, 1: subtract, 2: multiply, 3: divide, ");
  printf("4: modulus\n");
  do {
    printf("Enter number of operation: ");
    scanf("%d", &op);
  } while(op<0 || op>4);

  result = (*p[op]) (i, j);
  printf("%d", result);

  return 0;
}

int sum(int a, int b)
{
  return a+b;
}

int subtract(int a, int b)
{
  return a-b;
}

int mul(int a, int b)
{
  return a*b;
}

int div(int a, int b)
{
  if(b) return a/b;
  else return 0;
}
```

```
int modulus(int a, int b)
{
  if(b) return a%b;
  else return 0;
}
```

EXERCISES

2.
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int *p, i;

  p = malloc(10*sizeof(int));
  if(!p) {
    printf("Allocation Error");
    exit(1);
  }

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

  for(i=0; i<10; i++) printf("%d ", *(p+i));

  free(p);

  return 0;
}
```

3. The statement

```
*p = malloc(10);
```

should be

```
p = malloc(10);
```

Also, the value returned by **malloc( )** is not verified as a valid pointer.

MASTERY SKILLS CHECK

1. When you specify the file name within angle brackets, the compiler searches for the file in an implementation-defined manner. When you enclose the file name within double quotes, the compiler first tries some other implementation-defined manner to find the file. If that fails, it restarts the search as if you had enclosed the file name within angle brackets.

2.
   ```
   #ifdef DEBUG
   if(!(j%2)) {
     printf("j = %d\n", j);
     j = 0;
   }
   #endif
   ```

3.
   ```
   #if DEBUG==1
   if(!(j%2)) {
     printf("j = %d\n", j);
     j = 0;
   }
   #endif
   ```

4. To undefine a macro name use **#undef**.

5. **__FILE__** is a predefined macro that contains the name of the source file currently being compiled.

6. The # operator makes the argument it precedes into a quoted string. The ## operator concatenates two arguments.

7.
   ```
   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>

   int comp(const void *i, const void *j);

   int main(void)
   {
     char str[] = "this is a test of qsort";

     qsort(str, strlen(str), 1, comp);

     printf(str);
   ```

```
  return 0;
}

int comp(const void *i, const void *j)
{
  return *(char*)i - *(char*)j;
}
```

8.
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double *p;

  p = malloc(sizeof(double));
  if(!p) {
    printf("Allocation Error");
    exit(1);
  }

  *p = 99.01;
  printf("%f", *p);
  free(p);

  return 0;
}
```

CUMULATIVE SKILLS CHECK

1.
```
/* An electronic card catalog. */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

int menu(void);
void display(int i);
void author_search(void);
void title_search(void);
void enter(void);
void save(void);
void load(void);
```

```
struct catalog {
  char name[80];      /* author name */
  char title[80];     /* title */
  char pub[80];       /* publisher */
  unsigned date;      /* date of publication */
  unsigned char ed; /* edition */
} *cat[MAX]; /* notice that this declares a pointer array */

int top = 0; /* last location used */

int main(void)
{
  int choice;

  load(); /* read in catalog */

  do {
    choice = menu();
    switch(choice) {
      case 1: enter(); /* enter books */
        break;
      case 2: author_search(); /* search by author */
        break;
      case 3: title_search(); /* search by title */
        break;
      case 4: save();
    }
  } while(choice!=5);

  return 0;
}

/* Return a menu selection. */
int menu(void)
{
  int i;
  char str[80];

  printf("Card Catalog:\n");
  printf(" 1. Enter\n");
  printf(" 2. Search by Author\n");
  printf(" 3. Search by Title\n");
  printf(" 4. Save catalog\n");
  printf(" 5. Quit\n");
  do {
    printf("Choose your selection: ");
    gets(str);
    i = atoi(str);
    printf("\n");
  } while(i<1 || i>5);

  return i;
}

/* Enter books into database. */
void enter(void)
{
  int i;
  char temp[80];

  for(i=top; i<MAX; i++){
    /* allocate memory for book info */
    cat[i] = malloc(sizeof(struct catalog));
    if(!cat[i]) {
      printf("Out of memory.\n");
      return;
    }

    printf("Enter author name (ENTER to quit): ");
    gets(cat[i]->name);
    if(!*cat[i]->name) break;
    printf("Enter title: ");
    gets(cat[i]->title);
    printf("Enter publisher: ");
    gets(cat[i]->pub);
    printf("Enter copyright date: ");
    gets(temp);
    cat[i]->date = (unsigned) atoi(temp);
    printf("Enter edition: ");
    gets(temp);
    cat[i]->ed = (unsigned char) atoi(temp);
  }
  top = i;
}

/* Search by author. */
void author_search(void)
{
  char name[80];
  int i, found;

  printf("Name: ");
  gets(name);
  found = 0;
  for( i=0; i<top; i++)
    if(!strcmp(name, cat[i]->name)) {
      display(i);
      found = 1;
      printf("\n");
    }

  if(!found) printf("Not Found\n");
}

/* Search by title. */
void title_search(void)
{
  char title[80];
  int i, found;

  printf("Title: ");
  gets(title);

  found = 0;
  for(i=0; i<top; i++)
    if(!strcmp(title, cat[i]->title)) {
      display(i);
      found = 1;
      printf("\n");
  }
  if(!found) printf("Not Found\n");
}

/* Display catalog entry. */
void display(int i)
{
  printf("%s\n", cat[i]->title);
  printf("by %s\n", cat[i]->name);
  printf("Published by %s\n", cat[i]->pub);
  printf("Copyright: %u, %u edition\n", cat[i]->date,
         cat[i]->ed);
}
```

```
/* Load the catalog file. */
void load(void)
{
  FILE *fp;
  int i;

  if((fp = fopen("catalog", "rb"))==NULL) {
    printf("Catalog file not on disk.\n");
    return;
  }


  if(fread(&top, sizeof top, 1, fp) != 1) {  /* read count */
    printf("Error reading count.\n");
    exit(1);
  }

  for(i=0; i<top; i++) {
    cat[i] = malloc(sizeof(struct catalog));
    if(!cat[i]) {
      printf("Out of memory.\n");
      top = i-1;
      break;
    }
    if(fread(cat[i], sizeof(struct catalog), 1, fp)!= 1) {
      printf("Error reading catalog data.\n");
      exit(1);
    }
  }

  fclose(fp);
}

/* Save the catalog file. */
void save(void)
{
  FILE *fp;
  int i;

  if((fp = fopen("catalog", "wb"))==NULL) {
    printf("Cannot open catalog file.\n");
    exit(1);
  }
```

```
    if(fwrite(&top, sizeof top, 1, fp) != 1) {  /* write count *
      printf("Error writing count.\n");
      exit(1);
    }

    for(i=0; i<top; i++)
      if(fwrite(cat[i], sizeof(struct catalog), 1, fp)!= 1)
        printf("Error writing catalog data.\n");
        exit(1);
      }

    fclose(fp);
  }
```

2.
```
#include <stdio.h>

#define CODE_IT(ch) ~ch

int main(void)
{
  int ch;
  printf("Enter a character: ");
  ch = getchar();
  printf("%c coded is %c", ch, CODE_IT(ch));

  return 0;
}
```