Chapter 19 Exploring the C Function Library

19-1. Mathematical Functions

The C standard library contains a variety of functions that perform mathematical operations. Prototypes for the mathematical functions are in the header file MATH.H. The math functions all return a type double. For the trigonometric functions, angles are expressed in radians. Remember, one radian equals 57.296 degrees, and a full circle (360 degrees) contains 2p radians.

19-1-1. Trigonometric Functions

The trigonometric functions perform calculations that are used in some graphical and engineering applications.

Function	*Prototype*	*Description*
acos()	double acos(double x)	Returns the arccosine of its argument. The argument must be in the range -1 <= x <= 1, and the return value is in the range 0 <= acos <= p.
asin()	double asin(double x)	Returns the arcsine of its argument. The argument must be in the range -1 <= x <= 1, and the return value is in the range -p/2 <= asin <= p/2.
atan()	double atan(double x)	Returns the arctangent of its argument. The return value is in the range -p/2 <= atan <= p/2.
atan2()	double atan2(double x, double y)	Returns the arctangent of x/y. The value returned is in the range -p <= atan2 <= p.
cos()	double cos(double x)	Returns the cosine of its argument.
sin()	double sin(double x)	Returns the sine of its argument.
tan()	double tan(double x)	Returns the tangent of its argument.

19-1-2. Exponential and Logarithmic Functions

The exponential and logarithmic functions are needed for certain types of mathematical calculations.

Function	*Prototype*	*Description*
exp()	double exp(double x)	Returns the natural exponent of its argument, that is, e^x where e equals 2.7182818284590452354.
log()	double log(double x)	Returns the natural logarithm of its argument. The argument must be greater than 0.
log10()	double log10(double x)	Returns the base-10 logarithm of its argument. The argument must be greater than 0.
frexp()	double frexp(double x, int *y)	The function calculates the normalized fraction representing the value x. The function's return value r is a fraction in the range 0.5 <= r <= 1.0. The function assigns to y an integer exponent such that x = r * 2y. If the value passed to the function is 0, both r and y are 0.
ldexp()	double ldexp(double x,	Returns x * 2y.

	int y)	

19-1-3. Hyperbolic Functions

The hyperbolic functions perform hyperbolic trigonometric calculations.

Function	*Prototype*	*Description*
cosh()	double cosh(double x)	Returns the hyperbolic cosine of its argument.
sinh()	double sinh(double x)	Returns the hyperbolic sine of its argument.
tanh()	double tanh(double x)	Returns the hyperbolic tangent of its argument.

19-1-4. Other Mathematical Functions

The standard C library contains the following miscellaneous mathematical functions:

Function	*Prototype*	*Description*
sqrt()	double sqrt(double x)	Returns the square root of its argument. The argument must be zero or greater.
ceil()	double ceil(double x)	Returns the smallest integer not less than its argument. For example, ceil(4.5) returns 5.0, and ceil(-4.5) returns -4.0. Although ceil() returns an integer value, it is returned as a type double.
abs()	int abs(int x)	Returns the absolute
labs()	long labs(long x)	value of their arguments.
floor()	double floor(double x)	Returns the largest integer not greater than its argument. For example, floor(4.5) returns 4.0, and floor(-4.5) returns -5.0.
modf()	double modf(double x, double *y)	Splits x into integral and fractional parts, each with the same sign as x. The fractional part is returned by the function, and the integral part is assigned to *y.
pow()	double pow(double x, double y)	Returns xy. An error occurs if x == 0 and y <= 0, or if x < 0 and y is not an integer.
fmod()	double fmod(double x, double y)	Returns the floating-point remainder of x/y, with the same sign as x. The function returns 0 if x == 0.

19-1-5. A Demonstration of the Math Functions

An entire book could be filled with programs demonstrating all of the math functions. Listing 19.1 contains a single program that demonstrates several of these functions.

Listing 19.1. Using the C library math functions.

```
1: /* Demonstrates some of C's math functions */
2:
3: #include <stdio.h>
4: #include <math.h>
```

```
5:
6: main()
7: {
8:
9:     double x;
10:
11:     printf("Enter a number: ");
12:     scanf( "%lf", &x);
13:
14:     printf("\n\nOriginal value: %lf", x);
15:
16:     printf("\nCeil: %lf", ceil(x));
17:     printf("\nFloor: %lf", floor(x));
18:     if( x >= 0 )
19:         printf("\nSquare root: %lf", sqrt(x) );
20:     else
21:        printf("\nNegative number" );
22:
23:     printf("\nCosine: %lf\n", cos(x));
24:     return(0);
25: }
Enter a number: 100.95
Original value: 100.950000
Ceil: 101.000000
Floor: 100.000000
Square root: 10.047388
Cosine: 0.913482
```

ANALYSIS: This listing uses just a few of the math functions. A value accepted on line 12 is printed. Then it's passed to four of the C library math functions--ceil(), floor(), sqrt(), and cos(). Notice that sqrt() is called only if the number isn't negative. By definition, negative numbers don't have square roots. You can add any of the other math functions to a program such as this to test their functionality.

19-2. Dealing with Time

The C library contains several functions that let your program work with times. In C, the term *times* refers to dates as well as times. The function prototypes and the definition of the structure used by many of the time functions are in the header file TIME.H.

19-2-1. Representing Time

The C time functions represent time in two ways. The more basic method is the number of seconds elapsed since midnight on January 1, 1970. Negative values are used to represent times before that date.

These time values are stored as type long integers. In TIME.H, the symbols time_t and clock_t are both defined with a typedef statement as long. These symbols are used in the time function prototypes rather than long.

The second method represents a time broken down into its components: year, month, day, and so on. For this kind of time representation, the time functions use a structure tm, defined in TIME.H as follows:

```
struct tm {
int tm_sec;     /* seconds after the minute - [0,59]  */
int tm_min;     /* minutes after the hour - [0,59]    */
```

```
int tm_hour;    /* hours since midnight - [0,23]  */
int tm_mday;    /* day of the month - [1,31]      */
int tm_mon;     /* months since January - [0,11]      */
int tm_year;    /* years since 1900               */
int tm_wday;    /* days since Sunday - [0,6]      */
int tm_yday;    /* days since January 1 - [0,365] */
int tm_isdst;   /* daylight savings time flag     */
};
```

19-2-2. The Time Functions

This section describes the various C library functions that deal with time. Remember that the term *time* refers to the date as well as hours, minutes, and seconds. A demonstration program follows the descriptions.

Obtaining the Current Time

To obtain the current time as set on your system's internal clock, use the time() function. The prototype is

```
time_t time(time_t *timeptr);
```

Remember, time_t is defined in TIME.H as a synonym for long. The function time() returns the number of seconds elapsed since midnight, January 1, 1970. If it is passed a non-NULL pointer, time() also stores this value in the type time_t variable pointed to by timeptr. Thus, to store the current time in the type time_t variable now, you could write

```
time_t now;
now = time(0);
```

You also could write

```
time_t now;
time_t *ptr_now = &now;
time(ptr_now);
```

Converting Between Time Representations

Knowing the number of seconds since January 1, 1970, is not often useful. Therefore, C provides the ability to convert time represented as a time_t value to a tm structure, using the localtime() function. A tm structure contains day, month, year, and other time information in a format more appropriate for display and printing. The prototype of this function is

```
struct tm *localtime(time_t *ptr);
```

This function returns a pointer to a static type tm structure, so you don't need to declare a type tm structure to use--only a pointer to type tm. This static structure is reused and overwritten each time localtime() is called; if you want to save the value returned, your program must declare a separate type tm structure and copy the values from the static structure.

The reverse conversion--from a type tm structure to a type time_t value--is performed by the function mktime(). The prototype is

```
time_t mktime(struct tm *ntime);
```

This function returns the number of seconds between midnight, January 1, 1970, and the time represented by the type tm structure pointed to by ntime.

Displaying Times

To convert times into formatted strings appropriate for display, use the functions ctime() and asctime(). Both of these functions return the time as a string with a specific format. They differ because ctime() is passed the time as a type time_t value, whereas asctime() is passed the time as a type tm structure. Their prototypes are

```
char *asctime(struct tm *ptr);
char *ctime(time_t *ptr);
```

Both functions return a pointer to a static, null-terminated, 26-character string that gives the time of the function's argument in the following format:

```
Thu Jun 13 10:22:23 1991
```

The time is formatted in 24-hour "military" time. Both functions use a static string, overwriting it each time they're called.

For more control over the format of the time, use the strftime() function. This function is passed a time as a type tm structure. It formats the time according to a format string. The function prototype is

```
size_t strftime(char *s, size_t max, char *fmt, struct tm *ptr);
```

This function takes the time in the type tm structure pointed to by ptr, formats it according to the format string fmt, and writes the result as a null-terminated string to the memory location pointed to by s. The argument max should specify the amount of space allocated at s. If the resulting string (including the terminating null character) has more than max characters, the function returns 0, and the string s is invalid. Otherwise, the function returns the number of characters written--strlen(s).

The format string consists of one or more conversion specifiers from Table 19.1.

Table 19.1. Conversion specifiers that can be used with strftime().

Specifier	What It's Replaced By
%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%c	Date and time representation (for example, 10:41:50 30-Jun-91).
%d	Day of month as a decimal number 01 through 31.
%H	The hour (24-hour clock) as a decimal number 00 through 23.
%I	The hour (12-hour clock) as a decimal number 00 through 11.
%j	The day of the year as a decimal number 001 through 366.
%m	The month as a decimal number 01 through 12.
%M	The minute as a decimal number 00 through 59.
%p	AM or PM.
%S	The second as a decimal number 00 through 59.
%U	The week of the year as a decimal number 00 through 53. Sunday is considered the first day of the week.

%w	The weekday as a decimal number 0 through 6 (Sunday = 0).
%W	The week of the year as a decimal number 00 through 53. Monday is considered the first day of the week.
%x	The date representation (for example, 30-Jun-91).
%X	The time representation (for example, 10:41:50).
%y	The year, without century, as a decimal number 00 through 99.
%Y	The year, with century, as a decimal number.
%Z	The time zone name if the information is available or blank if not.
%%	A single percent sign %.

Calculating Time Differences

You can calculate the difference, in seconds, between two times with the difftime() macro, which subtracts two time_t values and returns the difference. The prototype is

```
double difftime(time_t later, time_t earlier);
```

This function subtracts earlier from later and returns the difference, the number of seconds between the two times. A common use of difftime() is to calculate elapsed time, as demonstrated (along with other time operations) in Listing 19.2.

You can determine duration of a different sort using the clock() function, which returns the amount of time that has passed since the program started execution, in 1/100-second units. The prototype is

```
clock_t clock(void);
```

To determine the duration of some portion of a program, call clock() twice--before and after the process occurs--and subtract the two return values.

19-2-3. Using the Time Functions

Listing 19.2 demonstrates how to use the C library time functions.

Listing 19.2. Using the C library time functions.

```
1: /* Demonstrates the time functions. */
2:
3: #include <stdio.h>
4: #include <time.h>
5:
6: main()
7: {
8:      time_t start, finish, now;
9:      struct tm *ptr;
10:      char *c, buf1[80];
11:      double duration;
12:
13:      /* Record the time the program starts execution. */
14:
15:      start = time(0);
```

```
16:
17:      /* Record the current time, using the alternate method of */
18:      /* calling time(). */
19:
20:      time(&now);
21:
22:      /* Convert the time_t value into a type tm structure. */
23:
24:      ptr = localtime(&now);
25:
26:      /* Create and display a formatted string containing */
27:      /* the current time. */
28:
29:      c = asctime(ptr);
30:      puts(c);
31:      getc(stdin);
32:
33:      /* Now use the strftime() function to create several
different */
34:      /* formatted versions of the time. */
35:
36:      strftime(buf1, 80, "This is week %U of the year %Y", ptr);
37:      puts(buf1);
38:      getc(stdin);
39:
40:      strftime(buf1, 80, "Today is %A, %x", ptr);
41:      puts(buf1);
42:      getc(stdin);
43:
44:      strftime(buf1, 80, "It is %M minutes past hour %I.", ptr);
45:      puts(buf1);
46:      getc(stdin);
47:
48:      /* Now get the current time and calculate program duration.
*/
49:
50:      finish = time(0);
51:      duration = difftime(finish, start);
52:      printf("\nProgram execution time using time() = %f seconds.",
¬duration);
53:
54:      /* Also display program duration in hundredths of seconds */
55:      /* using clock(). */
56:
57:      printf("\nProgram execution time using clock() = %ld
hundredths of               ¬sec.",
58:          clock());
59:      return(0);
60: }
Mon Jun 09 13:53:33 1997
This is week 23 of the year 1997
```

```
Today is Monday, 06/09/97
It is 53 minutes past hour 01.
Program execution time using time() = 4.000000 seconds.
Program execution time using clock() = 4170 hundredths of sec.
```

ANALYSIIS: This program has numerous comment lines, so it should be easy to follow. Because the time functions are being used, the TIME.H header file is included on line 4. Line 8 declares three variables of type time_t--start, finish, and now. These variables can hold the time as an offset from January 1, 1970, in seconds. Line 9 declares a pointer to a tm structure. The tm structure was described earlier. The rest of the variables have types that should be familiar to you.

The program records its starting time on line 15. This is done with a call to time(). The program then does virtually the same thing in a different way. Instead of using the value returned by the time() function, line 20 passes time() a pointer to the variable now. Line 24 does exactly what the comment on line 22 states: It converts the time_t value of now to a type tm structure. The next few sections of the program print the value of the current time to the screen in various formats. Line 29 uses the asctime() function to assign the information to a character pointer, c. Line 30 prints the formatted information. The program then waits for the user to press Enter.

Lines 36 through 46 use the strftime() function to print the date in three different formats. Using Table 19.1, you should be able to determine what these lines print.

The program then determines the time again on line 50. This is the program-ending time. Line 51 uses this ending time along with the starting time to calculate the program's duration by means of the difftime() function. This value is printed on line 52. The program concludes by printing the program execution time from the clock() function.

19-3. Error-Handling Functions

The C standard library contains a variety of functions and macros that help you deal with program errors.

19-3-1. The assert() Function

The macro assert() can diagnose program bugs. It is defined in ASSERT.H, and its prototype is

```
void assert(int expression);
```

The argument *expression* can be anything you want to test--a variable or any C expression. If *expression* evaluates to TRUE, assert() does nothing. If *expression* evaluates to FALSE, assert() displays an error message on stderr and aborts program execution.

How do you use assert()? It is most frequently used to track down program bugs (which are distinct from compilation errors). A bug doesn't prevent a program from compiling, but it causes it to give incorrect results or to run improperly (locking up, for example). For instance, a financial-analysis program you're writing might occasionally give incorrect answers. You suspect that the problem is caused by the variable interest_rate taking on a negative value, which should never happen. To check this, place the statement

```
assert(interest_rate >= 0);
```

at locations in the program where interest_rate is used. If the variable ever does become negative, the assert() macro alerts you. You can then examine the relevant code to locate the cause of the problem.

To see how assert() works, run Listing 19.3. If you enter a nonzero value, the program displays the value and terminates normally. If you enter zero, the assert() macro forces abnormal program termination. The exact error message you see will depend on your compiler, but here's a typical example:

```
Assertion failed: x, file list19_3.c, line 13
```

Note that, in order for assert() to work, your program must be compiled in debug mode. Refer to your compiler documentation for information on enabling debug mode (as explained in a moment). When you later compile the final version in release mode, the assert() macros are disabled.

Listing 19.3. Using the assert() macro.

```
1: /* The assert() macro. */
2:
3: #include <stdio.h>
4: #include <assert.h>
5:
6: main()
7: {
8:     int x;
9:
10:     printf("\nEnter an integer value: ");
11:     scanf("%d", &x);
12:
13:     assert(x >= 0);
14:
15:     printf("You entered %d.\n", x);
16:     return(0);
17: }
Enter an integer value: 10
You entered 10.
Enter an integer value: -1
Assertion failed: x, file list19_3.c, line 13
Abnormal program termination
```

Your error message might differ, depending on your system and compiler, but the general idea is the same.

ANALYSIS: Run this program to see that the error message displayed by assert() on line 13 includes the expression whose test failed, the name of the file, and the line number where the assert() is located.

The action of assert() depends on another macro named NDEBUG (which stands for "no debugging"). If the macro NDEBUG isn't defined (the default), assert() is active. If NDEBUG is defined, assert() is turned off and has no effect. If you placed assert() in various program locations to help with debugging and then solved the problem, you can define NDEBUG to turn assert() off. This is much easier than going through the program and removing the assert() statements (only to discover later that you want to use them again). To define the macro NDEBUG, use the #define directive. You can demonstrate this by adding the line

```
#define NDEBUG
```

to Listing 19.3, on line 2. Now the program prints the value entered and then terminates normally, even if you enter -1.

Note that NDEBUG doesn't need to be defined as anything in particular, as long as it's included in a #define directive. You'll learn more about the #define directive on Day 21, "Advanced Compiler Use."

19-3-2. The ERRNO.H Header File

The header file ERRNO.H defines several macros used to define and document runtime errors. These macros are used in conjunction with the perror() function, described in the next section.

The ERRNO.H definitions include an external integer named errno. Many of the C library functions assign a value to this variable if an error occurs during function execution. The file ERRNO.H also defines a group of symbolic constants for these errors, listed in Table 19.2.

Table 19.2. The symbolic error constants defined in ERRNO.H.

Name	Value	Message and Meaning
E2BIG	1000	Argument list too long (list length exceeds 128 bytes).
EACCES	5	Permission denied (for example, trying to write to a file opened for read only).
EBADF	6	Bad file descriptor.
EDOM	1002	Math argument out of domain (an argument passed to a math function was outside the allowable range).
EEXIST	80	File exists.
EMFILE	4	Too many open files.
ENOENT	2	No such file or directory.
ENOEXEC	1001	Exec format error.
ENOMEM	8	Not enough core (for example, not enough memory to execute the exec() function).
ENOPATH	3	Path not found.
ERANGE	1003	Result out of range (for example, result returned by a math function is too large or too small for the return data type).

You can use errno two ways. Some functions signal, by means of their return value, that an error has occurred. If this happens, you can test the value of errno to determine the nature of the error and take appropriate action. Otherwise, when you have no specific indication that an error occurred, you can test errno. If it's nonzero, an error has occurred, and the specific value of errno indicates the nature of the error. Be sure to reset errno to zero after handling the error. The next section explains perror(), and then Listing 19.4 illustrates the use of errno.

19-3-3. The perror() Function

The perror() function is another of C's error-handling tools. When called, perror() displays a message on stderr describing the most recent error that occurred during a library function call or system call. The prototype, in STDIO.H, is

```
void perror(char *msg);
```

The argument msg points to an optional user-defined message. This message is printed first, followed by a colon and the implementation-defined message that describes the most recent error. If you call perror() when no error has occurred, the message displayed is no error.

A call to perror() does nothing to deal with the error condition. It's up to the program to take action, which might consist of prompting the user to do something such as terminate the program. The action the program takes can be determined by testing the value of errno and by the nature of the error. Note that a program need not include the header file ERRNO.H to use the external variable errno. That header file is required only if your program uses the symbolic error constants listed in Table 19.2. Listing 19.4 illustrates the use of perror() and errno for handling runtime errors.

Listing 19.4. Using perror() and errno to deal with runtime errors.

```
1: /* Demonstration of error handling with perror() and errno. */
```

```
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <errno.h>
6:
7: main()
8: {
9:     FILE *fp;
10:     char filename[80];
11:
12:     printf("Enter filename: ");
13:     gets(filename);
14:
15:     if (( fp = fopen(filename, "r")) == NULL)
16:     {
17:         perror("You goofed!");
18:         printf("errno = %d.\n", errno);
19:         exit(1);
20:     }
21:     else
22:     {
23:         puts("File opened for reading.");
24:         fclose(fp);
25:     }
26:     return(0);
27: }
Enter file name: list19_4.c
File opened for reading.
Enter file name: notafile.xxx
You goofed!: No such file or directory
errno = 2.
```

ANALYSIS: This program prints one of two messages based on whether a file can be opened for reading. Line 15 tries to open a file. If the file opens, the else part of the if loop executes, printing the following message:

```
File opened for reading.
```

If there is an error when the file is opened, such as the file not existing, lines 17 through 19 of the if loop execute. Line 17 calls the perror() function with the string "You goofed!". This is followed by printing the error number. The result of entering a file that does not exist is

```
You goofed!: No such file or directory.
errno = 2
```

DO include the ERRNO.H header file if you're going to use the symbolic errors listed in Table 19.2.

DON'T include the ERRNO.H header file if you aren't going to use the symbolic error constants listed in Table 19.2.

DO check for possible errors in your programs. Never assume that everything is okay.

19-4. Searching and Sorting

Among the most common tasks that programs perform are searching and sorting data. The C standard library contains general-purpose functions that you can use for each task.

19-4-1. Searching with bsearch()

The library function bsearch() performs a binary search of a data array, looking for an array element that matches a key. To use bsearch(), the array must be sorted into ascending order. Also, the program must provide the comparison function used by bsearch() to determine whether one data item is greater than, less than, or equal to another item. The prototype of bsearch() is in STDLIB.H:

```
void *bsearch(void *key, void *base, size_t num, size_t width,
int (*cmp)(void *element1, void *element2));
```

This is a fairly complex prototype, so go through it carefully. The argument key is a pointer to the data item being searched for, and base is a pointer to the first element of the array being searched. Both are declared as type void pointers, so they can point to any of C's data objects.

The argument num is the number of elements in the array, and width is the size (in bytes) of each element. The type specifier size_t refers to the data type returned by the sizeof() operator, which is unsigned. The sizeof() operator is usually used to obtain the values for num and width.

The final argument, cmp, is a pointer to the comparison function. This can be a user-written function or, when searching string data, it can be the library function strcmp(). The comparison function must meet the following two criteria:

- It is passed pointers to two data items
- It returns a type int as follows:
- < 0 Element 1 is less than element 2.
- 0 Element 1 is equal to element 2.
- > 0 Element 1 is greater than element 2.

The return value of bsearch() is a type void pointer. The function returns a pointer to the first array element it finds that matches the key, or NULL if no match is found. You must cast the returned pointer to the proper type before using it.

The sizeof() operator can provide the num and width arguments as follows. If array[] is the array to be searched, the statement

```
sizeof(array[0]);
```

returns the value of width--the size (in bytes) of one array element. Because the expression sizeof(array) returns the size, in bytes, of the entire array, the following statement obtains the value of num, the number of elements in the array:

```
sizeof(array)/sizeof(array[0])
```

The binary search algorithm is very efficient; it can search a large array quickly. Its operation is dependent on the array elements being in ascending order. Here's how the algorithm works:

1. The key is compared to the element at the middle of the array. If there's a match, the search is done. Otherwise, the key must be either less than or greater than the array element.

2. If the key is less than the array element, the matching element, if any, must be located in the first half of the array. Likewise, if the key is greater than the array element, the matching element must be located in the second half of the array.
3. The search is restricted to the appropriate half of the array, and then the algorithm returns to step 1.

You can see that each comparison performed by a binary search eliminates half of the array being searched. For example, a 1,000-element array can be searched with only 10 comparisons, and a 16,000-element array can be searched with only 14 comparisons. In general, a binary search requires n comparisons to search an array of 2^n elements.

19-4-2. Sorting with qsort()

The library function qsort() is an implementation of the quicksort algorithm, invented by C.A.R. Hoare. This function sorts an array into order. Usually the result is in ascending order, but qsort() can be used for descending order as well. The function prototype, defined in STDLIB.H, is

```
void qsort(void *base, size_t num, size_t size,
int (*cmp)(void *element1, void *element2));
```

The argument base points at the first element in the array, num is the number of elements in the array, and size is the size (in bytes) of one array element. The argument cmp is a pointer to a comparison function. The rules for the comparison function are the same as for the comparison function used by bsearch(), described in the preceding section: You often use the same comparison function for both bsearch() and qsort(). The function qsort() has no return value.

19-4-3. Searching and Sorting: Two Demonstrations

Listing 19.5 demonstrates the use of qsort() and bsearch(). The program sorts and searches an array of values. Note that the non-ANSI function getch() is used. If your compiler doesn't support it, you should replace it with the ANSI standard function getchar().

Listing 19.5. Using the qsort() and bsearch() functions with values.

```
1: /* Using qsort() and bsearch() with values.*/
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5:
6: #define MAX 20
7:
8: int intcmp(const void *v1, const void *v2);
9:
10: main()
11: {
12:      int arr[MAX], count, key, *ptr;
13:
14:      /* Enter some integers from the user. */
15:
16:      printf("Enter %d integer values; press Enter after each.\n",
MAX);
17:
18:      for (count = 0; count < MAX; count++)
19:          scanf("%d", &arr[count]);
20:
```

```
21:      puts("Press Enter to sort the values.");
22:      getc(stdin);
23:
24:      /* Sort the array into ascending order. */
25:
26:      qsort(arr, MAX, sizeof(arr[0]), intcmp);
27:
28:      /* Display the sorted array. */
29:
30:      for (count = 0; count < MAX; count++)
31:          printf("\narr[%d] = %d.", count, arr[count]);
32:
33:      puts("\nPress Enter to continue.");
34:      getc(stdin);
35:
36:      /* Enter a search key. */
37:
38:      printf("Enter a value to search for: ");
39:      scanf("%d", &key);
40:
41:      /* Perform the search. */
42:
43:      ptr = (int *)bsearch(&key, arr, MAX, sizeof(arr[0]),intcmp);
44:
45:      if ( ptr != NULL )
46:          printf("%d found at arr[%d].", key, (ptr - arr));
47:      else
48:          printf("%d not found.", key);
49:      return(0);
50: }
51:
52: int intcmp(const void *v1, const void *v2)
53: {
54:      return (*(int *)v1 - *(int *)v2);
55: }
Enter 20 integer values; press Enter after each.
45
12
999
1000
321
123
2300
954
1968
12
2
1999
1776
1812
1456
```

```
1
9999
3
76
200
Press Enter to sort the values.
arr[0] = 1.
arr[1] = 2.
arr[2] = 3.
arr[3] = 12.
arr[4] = 12.
arr[5] = 45.
arr[6] = 76.
arr[7] = 123.
arr[8] = 200.
arr[9] = 321.
arr[10] = 954.
arr[11] = 999.
arr[12] = 1000.
arr[13] = 1456.
arr[14] = 1776.
arr[15] = 1812.
arr[16] = 1968.
arr[17] = 1999.
arr[18] = 2300.
arr[19] = 9999.
Press Enter to continue.
Enter a value to search for:
1776
1776 found at arr[14]
```

ANALLYSIS: Listing 19.5 incorporates everything described previously about sorting and searching. This program lets you enter up to MAX values (20 in this case). It sorts the values and prints them in order. Then it lets you enter a value to search for in the array. A printed message states the search's status.

Familiar code is used to obtain the values for the array on lines 18 and 19. Line 26 contains the call to qsort() to sort the array. The first argument is a pointer to the array's first element. This is followed by MAX, the number of elements in the array. The size of the first element is then provided so that qsort() knows the width of each item. The call is finished with the argument for the sort function, intcmp.

The function intcmp() is defined on lines 52 through 55. It returns the difference of the two values passed to it. This might seem too simple at first, but remember what values the comparison function is supposed to return. If the elements are equal, 0 should be returned. If element one is greater than element two, a positive number should be returned. If element one is less than element two, a negative number should be returned. This is exactly what intcmp() does.

The searching is done with bsearch(). Notice that its arguments are virtually the same as those of qsort(). The difference is that the first argument of bsearch() is the key to be searched for. bsearch() returns a pointer to the location of the found key or NULL if the key isn't found. On line 43, ptr is assigned the returned value of bsearch(). ptr is used in the if loop on lines 45 through 48 to print the status of the search.

Listing 19.6 has the same functionality as Listing 19.5; however, Listing 19.6 sorts and searches strings.

Listing 19.6. Using qsort() and bsearch() with strings.

```
1: /* Using qsort() and bsearch() with strings. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: #define MAX 20
8:
9: int comp(const void *s1, const void *s2);
10:
11: main()
12: {
13:     char *data[MAX], buf[80], *ptr, *key, **key1;
14:     int count;
15:
16:     /* Input a list of words. */
17:
18:     printf("Enter %d words, pressing Enter after each.\n",MAX);
19:
20:     for (count = 0; count < MAX; count++)
21:     {
22:         printf("Word %d: ", count+1);
23:         gets(buf);
24:         data[count] = malloc(strlen(buf)+1);
25:         strcpy(data[count], buf);
26:     }
27:
28:     /* Sort the words (actually, sort the pointers). */
29:
30:     qsort(data, MAX, sizeof(data[0]), comp);
31:
32:     /* Display the sorted words. */
33:
34:     for (count = 0; count < MAX; count++)
35:         printf("\n%d: %s", count+1, data[count]);
36:
37:     /* Get a search key. */
38:
39:     printf("\n\nEnter a search key: ");
40:     gets(buf);
41:
42:     /* Perform the search. First, make key1 a pointer */
43:     /* to the pointer to the search key.*/
44:
45:     key = buf;
46:     key1 = &key;
47:     ptr = bsearch(key1, data, MAX, sizeof(data[0]), comp);
48:
49:     if (ptr != NULL)
50:         printf("%s found.\n", buf);
51:     else
```

```
52:         printf("%s not found.\n", buf);
53:     return(0);
54: }
55:
56: int comp(const void *s1, const void *s2)
57: {
58:     return (strcmp(*(char **)s1, *(char **)s2));
59: }
Enter 20 words, pressing Enter after each.
Word 1: apple
Word 2: orange
Word 3: grapefruit
Word 4: peach
Word 5: plum
Word 6: pear
Word 7: cherries
Word 8: banana
Word 9: lime
Word 10: lemon
Word 11: tangerine
Word 12: star
Word 13: watermelon
Word 14: cantaloupe
Word 15: musk melon
Word 16: strawberry
Word 17: blackberry
Word 18: blueberry
Word 19: grape
Word 20: cranberry
1: apple
2: banana
3: blackberry
4: blueberry
5: cantaloupe
6: cherries
7: cranberry
8: grape
9: grapefruit
10: lemon
11: lime
12: musk melon
13: orange
14: peach
15: pear
16: plum
17: star
18: strawberry
19: tangerine
20: watermelon
Enter a search key: orange
orange found.
```

[bbeg]ANALYSIS: A couple of points about Listing 19.6 bear mentioning. This program makes use of an array of pointers to strings, a technique introduced on Day 15, "Pointers: Beyond the Basics." As you saw in that chapter, you can "sort" the strings by sorting the array of pointers. However, this method requires a modification in the comparison function. This function is passed pointers to the two items in the array that are compared. However, you want the array of pointers sorted based not on the values of the pointers themselves but on the values of the strings they point to.

Because of this, you must use a comparison function that is passed pointers to pointers. Each argument to comp() is a pointer to an array element, and because each element is itself a pointer (to a string), the argument is therefore a pointer to a pointer. Within the function itself, you dereference the pointers so that the return value of comp() depends on the values of the strings pointed to.

The fact that the arguments passed to comp() are pointers to pointers creates another problem. You store the search key in buf[], and you also know that the name of an array (buf in this case) is a pointer to the array. However, you need to pass not buf itself, but a pointer to buf. The problem is that buf is a pointer constant, not a pointer variable. buf itself has no address in memory; it's a symbol that evaluates to the address of the array. Because of this, you can't create a pointer that points to buf by using the address-of operator in front of buf, as in &buf.

What to do? First, create a pointer variable and assign the value of buf to it. In the program, this pointer variable has the name key. Because key is a pointer variable, it has an address, and you can create a pointer that contains that address--in this case, key1. When you finally call bsearch(), the first argument is key1, a pointer to a pointer to the key string. The function bsearch() passes that argument on to comp(), and everything works properly.

DON'T forget to put your search array into ascending order before using bsearch().

19-5. Summary

This chapter explored some of the more useful functions supplied in the C function library. There are functions that perform mathematical calculations, deal with time, and assist your program with error handling. The functions for sorting and searching data are particularly useful; they can save you considerable time when you're writing your programs.

Q&A

Q Why do nearly all of the math functions return doubles?
A The answer to this question is precision, not consistency. A double is more precise than the other variable types; therefore, your answers are more accurate. On Day 20, "Working with Memory," you will learn the specifics of casting variables and variable promotion. These topics are also applicable to precision.
Q Are bsearch() and qsort() the only ways in C to sort and search?
A These two functions are provided in the standard library; however, you don't have to use them. Many computer programming textbooks teach you how to write your own searching and sorting programs. C contains all the commands you need to write your own. You can purchase specially written searching and sorting routines. The biggest benefits of bsearch() and qsort() are that they are already written and that they are provided with any ANSI-compatible compiler.
Q Do the math functions validate bad data?
A Never assume that data entered is correct. Always validate user-entered data. For example, if you pass a negative value to sqrt(), the function generates an error. If you're formatting the output, you probably don't want this error displayed as it is. Remove the if statement in Listing 19.1 and enter a negative number to see what I mean.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the return data type for all of C's mathematical functions?
2. What C variable type is time_t equivalent to?
3. What are the differences between the time() function and the clock() function?
4. When you call the perror() function, what does it do to correct an existing error condition?
5. Before you search an array with bsearch(), what must you do?
6. Using bsearch(), how many comparisons would be required to find an element if the array had 16,000 items?
7. Using bsearch(), how many comparisons would be required to find an element if an array had only 10 items?
8. Using bsearch(), how many comparisons would be required to find an element if an array had 2,000,000 items?
9. What values must a comparison function for bsearch() and qsort() return?
10. What does bsearch() return if it can't find an element in an array?

Exercises

1. Write a call to bsearch(). The array to be searched is called names, and the values are characters. The comparison function is called comp_names(). Assume that all the names are the same size.
2. BUG BUSTER: What is wrong with the following program?

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int values[10], count, key, *ptr;
printf("Enter values");
for( ctr = 0; ctr < 10; ctr++ )
scanf( "%d", &values[ctr] );
qsort(values, 10, compare_function());
}
```

3. BUG BUSTER: Is anything wrong with the following compare function?

```
int intcmp( int element1, int element2)
{
if ( element 1 > element 2 )
return -1;
else if ( element 1 < element2 )
return 1;
else
return 0;
}
```

Answers are not provided for the following exercises:
4. Modify Listing 19.1 so that the sqrt() function works with negative numbers. Do this by taking the absolute value of x.
5. Write a program that consists of a menu that performs various math functions. Use as many of the math functions as you can.
6. Using the time functions discussed in this chapter, write a function that causes the program to pause for approximately five seconds.
7. Add the assert() function to the program in exercise 4. The program should print a message if a negative value is entered.

8. Write a program that accepts 30 names and sorts them using qsort(). The program should print the sorted names.
9. Modify the program in exercise 8 so that if the user enters QUIT, the program stops accepting input and sorts the entered values.
10. Refer to Day 15 for a "brute-force" method of sorting an array of pointers to strings based on the string values. Write a program that measures the time required to sort a large array of pointers with that method and then compares that time with the time required to perform the same sort with the library function qsort().