Click to Print

Welcome to C in 21 Days: Week 3

This version of the course is designed for use with a screen-reading program.
Click here or press H to view information about using this course
Click here or press G to open the Glossary
Click here or press A to take the Skill Assessment
Click here or press S to view your scores
Click here or press X to close this course

Unit 1. Day 15 More on Pointers

On Day 9, "Pointers," you were introduced to the basics of pointers, an important part of the C programming language. Today, you go further, exploring some advanced pointer topics that can add flexibility to your programming.

Today, you learn

- How to declare a pointer to a pointer.
- How to use pointers with multidimensional arrays.
- How to declare arrays of pointers.
- How to declare pointers to functions.

Topic 1.1: Pointers-to-Pointers

Pointers: A Review
As you learned on Day 9, "Pointers," a pointer is a numeric variable with a value that is the address of another variable. Click the Example link.

Example

Using Pointers	Example

You declare a pointer using the indirection operator * (also known as the dereferencing operator).	The declaration `int *ptr;` declares a pointer named ptr that can point to a type int variable.
You then use the address of operator (&) to make the pointer point to a specific variable of the corresponding type.	Assuming x has been declared as a type int variable, the statement `ptr = &x;` assigns the address of x to ptr and makes ptr point to x.

Again using the indirection operator, you can access the pointed-to variable by using its pointer.	Both of the following statements assign the value 12 to x:
	
	`x = 12;`
	`*ptr = 12;`

Creating a Pointer to a Pointer
Because a pointer is itself a numeric variable, it's stored in your computer's memory at a particular address. Therefore, you can create a pointer to a pointer, a variable whose value is the address of a pointer. Click the Example link to find out how:

Example

```
/* x is a type int variable. */
int x = 12;

/* ptr is a pointer to x. */
int *ptr = &x;

/* ptr_to_ptr is a pointer to a pointer to x */
int **ptr_to_ptr = &ptr;
```

Using the Double Indirection Operator
Note the use of a double indirection operator (**) when declaring the pointer-to-pointer. You also use the double indirection operator when accessing the pointed-to variable with a pointer-to-pointer. Click the Example link.

Example

Thus, if

```
int *ptr = &x;
**ptr_to_ptr = &ptr;
```

then the statement

```
**ptr_to_ptr = 12;
```

assigns the value 12 to the variable x, and the statement

```
printf("%d", **ptr_to_ptr);
```

displays the value of x on the screen.

Error: Using a Single Indirection Operator
If you mistakenly use a single indirection operator, you get errors. Click the Example link.

Example

The statement

```
*ptr_to_ptr = 12;
```

assigns the value 12 to ptr, which results in ptr pointing to whatever happens to be stored at address 12. This clearly is a mistake.

Multiple Indirection
When you declare and use a pointer-to-pointer, it is called multiple indirection. The relationships between a variable, a pointer, and a pointer-to-pointer are illustrated in Figure 15.1. There's really no limit to the level of multiple indirection possible—you can have a pointer-to-pointer-to-pointer *ad infinitum,* but there's rarely any advantage to going beyond two levels.

Usage
What can you use pointers-to-pointers for? The most common use involves arrays of pointers, which are covered later in this course. Day 19, "Exploring the Function Library," presents an example of using multiple indirection in Listing 19.5.

Topic 1.2: Pointers and Multidimensional Arrays

Pointers and Arrays: A Review
Day 8, "Numeric Arrays," covers the special relationship between pointers and arrays. Specifically, the name of an array without its following brackets is a pointer to the first element of the array. As a result, it is easier to use pointer notation when accessing certain types of arrays. These earlier examples, however, were limited to single-dimensional arrays. What about multidimensional arrays? Click the Tip button.

DON'T forget to use the double indirection operator (`**`) when declaring a pointer to a pointer.

DON'T forget that a pointer increments by the size of the pointer's type (usually what is being pointed to).

DON'T forget to use parentheses when declaring pointers to arrays.

Declaring a pointer to an array of characters:

```
char (*letters)[26];
```

Declaring an array of pointers to characters:

```
char *letters[26];
```

One Way to Visualize the Array: Row and Column
Remember that a multidimensional array is declared with one set of brackets for each dimension. For example, the statement

```
int multi[2][4];
```

declares a two-dimensional array that contains eight type int variables. You can visualize the array as having a row and column structure—two rows and four columns in this case.

Another Way to Visualize the Array: Array-of-Arrays
There's another way to visualize a multidimensional array, however, and one that is closer to the way C actually handles the arrays. You can consider `multi` a two-element array; each of those two elements is an array of four integers. Click the Example link.

Example

Schematic Representation
Under the array-of-arrays scheme, you visualize a multidimensional array as shown in Figure 15.2.

The following table dissects the array declaration statement `int multi[2][4];` into its component parts. You interpret the components of the declaration as follows:

Component	Description
`multi`	1. Declare an array named `multi`.
`[2]`	2. The array `multi` contains two elements.
`[4]`	3. Each of those two elements contains four elements.
`int`	4. Each of the four elements is a type int.

A multidimensional array declaration is read starting with the array name and moving to the right, one set of brackets at a time. When the last set of brackets (the last dimension) is read, the array's basic data type, to the left of the name, is considered.

Array Names as Pointers
Now, back to the topic of array names as pointers. (This is a unit about pointers, after all!) As with a one-dimensional array, the name of a multidimensional array is a pointer to the first array element. Click the Example link.

Example

Continuing with the example, `multi` is a pointer to the first element of the two-dimensional array that was declared as `int multi[2][4]`. What exactly is the first element of `multi`? It is not the type int variable `multi[0][0]`, as you might think. Remember that `multi` is an array of arrays, so its first element is `multi[0]`, which is an array of four type int variables (one of the two such arrays contained in `multi`).

Now, if `multi[0]` is also an array, does it point to anything? Yes indeed! `multi[0]` points to its first element, `multi[0][0]`. You might wonder why `multi[0]` is a pointer. Remember that the name of an array without brackets is a pointer to the first array element. Well, the term `multi[0]` is the name of the array `multi[0][0]` with the last pair of brackets missing, so it qualifies as a pointer.

Rules for Any Array of n Dimensions
If you're a bit confused at this point, don't worry. This material is difficult to grasp. It might help if you remember the following rules for any array of *n* dimensions:

- The array name followed by *n* pairs of brackets (each pair containing an appropriate index, of course) evaluates as array data (that is, the data stored in the specified array element).
- The array name followed by fewer than *n* pairs of brackets evaluates as a pointer to an array element.

Click the Example link.

Example

In the example, therefore, `multi` evaluates as a pointer, `multi[0]` evaluates as a pointer, and `multi[0][0]` evaluates as array data.

Relationship Between Multidimensional Arrays and Pointers

Now look at what all these pointers actually point to. The program in Listing 15.1 declares a two-dimensional array—similar to those you've been using in the examples, prints the pointers that we've been discussing, and also prints the address of the first array element.

If these pointers have the same value, what is the practical difference between them in terms of your program? Remember from Day 9, "Pointers," that the C compiler "knows" what a pointer is pointing to. To be more exact, the compiler knows the size of the item a pointer is pointing to.

Listing 15.1: The Relationship Between a Multidimensional Array and Pointers

Code

```
/* LIST1501.c: Day 15 Listing 15.1 */
/* Demonstrates pointers and multidimensional */
/* arrays. */

#include <stdio.h>

int multi[2][4];

int main(void) {
  printf("\nmulti = %u", multi);
  printf("\nmulti[0] = %u", multi[0]);
  printf("\n&multi[0][0] = %u", &multi[0][0]);
  return 0;
}
```

Output

```
multi = 1328
multi[0] = 1328
&multi[0][0] = 1328
```

Description

The actual value may not be 1328 on your system, but all three values are the same. The address of the array `multi` is the same as the address of the array `multi[0]`, and both are equal to the address of the first integer in the array, `multi[0][0]`.

Example Program: Size of Array Elements

What are the sizes of the elements you have been using? Listing 15.2 uses the operator sizeof() to display the sizes, in bytes, of these elements.

Listing 15.2: Determining the Sizes of the Elements

Code

```
/* LIST1502.c: Day 15 Listing 15.2 */
/* Demonstrates the sizes of multidimensional */
/* array elements. */

#include <stdio.h>

int multi[2][4];

int main(void) {
  printf("\nThe size of multi = %u",
         sizeof(multi));
  printf("\nThe size of multi[0] = %u",
         sizeof(multi[0]));
  printf("\nThe size of multi[0][0] = %u",
```

```
15:           sizeof(multi[0][0]));
16:   return 0;
17: }
```

Output

The output of this program (assuming your compiler uses four-byte integers) is as follows:

```
The size of multi = 32
The size of multi[0] = 16
The size of multi[0][0] = 4
```

Description

Think about these size values. The array `multi` contains two arrays, each of which contains four integers. Each integer requires four bytes of storage. With a total of eight integers, the size of 32 bytes makes sense.

Next, `multi[0]` is an array containing four integers. Each integer takes four bytes, so the size of sixteen bytes for `multi[0]` also makes sense.

Finally, `multi[0][0]` is an integer; so its size is, of course, four bytes.

Pointer Arithmetic with Multidimensional Arrays

Now, keeping these sizes in mind (`multi = 32`, `multi[0] = 16`, and `multi[0][0] = 4`) recall the discussion on Day 9, "Pointers," about pointer arithmetic. The C compiler "knows" the size of the object being pointed to, and pointer arithmetic takes this size into account. When you increment a pointer, its value is increased by the amount needed to make it point to the "next" of whatever it is pointing to. In other words, it is incremented by the size of the object it points to. Click the Example link.

Example

The program in Listing 15.3 tests this theory.

Listing 15.3: Demonstration of Pointer Arithmetic with Multidimensional Arrays

```
Code  1: /* LIST1503.c: Day 15 Listing 15.3 */
      2: /* Demonstrates pointer arithmetic with pointers */
      3: /* to multidimensional arrays. */
      4:
      5: #include <stdio.h>
      6:
      7: int multi[2][4];
      8:
      9: int main(void) {
     10:   printf("\nThe value of (multi) = %u", multi);
     11:   printf("\nThe value of (multi + 1) = %u",
     12:          (multi + 1));
     13:   printf("\nThe address of multi[1] = %u",
     14:           &multi[1]);
     15:   return 0;
     16: }
```

Output

```
The value of (multi) = 1376
The value of (multi + 1) = 1392
The address of multi[1] = 1392
```

Description

The precise values might be different on your system, but the relationships are the same. Incrementing `multi` by 1 increases its value by 16 and makes it point to the next element of the array, `multi[1]`.

When we apply this to the example, `multi` is a pointer to a four-element integer array with a size of 16. If you increment `multi`, its value should increase by 16 (the size of a four-element integer array). If `multi` points to `multi[0]`, therefore, `multi + 1` should point to `multi[1]`.

Summary
You've seen that `multi` is a pointer to `multi[0]`. You've also seen that `multi[0]` is itself a pointer (to `multi[0][0]`). Therefore, `multi` is a pointer to a pointer. To use the expression `multi` to access array data, you must use double indirection. Click the Example link.

Example

Arrays with Three or More Dimensions
These concepts apply equally to arrays with three or more dimensions. Thus, a three-dimensional array is an array with elements that are each a two-dimensional array; each of these elements is itself an array of one-dimensional arrays.

To print the value stored in `multi[0][0]`, you could use any of the following three statements:

```
printf("%d", multi[0][0]);
printf("%d", *multi[0]);
printf("%d", **multi);
```

Declaring Pointers to Elements of a Multidimensional Array
So far, you have been using array names that are pointer constants and cannot be changed. How would you declare a pointer variable that points to an element of a multidimensional array? Click the Example link and then the Tip button.

Example

This material on multidimensional arrays and pointers might seem a bit confusing. When you work with multidimensional arrays, keep this point in mind: an array with *n* dimensions has elements that are arrays with *n-1* dimensions. When *n* becomes 1, that array's elements are variables of the data type specified at the beginning of the array declaration line.

Continuing with the previous example, which has declared a two-dimensional array as

```
int multi[2][4];
```

To declare a pointer variable ptr that can point to an element of `multi` (that is, can point to a four-element integer array), you would write

```
int (*ptr)[4];
```

and then you can make ptr point to the first element of `multi` by writing

```
ptr = multi;
```

Parentheses Required
You might wonder why the parentheses are necessary in the pointer declaration `int (*ptr)[4];`. Brackets [ ] have a higher precedence than *. If you wrote

```
int *ptr[4];
```

you would be declaring an array of four pointers to type int. Indeed, you can declare and use arrays of pointers. This is not what you want to do now, however.

Using Pointers to Elements of Multidimensional Arrays
How can you use pointers to elements of multidimensional arrays? As with single-dimensional arrays, pointers must be used to pass an array to a function. This is illustrated for a multidimensional array in Listing 15.4, which uses two

methods of passing a multidimensional array to a function.

Listing 15.4. Passing a Multidimensional Array to a Function Using a Pointer

Code

```
 1: /* LIST1504.c: Day 15 Listing 15.4 */
 2: /* Demonstrates passing a pointer to a */
 3: /* multidimensional array to a function. */
 4:
 5: #include <stdio.h>
 6:
 7: void printarray_1(int (*ptr)[4]);
 8: void printarray_2(int (*ptr)[4], int n);
 9:
10: int main(void) {
11:   int multi[3][4] = {{1,2,3,4},
12:                      {5,6,7,8},
13:                      {9,10,11,12}};
14:   /* ptr is a pointer to an array of 4 ints. */
15:
16:   int (*ptr)[4], count;
17:
18:   /* Set ptr to point to the first element of */
19:   /* multi. */
20:
21:   ptr = multi;
22:
23:   /* With each loop, ptr is incremented to point */
24:   /* at the next element (that is, the next 4- */
25:   /* element integer array) of multi. */
26:
27:   for (count = 0; count < 3; count++)
28:     printarray_1(ptr++);
29:
30:   puts("\n\nPress ENTER...");
31:   getchar();
32:
33:   printarray_2(multi, 3);
34:   return 0;
35: }
36:
37: void printarray_1(int (*ptr)[4]) {
38:   /* Prints the elements of a single 4-element */
39:   /* integer array. p is a pointer to type int. */
40:   /* You must use a type cast to make p equal */
41:   /* to the address in ptr. */
42:
43:   int *p, count;
44:   p = (int *)ptr;
45:
46:   for (count = 0; count < 4; count++)
47:     printf("\n%d", *p++);
48: }
49:
50: void printarray_2(int (*ptr)[4], int n) {
51:   /* Prints the elements of an n by 4-element */
52:   /* integer array. */
53:
54:   int *p, count;
55:   p = (int *)ptr;
56:
57:   for (count = 0; count < (4 * n); count++)
58:     printf("\n%d", *p++);
59: }
```

Output

```
1
2
3
4
5
6
7

8
```

```
8
9
10
11
12

Press a key...

1
2
3
4
5
6
7
8
9
10
11
12
```

Description

The program declares and initializes an array of integers, `multi[3][4]` on lines 11 – 13. It has two functions, printarray_1() and printarray_2(), that print the contents of the array.

The function printarray_1() (lines 37 – 48) is passed only one argument, a pointer to an array of 4 integers. The function prints all four elements of the array. The first time main() calls printarray_1() on line 28, it passes a pointer to the first element (the first four-element integer array) in `multi`. It then calls the function two more times, incrementing the pointer each time to point to the second, and then the third, element of `multi`. After all three calls are made, the 12 integers in `multi` are displayed.

The second function, printarray_2(), takes a different approach. It too is passed a pointer to an array of 4 integers, but it also is passed an integer variable giving the number of elements (the number of arrays of 4 integers) that the multidimensional array contains. With a single call from line 33, printarray_2() displays the entire contents of `multi`.

Both functions use pointer notation to step through the individual integers in the array. The notation `(int *)ptr` in both functions (lines 44 and 55) might not be clear. The `(int *)` is a type cast, which temporarily changes the variable's data type from its declared data type to a new one. The type cast is required when assigning the value of ptr to p because they are pointers to different types (p is a pointer to type int, whereas ptr is a pointer to an array of four integers). C doesn't allow you to assign the value of one pointer to a pointer of a different type. The type cast tells the compiler "for this statement only, treat ptr as a pointer to type int." Day 20, "Odds and Ends," covers type casts in more detail.

Topic 1.3: Arrays of Pointers

Arrays of Pointers

Recall from Day 8, "Numeric Arrays," that an array is a collection of data storage locations that have the same data type and are referred to by the same name. Because pointers are one of C's data types, you can declare and use arrays of pointers. This type of program construct can be very powerful in certain situations.

Usage

Perhaps the most common use for an array of pointers is with strings. A string, as you learned on Day 10, "Characters and Strings," is a sequence of characters stored in memory. The start of the string is indicated by a pointer to the first character (a pointer to type char); the end of the string is marked by a null character. By declaring and initializing an array of pointers to type char, you can access and manipulate a large number of strings using the pointer array.

Topic 1.3.1: Strings and Pointers: A Review

Topic 1.3.1: Strings and Pointers: A Review

Declaration with Allocation and Initialization
This is a good time to review some material from Day 10, "Characters and Strings," regarding string allocation and initialization. If you want to allocate and initialize a string, you declare an array of type char as follows:

```
char message[] = "This is the message.";
```

You could also declare a pointer to type char.

```
char *message = "This is the message.";
```

Both declarations are equivalent. In each case, the compiler allocates enough space to hold the string with its terminating null character, and the expression `message` is a pointer to the start of the string.

Using the malloc() Function
What about these two declarations?

```
char message1[20];
char *message2;
```

The first line declares an array of type char that is 20 characters long, and `message1` is a pointer to the first array position. Although the array space is allocated, it has not been initialized. The second line declares `message2`, a pointer to type char. No storage space for a string is allocated by this statement. If you want to create a string and then point `message2` at it, you must allocate space for the string first. On Day 10, "Characters and Strings," you learned how to use the malloc() memory allocation function for this purpose. Remember that any string must have space allocated for it, whether at compilation in a declaration or at runtime with malloc().

Topic 1.3.2: Array of Pointers to char

Declaring an Array of Pointers
Now that you're done with the review, how do you declare an array of pointers? The following statement declares an array of 10 pointers to type char:

```
char *message[10];
```

Each element of the array `message[]` is an individual pointer to type char.

Declaration with Allocation and Initialization
As you might have guessed, you can combine the declaration with initialization and allocation of storage space for the strings. Click the Example link.

Example

Relationship
The declaration `char *message[10] = { "one", "two", "three" };` shows the relationship between the array of pointers and the strings. Note that in this example, array elements message[3] through message[9] are not initialized to point at anything.

Now click the button to view Listing 15.5, which is an example of using an array of pointers.

Listing 15.5: Initializing and Using an Array of Pointers to Type char

```
Code  1: /* LIST1505.c: Day 15 Listing 15.5 */
      2: /* Initializing an array of pointers to */
      3: /* type char. */
      4:
      5: #include <stdio.h>
      6:
```

```
 7: int main(void)
 8: {
 9:     char *message[8] = {"Four", "score", "and",
10:     "seven", "years", "ago,", "our", "forefathers" };
11:     int count;
12:
13:     for (count = 0; count < 8; count++)
14:         printf("%s ", message[count]);
15:     return 0;
16: }
```

Output

```
Four score and seven years ago, our forefathers
```

Description The program in Listing 15.5 declares an array of 8 pointers to type char and initializes them to point to 8 strings (lines 9 – 10). It then uses a for loop on lines 13 and 14 to display each element of the array on the screen.

```
char *message[10] = { "one", "two", "three" };
```

This declaration does the following:

* It allocates a 10-element array named message; each element of message is a pointer to type char.
* It allocates space somewhere in memory (where exactly doesn't concern you) and stores the three initialization strings, each with a terminating null character.
* It initializes message[0] to point at the first character of the string one, message[1] to point at the first character of the string two, and message[2] to point at the first character of the string three.

Passing an Array of Pointers to a Function

You probably can see how manipulating the array of pointers is easier than manipulating the strings themselves. This advantage is obvious in more complicated programs, such as the one presented later in this unit. As you will see in that program, the advantage is greatest when you are using functions. It's much easier to pass an array of pointers to a function than several strings.

Passing an array of pointers to a function can be illustrated by rewriting the program in Listing 15.5 so that it uses a function to display the strings. This modified program is given in Listing 15.6.

Listing 15.6: Passing an Array of Pointers to a Function

Code

```
 1: /* LIST1506.c: Day 15 Listing 15.6 */
 2: /* Passing an array of pointers to a function. */
 3:
 4: #include <stdio.h>
 5:
 6: void print_strings(char *p[], int n);
 7:
 8: int main(void) {
 9:   char *message[8] = {"Four", "score", "and",
10:   "seven", "years", "ago,", "our", "forefathers" };
11:   print_strings(message, 8);
12:   return 0;
13: }
14:
15: void print_strings(char *p[], int n) {
16:   int count;
17:
18:   for (count = 0; count < n; count++)
19:     printf("%s ", p[count]);
20: }
```

Output

```
Four score and seven years ago, our forefathers
```

Output	
Description	Looking at line 15, you see that the function print_strings() takes two arguments. One is an array of pointers to type char, the other is the number of elements in the array. Thus, print_strings() could be used to print the strings pointed to by any array of pointers. You may remember that in the section on pointers-to-pointers, you were informed that you would see a demonstration later. Well, you've just seen it. Listing 15.6 declared an array of pointers, and the name of the array is a pointer to its first element. When you pass that array to a function, you are passing a pointer (the array name) to a pointer (the first array element).

Topic 1.3.3: An Example

Example Program

Now it's time for a more complicated example. The program in Listing 15.7 uses many of the programming skills that you have learned, including arrays of pointers. The program accepts lines of input from the keyboard, allocating space for each line as it is entered and keeping track of the lines by means of an array of pointers to type char. When you signal the end of an entry by entering a blank line, the program sorts the strings alphabetically and displays them on-screen.

Listing 15.7: Reading Lines of Text from the Keyboard, Sorting and Displaying

```
Code  1: /* LIST1507.c: Day 15 Listing 15.7 */
      2: /* Inputs a list of strings from the keyboard, */
      3: /* sorts them, then displays them on the screen. */
      4:
      5: #include <stdio.h>
      6: #include <string.h>
      7: #include <stdlib.h>
      8:
      9: #define MAXLINES 25
     10:
     11: int get_lines(char *lines[]);
     12: void sort(char *p[], int n);
     13: void print_strings(char *p[], int n);
     14:
     15: char *lines[MAXLINES];
     16:
     17: int main(void) {
     18:   int number_of_lines;
     19:
     20:   /* Read in the lines from the keyboard. */
     21:
     22:   number_of_lines = get_lines(lines);
     23:
     24:   if (number_of_lines < 0) {
     25:     puts("Memory allocation error");
     26:     exit(-1);
     27:   }
     28:
     29:   sort(lines, number_of_lines);
     30:   print_strings(lines, number_of_lines);
     31:   return 0;
     32: }
     33:
     34: int get_lines(char *lines[]) {
     35:   int n = 0;
     36:   char buffer[80];
     37:   /* Temporary storage for each line. */
     38:
     39:   puts("Enter one line at a time; "
     40:        "enter a blank when done.");
     41:
     42:   while ((n < MAXLINES) && (gets(buffer) != 0) &&
     43:          (buffer[0] != '\0')) {
```

```
44:     if ((lines[n] =
45:        (char *)malloc(strlen(buffer)+1)) == NULL)
46:       return -1;
47:     strcpy(lines[n++], buffer);
48:   }
49:   return n;
50: }
51:
52: void sort(char *p[], int n) {
53:   int a, b;
54:   char *x;
55:
56:   for (a = 1; a < n; a++) {
57:     for (b = 0; b < n-1; b++) {
58:       if (strcmp(p[b], p[b+1]) > 0) {
59:         x = p[b];
60:         p[b] = p[b+1];
61:         p[b+1] = x;
62:       }
63:     }
64:   }
65: }
66:
67: void print_strings(char *p[], int n) {
68:   int count;
69:
70:   for (count = 0; count < n; count++)
71:     printf("\n%s", p[count]);
72: }
```

Output

```
Enter one line at time; enter a blank when done.
dog
apple
zoo
program
merry



apple
dog
merry
program
zoo
```

Description

Examine some of the details of the program. Several new library functions are used for various types of string manipulation. They are explained briefly here, and in more detail on Day 17, "Manipulating Strings." The header file STRING.H must be included in a program that uses these functions.

In the get_lines() function, input is controlled by the while statement on lines 42 – 43, which read as follows:

```
while ((n < MAXLINES) && (gets(buffer) != 0) &&
     (buffer[0] != '\0'))
```

The condition tested by the while has three parts. The first part, `n < MAXLINES`, ensures that the maximum number of lines has not been input already. The second part, `gets(buffer) != 0`, calls the gets() library function to read a line from the keyboard into buffer, and verifies that end-of-file or some other error has not occurred. The third part, `buffer[0] != '\0'`, verifies that the first character of the line just input is not the null character, which would signal that a blank line was entered.

If any of the three conditions is not satisfied, the while loop terminates, and execution returns to the calling program, with the number of lines entered as the return value. If all three conditions are satisfied, the following if statement on line 44 is executed:

```
if ((lines[n] =
   (char *)malloc(strlen(buffer)+1)) == NULL)
```

The first part of the condition calls malloc() to allocate space for the string just input. The strlen() function returns the length of the string passed as an argument; the value is incremented by 1 so that malloc() allocates space for the string plus its terminating null character.

The library function malloc(), you might remember, returns a pointer. The statement assigns the value of the pointer returned by malloc() to the corresponding element of the array of pointers. If malloc() returns NULL, the if loop returns execution to the calling program with a return value of –1. The code in main() tests the return value of get_lines() and whether a value less than 0 is returned; lines 24 – 27 report a memory allocation error and terminate the program.

If the memory allocation was successful, the program uses the strcpy() function on line 47 to copy the string from the temporary storage location buffer to the storage space just allocated by malloc(). The while loop then repeats, getting another line of input.

Once execution returns to main() from get_lines(), the following has been accomplished (assuming a memory allocation error did not occur):

A number of lines of text have been read from the keyboard and stored in memory as null-terminated strings.

The array `lines[]` contains a pointer to each string. The order of pointers in the array is the order in which the strings were input.

The variable number_of_lines holds the number of lines that were input.

Now it's time to sort. Remember, you're not actually going to move the strings around, only the order of the pointers in the array `lines[]`. Look at the code in the function sort(). It contains one for loop nested in another (lines 56 – 64). The outer loop executes number_of_lines - 1 times. Each time the outer loop executes, the inner loop steps through the array of pointers, comparing (string n) with (string n+1) for n = 0 to n = number_of_lines - 1. The comparison is performed by line 58's library function strcmp(), which receives pointers to two strings. The function strcmp() returns one of the following:

A value greater than 0 if the first string is greater than the second string.

Zero if the two strings are identical.

A value less than zero if the second string is greater than the first string.

In the program, a return value from strcmp() of greater than 0 means the first string is "greater than" the second string, and they must be swapped (that is, their pointers in lines[ ] must be swapped). This is done with the help of a temporary variable x. Lines 59, 60, and 61 perform the swap.

When program execution returns from sort(), the pointers in `lines[]` are ordered properly: a pointer to the "lowest" string is in `lines[0]`, a pointer to the next "lowest" is in `lines[1]`, and so on. Say, for example, you entered the following five lines, in this order:

```
dog
apple
zoo
program
merry
```

Finally, the program calls the function print_strings() to display the sorted list of strings on

the screen. This function should be familiar from previous examples in this unit.

	the screen. This function should be familiar from previous examples in this unit.

List What the Program Must Do

You should approach the design of this program from a structured programming perspective. First, make a list of the things the program must do:

Step	Action
1.	Accept lines of input from the keyboard one at a time until a blank line is entered.
2.	Sort the lines into alphabetical order.
3.	Display the sorted lines on the screen.

Three Functions

The list suggests that the program should have at least three functions: one to accept input, one to sort the lines, and one to display the lines. Now you can design each function independently.

The First Function: Accepting Input

What do you need the input function— called get_lines()—to do? Again, make a list:

Step	Action
1.	Keep track of the number of lines entered and return that value to the calling program once all lines are entered.
2.	Do not allow input of more than a preset maximum number of lines.
3.	Allocate storage space for each line.
4.	Keep track of all lines by storing pointers to strings in an array.
5.	Return to the calling program when a blank line is entered.

The Second Function: Sorting the Lines

The Sort Technique

Now think about the second function, the one that sorts the lines. It could be called sort(). (Really original, right?) The sort technique used is a simple, brute force method that compares adjacent strings and swaps them if the second string is less than the first string. More exactly, the function compares the two strings whose pointers are adjacent in the array of pointers and swaps the two pointers if necessary.

The Second Function: Sorting the Lines

Go Through the Array n-1 Times

To be sure that the sorting is complete, you must go through the array from start to finish, comparing each pair of strings and swapping if necessary. For an array of n elements, you must go through the array n-1 times. Why is this necessary?

Each time you go through the array, a given element can be shifted by at most one position. For example, if the string that should be first is actually in the last position, the first pass through the array moves it to the next-to-last position, the second pass through the array moves it up one more position, and so on. It requires n-1 passes to move it to the first position, where it belongs.

The Second Function: Sorting the Lines

Inefficient and Inelegant Sorting Method

Please note that this is a very inefficient and inelegant sorting method. It is, however, easy to implement and understand and adequate for the short lists that the example program sorts.

The Third Function: Displaying the Lines

The final function displays the sorted lines on the screen. It was, in effect, already written in Listing 15.6, requiring

The final function displays the sorted lines on the screen. It was, in effect, already written in Listing 15.6, requiring only minor modification.

Listing 15.6: Passing an Array of Pointers to a Function

```
Code  1: /* LIST1506.c: Day 15 Listing 15.6 */
      2: /* Passing an array of pointers to a function. */
      3:
      4: #include <stdio.h>
      5:
      6: void print_strings(char *p[], int n);
      7:
      8: int main(void) {
      9:   char *message[8] = {"Four", "score", "and",
     10:   "seven", "years", "ago,", "our", "forefathers" };
     11:   print_strings(message, 8);
     12:   return 0;
     13: }
     14:
     15: void print_strings(char *p[], int n) {
     16:   int count;
     17:
     18:   for (count = 0; count < n; count++)
     19:     printf("%s ", p[count]);
     20: }
```

Output

```
Four score and seven years ago, our forefathers
```

Description

Looking at line 15, you see that the function print_strings() takes two arguments. One is an array of pointers to type char, the other is the number of elements in the array. Thus, print_strings() could be used to print the strings pointed to by any array of pointers.

You may remember that in the section on pointers-to-pointers, you were informed that you would see a demonstration later. Well, you've just seen it. Listing 15.6 declared an array of pointers, and the name of the array is a pointer to its first element. When you pass that array to a function, you are passing a pointer (the array name) to a pointer (the first array element).

Example Program

The program in Listing 15.7 is the most complex you have encountered in this course. It uses many of the C programming techniques that have been covered in previous units. With the aid of the preceding explanation, you should be able to follow the operation of the program and understand each step. If you find areas that are not clear to you, review those related sections of the course until you do understand, before going on to the next section. 7.

Listing 15.7: Reading Lines of Text from the Keyboard, Sorting and Displaying

```
Code  1: /* LIST1507.c: Day 15 Listing 15.7 */
      2: /* Inputs a list of strings from the keyboard, */
      3: /* sorts them, then displays them on the screen. */
      4:
      5: #include <stdio.h>
      6: #include <string.h>
      7: #include <stdlib.h>
      8:
      9: #define MAXLINES 25
     10:
     11: int get_lines(char *lines[]);
     12: void sort(char *p[], int n);
     13: void print_strings(char *p[], int n);
     14:
     15: char *lines[MAXLINES];
     16:
     17: int main(void) {
     18:   int number_of_lines;
     19:
     20:   /* Read in the lines from the keyboard. */
     21:

     22:   number of lines = get lines(lines);
```

```
23:
24:   if (number_of_lines < 0) {
25:     puts("Memory allocation error");
26:     exit(-1);
27:   }
28:
29:   sort(lines, number_of_lines);
30:   print_strings(lines, number_of_lines);
31:   return 0;
32: }
33:
34: int get_lines(char *lines[]) {
35:   int n = 0;
36:   char buffer[80];
37:   /* Temporary storage for each line. */
38:
39:   puts("Type a single string, then press enter."
40:        "Enter the next string on the next line. Enter a blank line when done.");
41:
42:   while ((n < MAXLINES) && (gets(buffer) != 0) &&
43:        (buffer[0] != '\0')) {
44:     if ((lines[n] =
45:        (char *)malloc(strlen(buffer)+1)) == NULL)
46:       return -1;
47:     strcpy(lines[n++], buffer);
48:   }
49:   return n;
50: }
51:
52: void sort(char *p[], int n) {
53:   int a, b;
54:   char *x;
55:
56:   for (a = 1; a < n; a++) {
57:     for (b = 0; b < n-1; b++) {
58:       if (strcmp(p[b], p[b+1]) > 0) {
59:         x = p[b];
60:         p[b] = p[b+1];
61:         p[b+1] = x;
62:       }
63:     }
64:   }
65: }
66:
67: void print_strings(char *p[], int n) {
68:   int count;
69:
70:   for (count = 0; count < n; count++)
71:     printf("\n%s", p[count]);
72: }
```

Output

```
Enter one line at time; enter a blank when done.
dog
apple
zoo
program
merry


apple
dog
merry
program
zoo
```

Description

Examine some of the details of the program. Several new library functions are used for various types of string manipulation. They are explained briefly here, and in more detail on Day 17, "Manipulating Strings." The header file STRING.H must be included in a program that uses these functions.

In the get_lines() function, input is controlled by the while statement on lines 42 – 43, which read as

In the get_lines() function, input is controlled by the while statement on lines 42 – 43, which could be follows:

```
while ((n < MAXLINES) && (gets(buffer) != 0) &&
     (buffer[0] != '\0'))
```

The condition tested by the while has three parts. The first part, `n < MAXLINES`, ensures that the maximum number of lines has not been input already. The second part, `gets(buffer) != 0`, calls the gets() library function to read a line from the keyboard into buffer, and verifies that end-of-file or some other error has not occurred. The third part, `buffer[0] != '\0'`, verifies that the first character of the line just input is not the null character, which would signal that a blank line was entered.

If any of the three conditions is not satisfied, the while loop terminates, and execution returns to the calling program, with the number of lines entered as the return value. If all three conditions are satisfied, the following if statement on line 44 is executed:

```
if ((lines[n] =
   (char *)malloc(strlen(buffer)+1)) == NULL)
```

The first part of the condition calls malloc() to allocate space for the string just input. The strlen() function returns the length of the string passed as an argument; the value is incremented by 1 so that malloc() allocates space for the string plus its terminating null character.

The library function malloc(), you might remember, returns a pointer. The statement assigns the value of the pointer returned by malloc() to the corresponding element of the array of pointers. If malloc() returns NULL, the if loop returns execution to the calling program with a return value of –1. The code in main() tests the return value of get_lines() and whether a value less than 0 is returned; lines 24 – 27 report a memory allocation error and terminate the program.

If the memory allocation was successful, the program uses the strcpy() function on line 47 to copy the string from the temporary storage location buffer to the storage space just allocated by malloc(). The while loop then repeats, getting another line of input.

Once execution returns to main() from get_lines(), the following has been accomplished (assuming a memory allocation error did not occur):

A number of lines of text have been read from the keyboard and stored in memory as null-terminated strings.

The array `lines[]` contains a pointer to each string. The order of pointers in the array is the order in which the strings were input.

The variable number_of_lines holds the number of lines that were input.

Now it's time to sort. Remember, you're not actually going to move the strings around, only the order of the pointers in the array `lines[]`. Look at the code in the function sort(). It contains one for loop nested in another (lines 56 – 64). The outer loop executes number_of_lines - 1 times. Each time the outer loop executes, the inner loop steps through the array of pointers, comparing (string n) with (string n+1) for n = 0 to n = number_of_lines - 1. The comparison is performed by line 58's library function strcmp(), which receives pointers to two strings. The function strcmp() returns one of the following:

A value greater than 0 if the first string is greater than the second string.

Zero if the two strings are identical.

A value less than zero if the second string is greater than the first string.

In the program, a return value from strcmp() of greater than 0 means the first string is "greater than" the second string, and they must be swapped (that is, their pointers in lines[ ] must be swapped). This is done with the help of a temporary variable x. Lines 59, 60, and 61 perform the swap.

When program execution returns from sort(), the pointers in `lines[]` are ordered properly: a pointer to the "lowest" string is in `lines[0]`, a pointer to the next "lowest" is in `lines[1]`, and so on. Say, for example, you entered the following five lines, in this order:

```
dog
apple
zoo
program
merry
```

Finally, the program calls the function print_strings() to display the sorted list of strings on the screen. This function should be familiar from previous examples in this unit.

Topic 1.4: Pointers to Functions

Pointers to Functions
Pointers to functions provide another way of calling functions. "Hold on," you might be saying, "how can you have a pointer to a function? A pointer holds the address where a variable is stored, doesn't it?"

The Pointer Holds the Starting Address of a Function
Well, yes and no. It's true that a pointer holds an address, but it doesn't have to be the address where a variable is stored. When your program runs, the code for each function is loaded into memory starting at a specific address. A pointer to a function holds the starting address of a function, its entry point.

Flexible Way of Calling a Function
Why use a pointer to a function? As mentioned earlier, it provides a more flexible way of calling a function. It allows the program to "pick" among several functions, selecting the one that is appropriate for the current circumstances.

Topic 1.4.1: Declaring a Pointer to a Function

Syntax
Like other pointers, you must declare a pointer to a function. The general form of the declaration is

```
type (*ptr_to_func)(parameter_list);
```

This statement declares ptr_to_func as a pointer to a function that returns type and is passed the parameters in parameter_list. Click the Examples link.

Examples

Here are some more concrete examples:

Line	Description
`int (*func1)(int x);`	Declares func1 as a pointer to a function that takes one type int argument and returns a type int.
`void (*func2)(double y, double z);`	Declares func2 as a pointer to a function that takes two type double arguments and has a void return type (no return value).
`char (*func3)(char *p[]);`	Declares func3 as a pointer to a function that takes an array of pointers to type char as its argument and returns a type char.
`void (*func4)();`	Declares func4 as a pointer to a function that does not take any arguments and

`void (*func4)();`	Declares func4 as a pointer to a function that does not take any arguments and has a void return type.

Parentheses Required
Why do you need the parentheses around the pointer name? Why can't you write, for the first example,

```
int *func1(int x);
```

The reason has to do with the precedence of the indirection operator, *. It has a relatively low precedence, lower than the parentheses surrounding the parameter list. The declaration just given, without the first set of parentheses, declares func1 as a function that returns a pointer to type int. (Functions that return pointers are covered on Day 18, "Getting More from Functions.") When declaring a pointer to a function, always remember to include a set of parentheses around the pointer name and indirection operator, or you will get into trouble for sure.

Topic 1.4.2: Initializing and Using a Pointer to a Function

A Pointer to a Function
A pointer to a function must not only be declared, but initialized to point to something. That "something" is, of course, a function. There's nothing special about a function that gets pointed to. The only requirement is that its return type and parameter list match the return type and parameter list of the pointer declaration. Click the Example link, then

Example

DO use structured programming.

DON'T forget to use parentheses when declaring pointers to functions.

Declaring a pointer to a function that takes no arguments and returns a character looks like this:

```
char (*func)();
```

Declaring a function that returns a pointer to a character looks like this:

```
char *func();
```

DO initialize a pointer before using it.

DON'T use a function pointer that has been declared with a different return type or different arguments than what you need.

Listing 15.8: Using a Pointer to a Function to Call the Function

```
Code  1: /* LIST1508.c: Day 15 Listing 15.8 */
      2: /* Demonstration of declaring and  */
      3: /* using a pointer to a function. */
      4:
      5: #include <stdio.h>
      6:
      7: /* The function prototype. */
      8:
      9: float square(float x);
     10:
     11: /* The pointer declaration. */
     12:
     13: float (*p)(float x);

     14:
```

```
15: int main(void) {
16:   /* Initialize p to point to square(). */
17:
18:   p = square;
19:
20:   /* Call square() two ways. */
21:
22:   printf("%f %f", square(6.6), p(6.6));
23:   return 0;
24: }
25:
26: float square(float x) {
27:   return x * x;
28: }
```

Output

```
43.559999  43.559999
```

Note: Precision of the values may cause some numbers to not display as the exact values entered. For example, `45.56` may appear as `45.559999`.

Description Line 9 declares square(), and line 13 declares the pointer, p, to a function containing a float argument and returning a float value, matching the declaration of square(). Line 18 sets the pointer, p, equal to square. Notice that parentheses are not used with square or p. Line 22 prints the return values from calls to square() and p().

The following code declares and defines a function and a pointer to that function.

```
* The function prototype. */
float square(float x);

/* The pointer declaration. */
float (*p)(float x)

/* The function definition. */
float square(float x)
{
    return x * x;
}
```

Because the function square() and the pointer p have the same parameter and return types, you can initialize p to point to square as follows: `p = square;`. Then you can call the function using the pointer as follows: `answer = p(x);`

It's that simple. For a real example, compile and run the program in Listing 15.8, which declares and initializes a pointer to a function, and then calls the function twice, using the function name the first time and the pointer the second time. Both calls produce the same result.

Using a Pointer Constant Versus a Pointer Variable

A function name without the parentheses is a pointer to the function (sounds similar to the situation with arrays, doesn't it?). What's the point of declaring and using a separate pointer to the function? Well, the function name itself is a pointer constant and cannot be changed (again, a parallel to arrays). A pointer variable, in contrast, can be changed. Specifically, it can be made to point to different functions as the need arises.

Example Program

The program in Listing 15.9 calls a function, passing it an integer argument. Depending on the value of the argument, the function initializes a pointer to point to one of three other functions, and then uses the pointer to call the corresponding function. Each of these three functions displays a specific message on the screen.

Of course, the program in Listing 15.9 is for illustration purposes only. You easily could have accomplished the same result without using a pointer to a function.

Listing 15.9: Using a Pointer to a Function to Call Different Functions Depending on Program

Circumstances

Code

```
 1: /* LIST1509.c: Day 15 Listing 15.9 */
 2: /* Using a pointer to call different functions. */
 3:
 4: #include <stdio.h>
 5:
 6: /* The function prototypes. */
 7:
 8: void func1(int x);
 9: void one(void);
10: void two(void);
11: void other(void);
12:
13: int main(void) {
14:   int a;
15:
16:   for (;;) {
17:     puts("\nEnter an integer between 1 and 10,"
18:          " 0 to exit: ");
19:     scanf("%d", &a);
20:
21:     if (a == 0)
22:       break;
23:
24:     func1(a);
25:   }
26:   return 0;
27: }
28:
29: void func1(int x) {
30:   /* The pointer to function. */
31:
32:   void (*ptr)(void);
33:
34:   if (x == 1)
35:     ptr = one;
36:   else if (x == 2)
37:     ptr = two;
38:   else
39:     ptr = other;
40:
41:   ptr();
42: }
43:
44: void one(void) {
45:   puts("You entered 1.");
46: }
47:
48: void two(void) {
49:   puts("You entered 2.");
50: }
51:
52: void other(void) {
53:   puts("You entered something other than 1 or 2.");
54: }
```

Output

```
Enter an integer between 1 and 10, 0 to exit:
2
You entered 2.

Enter an integer between 1 and 10, 0 to exit:
11
You entered something other than 1 or 2.
Enter an integer between 1 and 10, 0 to exit:
0
```

Description

This program employs an infinite loop on line 16 to continue the program until a value of 0 is entered. When a value is entered, if it isn't 0, it's passed to func1(). Note that line 32, in func1(), contains a declaration for a pointer to a function (ptr). This makes it local to func1(), appropriate because no other part of the program needs access to it. func1() then uses this

value to set ptr equal to the appropriate function (lines 34 – 39). Line 41 then makes a single call to ptr(), which calls the appropriate function.

Passing a Pointer to a Function as an Argument

Now you can learn another way to use pointers to call different functions: passing the pointer as an argument to a function. The program in Listing 15.10 is a revision of Listing 15.9.

Listing 15.10: Passing a Pointer to a Function as an Argument

Code

```
 1: /* LIST1510.c: Day 15 Listing 15.10 */
 2: /* Passing a pointer to a function as an */
 3: /* argument. */
 4:
 5: #include <stdio.h>
 6:
 7: /* The function prototypes. The function func1() */
 8: /* takes as its one argument a pointer to a */
 9: /* function that takes no arguments and has no */
10: /* return value. */
11:
12: void func1(void (*p)(void));
13: void one(void);
14: void two(void);
15: void other(void);
16:
17: int main(void) {
18:   /* The pointer to a function. */
19:   void (*ptr)(void);
20:   int a;
21:
22:   for (;;) {
23:     puts("\nEnter an integer between 1 and 10,"
24:          " 0 to exit: ");
25:     scanf("%d", &a);
26:
27:     if (a == 0)
28:       break;
29:     else if (a==1)
30:       ptr = one;
31:     else if (a==2)
32:       ptr = two;
33:     else
34:       ptr = other;
35:
36:     func1(ptr);
37:   }
38:   return 0;
39: }
40:
41: void func1(void (*p)(void)) {
42:   p();
43: }
44:
45: void one(void) {
46:   puts("You entered 1.");
47: }
48:
49: void two(void) {
50:   puts("You entered 2.");
51: }
52:
53: void other(void) {
54:   puts("You entered something other than 1 or 2.");
55: }
```

Output

```
Enter an integer between 1 and 10, 0 to exit:
2

You entered 2.
```

```
Enter an integer between 1 and 10, 0 to exit:
11
You entered something other than 1 or 2.
Enter an integer between 1 and 10, 0 to exit:
0
```

Description Notice the differences between Listing 15.9 and Listing 15.10. The declaration of the pointer to a function has been moved to line 19 in main(), where it is needed. Code in main() now initializes the pointer to point to the correct function (lines 27 – 34), and then passes the initialized pointer to func1(). This function, func1() really serves no purpose in Listing 15.10; all it does is call the function pointed to by ptr. Again, this program is for illustration. The same principles can be used in real-world programs, such as the example in the next section.

Using Pointers to Functions to Control the Sort Order

One programming situation in which you might use pointers to functions is when sorting is required. At times, you may want different sorting rules used. For example, you might want to sort in alphabetical order one time and in reverse alphabetical order another time. By using pointers to functions, your program can call the correct sorting function. More precisely, it's usually a different comparison function that's called.

Example Program

Look back at the program in Listing 15.7. In the sort() function, the actual sort order is determined by the value returned by the strcmp() library function, which tells the program whether a given string is "less than" another string. What if you wrote two comparison functions, one that sorts alphabetically (saying that A is less than Z, for example), and another that sorts in reverse alphabetical order (saying that Z is less than A). The program can ask the user what order is desired and, by using pointers, the sorting function can call the proper comparison function. Listing 15.11 modifies the program in Listing 15.7 and incorporates this feature. 7 and 15.11.

Listing 15.7: Reading Lines of Text from the Keyboard, Sorting and Displaying

```
Code  1: /* LIST1507.c: Day 15 Listing 15.7 */
      2: /* Inputs a list of strings from the keyboard, */
      3: /* sorts them, then displays them on the screen. */
      4:
      5: #include <stdio.h>
      6: #include <string.h>
      7: #include <stdlib.h>
      8:
      9: #define MAXLINES 25
     10:
     11: int get_lines(char *lines[]);
     12: void sort(char *p[], int n);
     13: void print_strings(char *p[], int n);
     14:
     15: char *lines[MAXLINES];
     16:
     17: int main(void) {
     18:   int number_of_lines;
     19:
     20:   /* Read in the lines from the keyboard. */
     21:
     22:   number_of_lines = get_lines(lines);
     23:
     24:   if (number_of_lines < 0) {
     25:     puts("Memory allocation error");
     26:     exit(-1);
     27:   }
     28:
     29:   sort(lines, number_of_lines);
     30:   print_strings(lines, number_of_lines);
     31:   return 0;
     32: }
     33:
     34: int get_lines(char *lines[]) {
     35:   int n = 0;
     36:   char buffer[80];
     37:   /* Temporary storage for each line. */
```

```
38:
39:   puts("Enter one line at a time; "
40:        "enter a blank when done.");
41:
42:   while ((n < MAXLINES) && (gets(buffer) != 0) &&
43:        (buffer[0] != '\0')) {
44:     if ((lines[n] =
45:        (char *)malloc(strlen(buffer)+1)) == NULL)
46:       return -1;
47:     strcpy(lines[n++], buffer);
48:   }
49:   return n;
50: }
51:
52: void sort(char *p[], int n) {
53:   int a, b;
54:   char *x;
55:
56:   for (a = 1; a < n; a++) {
57:     for (b = 0; b < n-1; b++) {
58:       if (strcmp(p[b], p[b+1]) > 0) {
59:         x = p[b];
60:         p[b] = p[b+1];
61:         p[b+1] = x;
62:       }
63:     }
64:   }
65: }
66:
67: void print_strings(char *p[], int n) {
68:   int count;
69:
70:   for (count = 0; count < n; count++)
71:     printf("\n%s", p[count]);
72: }
```

Output

```
Enter one line at time; enter a blank when done.
dog
apple
zoo
program
merry


apple
dog
merry
program
zoo
```

Description

Examine some of the details of the program. Several new library functions are used for various types of string manipulation. They are explained briefly here, and in more detail on Day 17, "Manipulating Strings." The header file STRING.H must be included in a program that uses these functions.

In the get_lines() function, input is controlled by the while statement on lines 42 – 43, which read as follows:

```
while ((n < MAXLINES) && (gets(buffer) != 0) &&
     (buffer[0] != '\0'))
```

The condition tested by the while has three parts. The first part, `n < MAXLINES`, ensures that the maximum number of lines has not been input already. The second part, `gets(buffer) != 0`, calls the gets() library function to read a line from the keyboard into buffer, and verifies that end-of-file or some other error has not occurred. The third part, `buffer[0] != '\0'`, verifies that the first character of the line just input is not the null character, which would signal that a blank line was entered.

If any of the three conditions is not satisfied, the while loop terminates, and execution returns to the calling program, with the number of lines entered as the return value. If all three conditions are satisfied, the following if statement on line 44 is executed:

```
if ((lines[n] =
   (char *)malloc(strlen(buffer)+1)) == NULL)
```

The first part of the condition calls malloc() to allocate space for the string just input. The strlen() function returns the length of the string passed as an argument; the value is incremented by 1 so that malloc() allocates space for the string plus its terminating null character.

The library function malloc(), you might remember, returns a pointer. The statement assigns the value of the pointer returned by malloc() to the corresponding element of the array of pointers. If malloc() returns NULL, the if loop returns execution to the calling program with a return value of –1. The code in main() tests the return value of get_lines() and whether a value less than 0 is returned; lines 24 – 27 report a memory allocation error and terminate the program.

If the memory allocation was successful, the program uses the strcpy() function on line 47 to copy the string from the temporary storage location buffer to the storage space just allocated by malloc(). The while loop then repeats, getting another line of input.

Once execution returns to main() from get_lines(), the following has been accomplished (assuming a memory allocation error did not occur):

A number of lines of text have been read from the keyboard and stored in memory as null-terminated strings.

The array `lines[]` contains a pointer to each string. The order of pointers in the array is the order in which the strings were input.

The variable number_of_lines holds the number of lines that were input.

Now it's time to sort. Remember, you're not actually going to move the strings around, only the order of the pointers in the array `lines[]`. Look at the code in the function sort(). It contains one for loop nested in another (lines 56 – 64). The outer loop executes number_of_lines - 1 times. Each time the outer loop executes, the inner loop steps through the array of pointers, comparing (string n) with (string n+1) for n = 0 to n = number_of_lines - 1. The comparison is performed by line 58's library function strcmp(), which receives pointers to two strings. The function strcmp() returns one of the following:

A value greater than 0 if the first string is greater than the second string.

Zero if the two strings are identical.

A value less than zero if the second string is greater than the first string.

In the program, a return value from strcmp() of greater than 0 means the first string is "greater than" the second string, and they must be swapped (that is, their pointers in lines[ ] must be swapped). This is done with the help of a temporary variable x. Lines 59, 60, and 61 perform the swap.

When program execution returns from sort(), the pointers in `lines[]` are ordered properly: a pointer to the "lowest" string is in `lines[0]`, a pointer to the next "lowest" is in `lines[1]`, and so on. Say, for example, you entered the following five lines, in this order:

```
dog
apple
```

```
zoo
program
merry
```

Finally, the program calls the function print_strings() to display the sorted list of strings on the screen. This function should be familiar from previous examples in this unit.

Listing 15.11: Using Pointers to Functions to Control Sort Order

```
Code  1: /* LIST1511.c: Day 15 Listing 15.11 */
      2: /* Inputs a list of strings from the keyboard, */
      3: /* sorts them in ascending or descending order, */
      4: /* then displays them on the screen. */
      5:
      6: #include <stdio.h>
      7: #include <string.h>
      8: #include <stdlib.h>
      9:
     10: #define MAXLINES 25
     11:
     12: int get_lines(char *lines[]);
     13: void sort(char *p[], int n, int sort_type);
     14: void print_strings(char *p[], int n);
     15: int alpha(char *p1, char *p2);
     16: int reverse(char *p1, char *p2);
     17:
     18: char *lines[MAXLINES];
     19:
     20: int main(void) {
     21:   int number_of_lines, sort_type;
     22:
     23:   /* Read in the lines from the keyboard. */
     24:
     25:   number_of_lines = get_lines(lines);
     26:
     27:   if (number_of_lines < 0) {
     28:     puts("Memory allocation error");
     29:     exit(-1);
     30:   }
     31:
     32:   puts("Enter 0 for reverse order sort,"
     33:        " 1 for alphabetical: ");
     34:   scanf("%d", &sort_type);
     35:
     36:   sort(lines, number_of_lines, sort_type);
     37:   print_strings(lines, number_of_lines);
     38:   return 0;
     39: }
     40:
     41: int get_lines(char *lines[]) {
     42:   int n = 0;
     43:   char buffer[80];
     44:   /* Temporary storage for each line. */
     45:
     46:   puts("Enter one line at a time; "
     47:        "enter a blank when done.");
     48:
     49:   while ((n < MAXLINES) && (gets(buffer) != 0) &&
     50:        (buffer[0] != '\0')) {
     51:     if ((lines[n] =
     52:        (char *)malloc(strlen(buffer)+1)) == NULL)
     53:       return -1;
     54:     strcpy(lines[n++], buffer);
     55:   }
     56:   return n;
     57: }
     58:
     59: void sort(char *p[], int n, int sort_type) {
```

```
59: void sort(char *p[], int n, int sort_type) {
60:   int a, b;
61:   char *x;
62:
63:   /* The pointer to function. */
64:
65:   int (*compare)(char *s1, char *s2);
66:
67:   /* Initialize the pointer to point at the */
68:   /* proper comparison function depending on */
69:   /* the argument sort_type. */
70:
71:   compare = (sort_type) ? reverse : alpha;
72:
73:   for (a = 1; a < n; a++) {
74:     for (b = 0; b < n - 1; b++) {
75:       if (compare(p[b], p[b + 1]) > 0) {
76:         x = p[b];
77:         p[b] = p[b + 1];
78:         p[b + 1] = x;
79:       }
80:     }
81:   }
82: }
83:
84: void print_strings(char *p[], int n) {
85:   int count;
86:
87:   for (count = 0; count < n; count++)
88:     printf("\n%s", p[count]);
89: }
90:
91: int alpha(char *p1, char *p2) {
92: /* Alphabetical comparison. */
93:   return(strcmp(p2, p1));
94: }
95:
96: int reverse(char *p1, char *p2) {
97: /* Reverse alphabetical comparison. */
98:   return(strcmp(p1, p2));
99: }
```

Output

```
Enter one line at time; enter a blank when done.
Roses are red
Violets are blue
C has been around,
But it is new to you!
Enter 0 for reverse order sort, 1 for alphabetical:
0
Violets are blue
Roses are red
C has been around,
But it is new to you!
```

Description

Lines 32 – 34 in main() prompt the user for the desired sort order. The order selected is placed in sort_type. This value is passed to the sort() function along with the other information described for Listing 15.7. The sort function contains a couple of changes. Line 65 declares a pointer to a function called compare() that takes two character pointers (strings) as arguments. Line 71 sets compare() equal to one of the two new functions added to the listing based on the value of sort_type. The two new functions are alpha() and reverse(). alpha() uses the strcmp() library function just as it was used in Listing 15.7; reverse() uses this library function a different way. reverse() switches the parameters passed so that a reverse order sort is done.

Topic 1.5: Day 15 Q&A

Questions & Answers

Here are some questions to help you review what you have learned in this unit.

Question 1
How many levels can I go with pointers-to-pointers?

Answer
You need to check your compiler manuals to determine whether there are any limitations. It is usually impractical to go more than three levels deep with pointers (pointers-to-pointers-to-pointers). Most programs rarely go over two levels.

Question 2
Is there a difference between a pointer to a string and a pointer to an array?

Answer
No. A string can be looked at as an array of characters.

Question 3
Is it necessary to use the concepts presented in this unit to take advantage of C?

Answer
You can use C without ever using any advanced pointer concepts; however, you won't take advantage of the power that C offers. By doing pointer manipulations such as those shown in this unit, you should be able to do virtually any programming task in a quick, efficient manner.

Question 4
Are there other times when function pointers are useful?

Answer
Yes. Pointers to functions also are used with menus. Based on a value returned from a menu, a pointer is set to an appropriate function.

Topic 1.6: Day 15 Think About Its

Think About Its
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

`array` is an array with two elements. Each of those elements is itself an array that contains three elements. Each of these elements is an array that contains four type int variables.

Topic 1.7: Day 15 Try Its

Try Its
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
What do the following declare?

```
a.  char *z[10];
b.  char *y(int field);
c.  char (*x)(int field);
```

Answer

a. z is an array of 10 pointers to characters.
b. y is a function that takes an integer (field) as an argument and returns a pointer to a character.
c. x is a pointer to a function that takes an integer (field) as an argument and returns a character.

*** Exercise 2**
Write a declaration for a pointer to a function that takes an integer as an argument and returns a type float variable.

Answer

```
float (*func)(int field);
```

*** Exercise 3**
Write a declaration for an array of pointers to functions. The functions should all take a character string as a parameter and return an integer. What could such an array be used for?

Answer

An array of function pointers can be used in conjunction with a menuing system. The number selected from a menu could relate to the array index for the function pointer. For example, the function pointed to by the fifth element of the array would be executed if item 5 were selected from the menu.

```
int (*menu_option[10])(char *title);
```

*** Exercise 4**
Write a statement to declare an array of ten pointers to type char.

Answer

```
char *ptrs[10];
```

*** Exercise 5**
BUG BUSTERS: Is anything wrong with the following code?

```
int x[3][12];
int *ptr[12];
ptr = x;
```

Answer

ptr was declared as an array of 12 pointers to integers, not a pointer to an array of 12 integers. The corrected code would be

```
int x[3][12];
```

```
int x[3][12];
int (*ptr)[12];

ptr = x;
```

*** Exercise 6**

Write a program that declares a 12-by-12 array of characters. Place X's in every other element. Use a pointer to the array to print the values to the screen in a grid format.

Answer

```
/* EXER1506.C  Day 15 Exercise 6 */
/* Print values of 12x12 array of characters */

#define XLIM  12
#define YLIM  12

char xtable[XLIM][YLIM];
char (*ptr)[YLIM];
char fillchar;
int x;
int y;

void printarray_2(char (*ptr)[YLIM], int n);

int main(void)  {

  fillchar = 'X';
  ptr = xtable;

  for (x = 0; x < XLIM; x++)
    for (y = 0; y < YLIM; y++)  {
      xtable[x][y] = fillchar;
      fillchar = (fillchar == 'X') ? ' ' : 'X';
   }

   printarray_2(xtable,XLIM);

   return 0;
}

void printarray_2(char (*ptr)[YLIM], int n)  {

/* Prints the elements of an n by YLIM-element
 *character array. */
  int count, count2;
  char *p;
  p = (char *) ptr;

  for (count = 0; count < XLIM; count++)  {
    for (count2 = 0; count2 < YLIM; count2++)
      printf("%c ", *p++);
    printf("\n");
  }
}
```

*** Exercise 7**

Write a program that stores ten pointers to double variables. The program should accept the ten numbers from the user, sort them, and then print them to the screen. (Click the button to See Listing 15.10.)

Answer

Answer

Listing 15.10: Passing a Pointer to a Function as an Argument

```
Code  1: /* LIST1510.c: Day 15 Listing 15.10 */
      2: /* Passing a pointer to a function as an */
      3: /* argument. */
      4:
      5: #include <stdio.h>
      6:
      7: /* The function prototypes. The function func1() */
      8: /* takes as its one argument a pointer to a */
      9: /* function that takes no arguments and has no */
     10: /* return value. */
     11:
     12: void func1(void (*p)(void));
     13: void one(void);
     14: void two(void);
     15: void other(void);
     16:
     17: int main(void) {
     18:   /* The pointer to a function. */
     19:   void (*ptr)(void);
     20:   int a;
     21:
     22:   for (;;) {
     23:     puts("\nEnter an integer between 1 and 10,"
     24:          " 0 to exit: ");
     25:     scanf("%d", &a);
     26:
     27:     if (a == 0)
     28:       break;
     29:     else if (a==1)
     30:       ptr = one;
     31:     else if (a==2)
     32:       ptr = two;
     33:     else
     34:       ptr = other;
     35:
     36:     func1(ptr);
     37:   }
     38:   return 0;
     39: }
     40:
     41: void func1(void (*p)(void)) {
     42:   p();
     43: }
     44:
     45: void one(void) {
     46:   puts("You entered 1.");
     47: }
     48:
     49: void two(void) {
     50:   puts("You entered 2.");
     51: }
     52:
     53: void other(void) {
     54:   puts("You entered something other than 1 or 2.");
     55: }
```

```
/* EXER1507.C Day 15 Exercise 7 */
/* Reads numbers from keyboard, sorts them and displays list */

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


#define MAXLINES 10
```

```
int getlines(char *lines[]);
void sort(char *p[], int n);
void printstr(char *p[], int n);
int comp(char buff1[],char buff2[]);

char *lines[MAXLINES];
int numlines;

int main(void)  {

  /* Read in lines from the keyboard */
  numlines = getlines(lines);

  if (numlines < 0)
  {
    puts("Memory allocation error");
    exit(-1);
  }
  sort(lines,numlines);
  printstr(lines,numlines);

  return 0;
}

int getlines(char *lines[])  {
  int n = 0;
  char buffer[80];

  puts("Enter one number at a time; enter a blank when done.");

  while((n < MAXLINES) && (gets(buffer) != 0) && (buffer[0] != '\0'))  {
    if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
      return -1;
    strcpy(lines[n++], buffer);
  }
  return n;
}

void sort(char *p[], int n)  {
  int a, b;
  char *x;

  for (a = 1; a < n; a++)  {
    for (b = 0; b < n - 1; b++)  {
      if (comp(p[b], p[b + 1]) > 0)  {
  x = p[b];
  p[b] = p[b + 1];
  p[b + 1] = x;
      }
    }
  }
}

void printstr(char *p[], int n)  {
  int count;

  for (count = 0; count < n; count++)
    printf("\n%s ", p[count]);
}

/* Compare return positive number if 1st num > 2nd num. */
int comp(char buff1[],char buff2[])  {
  unsigned int num1, num2, x, retcode;

  /* Convert ASCII to integer */
  num1 = (unsigned char)buff1[0] - 0x30;

  for (x = 1; buff1[x] != '\0'; x++)
    num1 = (num1 * 10) + ((unsigned char)buff1[x] - 0x30);
```

```
  num2 = (unsigned char)buff2[0] - 0x30;
  for (x = 1; buff2[x] != '\0'; x++)
    num2 = (num2 * 10) + ((unsigned char)buff2[x] - 0x30);

  if (num1 > num2)
    retcode = 1;
  else
    retcode = 0;

  return retcode;
}
```

*** Exercise 8**

Modify the program in exercise seven to allow the user to determine whether the sort order is low-to-high or high-to-low.

Answer

```
/* EXER158.C Day 15 Exercise 8 */
/* Reads numbers from keyboard, sorts them and displays list */
/* User determines whether sort is in ascending or descending order */

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

#define MAXLINES 10

int getlines(char *lines[]);
void sort(char *p[], int n, char c);
void printstr(char *p[], int n);
int compa(char buff1[],char buff2[]);
int compd(char buff1[],char buff2[]);

char *lines[MAXLINES];
int numlines;
char c;

int main(void)  {
  /* Read in lines from the keyboard. */
  numlines = getlines(lines);

  if (numlines < 0) {
    puts("Memory allocation error");
    exit(-1);
  }

  printf("Enter 1 for ascending sort,");
  printf("anything else for descending sort\n");
  c = getchar();

  sort(lines,numlines,c);
  printstr(lines,numlines);

  return 0;
}

int getlines(char *lines[])  {
  int n = 0;
  char buffer[80];


  puts("Enter one number at a time; enter a blank when done.");
```

```
    puts("Enter one number at a time, enter a blank when done.");

    while((n < MAXLINES) && (gets(buffer) != 0) &&
          (buffer[0] != '\0'))  {
      if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
        return -1;
      strcpy(lines[n++], buffer);
    }
    return n;
  }

  void sort(char *p[], int n, char c)  {
    int a, b;
    char *x;

    /* The pointer to function. */

    int (*ptr)(char p1[], char p2[]);

    if (c == '1')
      ptr = compa;
    else
      ptr = compd;

    for (a = 1; a < n; a++)  {
      for (b = 0; b < n - 1; b++)  {
        if (ptr(p[b], p[b + 1]) > 0)  {
    x = p[b];
    p[b] = p[b + 1];
    p[b + 1] = x;
        }
      }
    }
  }

  void printstr(char *p[], int n)  {
    int count;

    for (count = 0; count < n; count++)
      printf("\n%s ", p[count]);
  }

  /* Compare return positive number if 1st num > 2nd num */
  int compa(char buff1[],char buff2[])  {
    unsigned int num1, num2, x, retcode;

    /* Convert ASCII to integer */
    num1 = (unsigned char)buff1[0] - 0x30;
    for (x = 1; buff1[x] != '\0'; x++)
      num1 = (num1 * 10) + ((unsigned char)buff1[x] - 0x30);

    num2 = (unsigned char)buff2[0] - 0x30;
    for (x = 1; buff2[x] != '\0'; x++)
      num2 = (num2 * 10) + ((unsigned char)buff2[x] - 0x30);

    if (num1 > num2)
      retcode = 1;
    else
      retcode = 0;

    return(retcode);
  }

  /* Compare return positive number if 1st num < 2nd num */
  int compd(char buff1[],char buff2[])  {
    unsigned int num1, num2, x, retcode;

    /* Convert ASCII to integer */
    num1 = (unsigned char)buff1[0] - 0x30;
```

```
  for (x = 1; buff1[x] != '\0'; x++)
    num1 = (num1 * 10) + ((unsigned char)buff1[x] - 0x30);

  num2 = (unsigned char)buff2[0] - 0x30;
  for (x = 1; buff2[x] != '\0'; x++)
    num2 = (num2 * 10) + ((unsigned char)buff2[x] - 0x30);

  if (num1 < num2)
    retcode = 1;
  else
    retcode = 0;

  return(retcode);
}
```

Topic 1.8: Day 15 Summary

Advanced Uses of Pointers
This unit has covered some of the advanced uses of pointers. As you might realize, pointers are central to the C language; C programs that don't use pointers are rare. You've seen how to use pointers to pointers and how arrays of pointers can be very useful when dealing with strings. You've also learned how C treats multidimensional arrays as being arrays of arrays, and you've seen how to use pointers with such arrays. Finally, you've learned how to declare and use pointers to functions, an important and flexible programming tool.

Capabilities of the C Language
This has been a long and involved unit. Although some of its topics are a bit complicated, they're exciting as well. With this unit, you're really getting into some of the sophisticated capabilities of the C language. Power and flexibility are among the main reasons C is such a popular language.

Unit 2. Day 16 Using Disk Files

Many of the programs you write use disk files for one purpose or another. This unit will cover file management, data storage, and configuration information.

Today, you learn

- Relating streams to disk files.
- C's two disk file types.
- Opening a file.
- Writing data to a file.
- Reading data from a file.
- Closing a file.
- Disk file management.
- Using temporary files.

Topic 2.1: Streams and Disk Files

Streams and Disk Files
As you learned on Day 14, "Working with the Screen, Printer, and Keyboard," C performs all input and output by means of streams. You saw how to use C's predefined streams that are connected to specific devices such as the keyboard, screen, and printer. Disk file streams essentially work the same way—one of the advantages of stream input/output. The major difference with disk file streams is that your program must explicitly create a stream associated with a specific disk file.

Topic 2.2: Types of Disk Files

Modes
On Day 14, "Working with the Screen, Printer, and Keyboard," you saw that C streams come in two flavors: *text* and *binary*. You can associate either type of stream with a file, and it's important that you understand the distinction in order to use the proper mode for your files.

Text Mode
A *text stream* (or text-mode file) is a sequence of lines; each line contains zero or more characters and ends with one or more characters that signal *end-of-line*. Maximum line length is 255 characters. A "line" is not a string; there is no terminating \0. When you use a text-mode stream, translation occurs between C's newline character \n and whatever character(s) the operating system uses to mark end-of-line on disk files. On PC systems, it's a carriage-return linefeed (CR-LF) combination. When data is written to a text-mode file, each \n is translated to a CR-LF; when data is read from a disk file, each CR-LF is translated to a \n.

Binary Mode
A *binary stream* (or binary mode file) is anything else. Any and all data is written and read unchanged. The null and end-of-line characters have no special significance.

Which Mode with Which Function
Some file input/output functions are restricted to one file mode, whereas other functions can use either mode. This unit teaches you which mode to use with what functions.

Topic 2.3: Filenames

Rules for Naming Files
You must use filenames when dealing with disk files. Filenames are stored in strings just like other text data. The names are the same as those used by the operating system, and they must follow the same rules. In DOS, a complete filename consists of a 1- to 8-character name, optionally followed by a period and a 1- to 3-character extension. The characters allowed in a filename are the letters a–z, numerals 0–9, and certain other characters like _, !, and $.

Filenames with Drive and Directory Information
A filename in a C program also can contain drive and directory information or both. Remember, DOS uses the backslash character to separate directory names. Click the Example link.

Example

You also know that the backslash character has a special meaning to C when it is in a string. To represent the backslash character itself, you must precede it by a backslash. Thus, in a C program, you represent the filename as follows:

```
char *filename = "c:\\data\\list.txt";
```

If you are entering a filename from the keyboard, however, enter only a single backslash.

To DOS, the name

```
c:\data\list.txt
```

refers to a file named LIST.TXT in the directory \DATA on drive C:.

Topic 2.4: Opening a File for Use

Opening a File
The process of creating a stream linked to a disk file is called *opening* the file. When you open a file, it becomes available for reading (meaning that data is input from the file to the program), writing (meaning that data from the program is saved in the file), or both. When you're done using the file, you must close it. Closing a file is covered later in the unit.

fopen() Syntax
To open a file, you use the fopen() library function. The prototype of fopen() is located in STDIO.H and reads as follows:

```
FILE *fopen(const char *filename,
            const char *mode);
```

Return Value
This prototype tells you that fopen() returns a pointer to type FILE, which is a structure declared in STDIO.H. The members of the FILE structure are used by the program in the various file access operations, but you don't need to be concerned about them. However, for each file that you want to open, you must declare a pointer to type FILE. When you call fopen(), that function creates an instance of the FILE structure and returns a pointer to that structure. You use this pointer in all subsequent operations on the file. If fopen() fails, it returns NULL.

filename Argument
The argument `filename` is the name of the file to be opened. As noted earlier, filename can contain drive and directory specifications. The filename argument can be a literal string enclosed in double quotation marks or a pointer to a string stored elsewhere in memory.

mode Argument
The argument `mode` specifies the mode in which to open the file. In this context, mode controls whether the file is

The argument mode specifies the mode in which to open the file. In this context, mode controls whether the file is binary or text and whether it is for reading, writing, or both. Possible values for mode are given here.

The default file mode is text. To open a file in binary mode, you append a `b` to the mode argument. Thus, a mode argument of `a` would open a text-mode file for appending, whereas `ab` would open a binary-mode file for appending.

Table 16.1: Values of Mode

mode	Meaning
`r`	Open the file for reading. If the file does not exist, fopen() returns NULL.
`w`	Open the file for writing. If a file of the specified name does not exist, it is created. If the file already exists, existing data in the file is erased.
`a`	Open the file for appending. If a file of the specified name does not exist, it is created. If the file already exists, new data is appended at the end of the file.
`r+`	Open the file for reading and writing. If a file of the specified name does not exist, it is created. If the file already exists, new data is added at the start of the file, overwriting existing data.
`w+`	Open the file for reading and writing. If a file of the specified name does not exist, it is created. If the file already exists, it is overwritten.
`a+`	Open a file for reading and appending. If a file of the specified name does not exist, it is created. If the file already exists, new data is appended to the end of the file.

Error Conditions

Remember that fopen() returns NULL if an error occurs. Error conditions that can cause a return value of NULL include the following:

- Using an invalid filename.
- Trying to open a file on a disk that isn't ready (the drive door is not closed or the disk is not formatted, for example).
- Trying to open a file in a nonexistent directory or on a nonexistent disk drive.
- Trying to open a nonexistent file in mode "r."

Whenever you use fopen(), you need to test for the occurrence of an error. There's no way to tell exactly which error occurred, but you can display a message to the user and try to open the file again; or you can end the program.

Example Program

The program in Listing 16.1 demonstrates fopen().

Listing 16.1: Using fopen() to Open Disk Files in Various Modes

```
Code  1: /* LIST1601.c: Day 16 Listing 16.1 */
      2: /* Demonstrates the fopen() function. */
      3:
      4: #include <stdio.h>
      5:
      6: int main(void) {
      7:   FILE *fp;
      8:   char ch, filename[40], mode[4];
      9:
     10:   while (1) {
     11:     /* Input filename and mode. */
     12:
     13:     printf("\nEnter a filename: ");
     14:     gets(filename);
     15:     printf("\nEnter a mode (max 3 characters): ");
     16:     gets(mode);
     17:
```

```
18:     /* Try to open the file. */
19:
20:     if ((fp=fopen(filename, mode)) != NULL) {
21:       printf("\nSuccessful opening %s in mode "
22:              "%s.\n",
23:           filename, mode);
24:       fclose(fp);
25:       puts("Enter x to exit, "
26:            "press ENTER to continue. ");
27:       if ((ch = getchar()) == 'x')
28:         break;
29:       else
30:         continue;
31:     }
32:     else {
33:       fprintf(stderr, "\nError opening file %s "
34:           "in mode %s.\n",
35:           filename, mode);
36:       puts("Enter x to exit,"
37:            "press ENTER to try again. ");
38:       if ((ch = getchar()) == 'x')
39:         break;
40:       else
41:         continue;
42:     }
43:   }
44:   return 0;
45: }
```

Output

```
Enter a filename: list1601.c

Enter a mode (max 3 characters): r

Successful opening list1601.c in mode r.
Enter x to exit, any other to continue.

Enter a filename: test.c

Enter a mode (max 3 characters): w

Successful opening test.c in mode w.
Enter x to exit, any other to continue.
```

Description

The program prompts you for both the filename and the mode specifier on lines 13 – 16. After getting the names, line 20 attempts to open the file and assign its file pointer to fp. As an example of good programming practice, the if statement on line 20 checks to see that the opened file's pointer is not equal to NULL. If fp is not equal to NULL, a message stating that the open was successful and that the user can continue is printed. If the file pointer is NULL, the else condition of the if loop executes. The else condition on lines 32 – 42 prints a message stating there was a problem. It then prompts the user to determine whether the program should continue.

You can experiment with different names and modes and see which ones give you an error. If an error occurs, you are given the choice of entering the information again or quitting the program. To force an error, enter an invalid filename such as `[]`.

Topic 2.5: Writing and Reading File Data

Writing Data to a File

A program that uses a disk file can write data to a file, read data from a file, or do a combination of both. You can write data to a disk file three different ways:

Formatted Output

Character Output
Direct Output

Formatted Output
You can use formatted output to save formatted data to a file. You should use formatted output only with text-mode files. The primary use of formatted output is to create files containing text and numeric data to be read by other programs, such as a spreadsheet or database. You rarely, if ever, use formatted output to create a file to be read again by a C program.

Character Output
You can use character output to save single characters or lines of characters to a file. Although technically it is possible to use character output with binary-mode files, it can be tricky. You should restrict character-mode output to text files. The main use for character output is to save text (but not numeric) data in a form that can be read by C as well as other programs, such as word processors.

Direct Output
You can use direct output to save the contents of a section of memory directly to a disk file. This method is for binary files only. Direct output is the best way to save data for later use by a C program.

Reading Data from a File
When you want to read data from a file, you have the same three options: formatted input, character input, or direct input. The type of input you use in a particular case depends almost entirely on the nature of the file being read.

Guidelines Versus Rules
The previous descriptions of the three types of file input and output suggest tasks best suited for each type of output. This is by no means a set of strict rules. The C language is very flexible (one of its advantages!), so a clever programmer can make any type of file output suit almost any need. As a beginning programmer, you might find things easier if you follow these guidelines, at least initially.

Topic 2.5.1: Formatted File Input and Output

Formatted File Input and Output
Formatted file input/output deals with text and numeric data that is formatted in a specific way. It is directly analogous to formatted keyboard input and screen output done with the printf() and scanf() functions, as described on Day 14, "Working with the Screen, Printer, and Keyboard." Formatted output is discussed first, followed by input.

Topic 2.5.2: Formatted File Output

fprintf() Syntax
Formatted file output is done with the library function fprintf(). The prototype of fprintf() is in the header file STDIO.H and reads as follows:

```
int fprintf(FILE *file-ptr,
            char *format-string, ...);
```

file-ptr Argument
The first argument is a pointer to type FILE. To write data to a particular disk file, you pass the pointer that was returned when you opened the file with fopen().

format-string Argument
The second argument is the format string (also known as control-string). You've learned about format strings before, in the discussion of printf() on Day 14, "Working with the Screen, Printer, and Keyboard." The format string used by fprintf() follows exactly the same rules as for printf(). Please refer to Day 14, "Working with the Screen, Printer, and

Keyboard," for details.

... Argument

The final argument is `...` What does that mean? In a function prototype, ellipses represent a variable number of arguments. In other words, in addition to the file pointer and the format string arguments, fprintf() takes zero, one, or more additional arguments. This is just like printf(). These arguments are the names of the variable(s) to be output to the specified stream.

fprintf() Versus printf()

Remember, fprintf() works just like printf() except it sends its output to the stream specified in the argument list. In fact, if you specify a stream argument of stdout, fprintf() is identical to printf().

The program in Listing 16.2 uses fprintf().

Listing 16.2: Demonstrating the Equivalence of fprintf() Formatted Output Both to a File and to stdout

Code

```
/* LIST1602.c: Day 16 Listing 16.2 */
/* Demonstrates the fprintf() function. */

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

void clear_kb(void);

int main(void) {
  FILE *fp;
  float data[5];
  int count;
  char filename[20];

  puts("Enter 5 floating point numerical values.");

  for (count = 0; count < 5; count++)
    scanf("%f", &data[count]);

  /* Get the filename and open the file. First */
  /* clear stdin of any extra characters. */

  clear_kb();

  puts("Enter a name for the file.:");
  gets(filename);

  if ((fp = fopen(filename, "w")) == NULL) {
    fprintf(stderr, "Error opening file %s.",
            filename);
    exit(1);
  }

  /* Write the numerical data to the file and to */
  /* stdout. */

  for (count = 0; count < 5; count++) {
    fprintf(fp, "\ndata[%d] = %f",
            count, data[count]);
    fprintf(stdout, "\ndata[%d] = %f",
            count, data[count]);
  }

  fclose(fp);
  return 0;
}

void clear_kb(void) {
/* Clears stdin of any waiting characters. */
  char junk[80];
  gets(junk);
}
```

Output

```
Enter 5 floating point numerical values.
```

Output	3.14159 9.99 1.50 3. 1000.0001 Enter a name for the file. numbers.txt data[0] = 3.141590 data[1] = 9.990000 data[2] = 1.500000 data[3] = 3.000000 data[4] = 1000.000122
Description	This program uses fprintf() on lines 38 – 41 to send some formatted text and numeric data to stdout and a disk file. The only difference between the two lines is the first argument. After running the program, use your editor to look at the contents of the file. They should be an exact duplicate of the screen output. Note that Listing 16.2 uses the clear_kb() function developed on Day 14, "Working with the Screen, Printer, and Keyboard." This is necessary to remove from stdin any extra characters that might be left over from the call to scanf(). If you don't clear stdin, these extra characters (specifically the newline) are read by the gets() that inputs the filename, and the result is a file creation error.

Topic 2.5.3: Formatted File Input

fscanf() Syntax
For formatted file input, use the fscanf() library function, which is used like scanf() (see Day 14, "Working with the Screen, Printer, and Keyboard") except that input comes from a specified stream instead of from stdin. The prototype for fscanf() is

```
int fscanf(FILE *file-ptr,
           const char *format-string, ...);
```

Arguments
The argument `file-ptr` is the pointer to type FILE returned by fopen(), and `format-string` is a pointer to the format string that specifies how fscanf() is to read the input. The components of the format string (also known as control-string) are the same as for scanf(). Finally, the ellipses ... indicate one or more additional arguments, the addresses of the variables where fscanf() is to assign the input.

fscanf() Versus scanf()
To use fscanf() you might want to review the section on scanf() on Day 14, "Working with the Screen, Printer, and Keyboard." The function fscanf() works exactly the same as scanf(), except that characters are taken from the specified stream rather than stdin.

Example Program
To demonstrate fscanf(), you need a text file containing some numbers or strings in a format readable by the function. Use your editor to create a file named input.txt and enter five floating point numbers with some spacing between them (spaces or newlines). For example, your file might look like this:

```
123.45     87.001
100.02
0.00456    1.0005
```

Now, compile and run the program in Listing 16.3.

Listing 16.3: Using fscanf() to Read Formatted Data from a Disk File

```
Code  1: /* LIST1603.c: Day 16 Listing 16.3 */
      2: /* Reading formatted file data with fscanf(). */
```

```
 3:
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6:
 7: int main(void) {
 8:   float f1, f2, f3, f4, f5;
 9:   FILE *fp;
10:
11:   if ((fp = fopen("INPUT.TXT", "r")) == NULL) {
12:     fprintf(stderr, "Error opening file.");
13:     exit(1);
14:   }
15:
16:   fscanf(fp, "%f %f %f %f %f",
17:          &f1, &f2, &f3, &f4, &f5);
18:   printf("The values are %f, %f, %f, %f, and %f.",
19:         f1, f2, f3, f4, f5);
20:
21:   fclose(fp);
22:   return 0;
23: }
```

Output

```
The values are 123.45, 87.0001, 100.02, 0.00456, and 1.0005.
```

Note: Precision of the values may cause some numbers to not display as the exact values entered. For example, `100.02` may appear as `100.01999`.

Description

This program reads the five values from the file you created and then displays them on the screen. The fopen() call on line 11 opens the file for read mode. It also checks to see that the file opened correctly. If the file was not opened, an error message is displayed on line 12, and the program exits (line 13). Line 16 demonstrates the use of the fscanf() function. With the exception of the first parameter, this is identical to scanf(), which you have been using throughout the course. The first parameter points to the file that you want the program to read. You can do further experiments with fscanf(), creating input files with your programming editor and seeing how fscanf() reads the data.

Topic 2.5.4: Character Input and Output

Character I/O
When used with disk files, the term *character I/O* refers to single characters as well as lines of characters. Remember, a line is a sequence of zero or more characters terminated by the newline character. Use character I/O with text mode files. The sections that follow describe character input/output functions, and they are followed by a demonstration program.

Character Input
There are three character input functions: getc() and fgetc() for single characters and fgets() for lines.

getc() Syntax
The functions getc() and fgetc() are identical and can be used interchangeably. They input a single character from the specified stream. The prototype of getc() is in STDIO.H.

```
int getc(FILE *file-ptr);
```

Argument
The argument file-ptr is the pointer returned by fopen() when the file was opened.

Return Value
The function returns the character input or EOF on error.

Syntax
To read a line of characters from a file, use the fgets() library function. The prototype is

```
char *fgets(char *s, int num, FILE *file-ptr);
```

Argument
The argument `s` is a pointer to a buffer where the input is to be stored, `num` is the maximum number of characters to be input, and `file-ptr` is the pointer to type FILE that was returned by fopen() when the file was opened.

Process
When called, fgets() reads characters from file-ptr into memory, starting at the location pointed to by s. Characters are read until a newline is encountered or num-1 characters have been read. By setting num equal to the number of bytes allocated for the buffer s, you prevent input from overwriting memory beyond allocated space. (The num-1 is to allow space for the terminating \0 that fgets() adds on.)

Return Value
If successful, fgets() returns s. Two types of errors can occur.

- If a read error or EOF is encountered before any characters have been assigned to s, NULL is returned and the memory pointed to by s is unchanged.
- If a read error or EOF is encountered after one or more characters have been assigned to s, NULL is returned, and the memory pointed to by s contains garbage.

Entire Line Not Necessarily Inputted
You can see that fgets() does not necessarily input an entire line (that is, up to the next newline character). If num-1 characters are read before a newline is encountered, fgets() stops. The next read operation from the file starts where the last one leaves off. To be sure that fgets() reads in entire strings, stopping only at newlines, be sure that the size of your input buffer and the corresponding value of num passed to fgets() are large enough.

Character Output
You also need to know about two character output functions, putc() and fputs().

Syntax
The library function putc() writes a single character to a specified stream. Its prototype, in STDIO.H, reads

```
int putc(int c, FILE *file-ptr);
```

Arguments
The argument `c` is the character to output. As with other character functions, it is formally called a type int, but only the lower-order byte is used. The argument `file-ptr` is the pointer associated with the file (the pointer returned by fopen() when the file was opened).

Return Value
The function putc() returns the character just written if successful or EOF if an error occurs. The symbolic constant EOF is defined in STDIO.H, and it has the value –1. Because no "real" character has that numeric value, EOF can be used as an error indicator (with text-mode files only).

fputs() Versus puts()
To write a line of characters to a stream, use the library function fputs(). This function works just like puts(), covered on Day 14, "Working with the Screen, Printer, and Keyboard." The only difference is that with fputs(), you can specify the output stream. Also, fputs() does not add a newline at the end of the string; you must explicitly include it, if desired.

Syntax
Its prototype in STDIO.H is

```
char fputs(char *s, FILE *file-ptr);
```

Arguments

The argument `s` is a pointer to the null-terminated string to be written, and `file-ptr` is the pointer to type FILE returned by fopen() when the file was opened. The string pointed to by s is written to the file, minus its terminating \0.

Return Value
The function fputs() returns a non-negative value if successful, EOF on error.

Topic 2.5.5: Direct File Input and Output

Direct File I/O
You use direct file I/O most often when you save data to be read later by the same or a different C program. Direct I/O is used only with binary-mode files. With direct output, blocks of data are written from memory to disk. Direct input reverses the process: a block of data is read from a disk file into memory. For example, a single direct output function call can write an entire array of type double to disk, and a single direct input function call can read the entire array from disk back into memory. The direct I/O functions are fread() and fwrite().

Description
The fwrite() library function writes a block of data from memory to a binary-mode file.

Syntax
Its prototype, in STDIO.H, is

```
int fwrite(void *a-ptr, int el-size,
           int num, FILE *file-ptr);
```

a-ptr Argument
The argument `a-ptr` is a pointer to the region of memory holding the data to be written to the file. The pointer type is void; it can be a pointer to anything.

el-size Argument
The argument `el-size` specifies the size, in bytes, of the individual data items, and `num` specifies the number of items to be written. For example, if you want to save a 100-element integer array, el-size would be 4 (because each int occupies 4 bytes), and num would be 100 (because the array contains 100 elements). To obtain the el-size argument, you can use the sizeof() operator.

file-ptr Argument
The argument `file-ptr` is, of course, the pointer to type FILE, returned by fopen() when the file was opened.

Return Value
The fwrite() function returns the number of items written on success; if the value returned is less than `num`, it means some error has occurred. To check for errors, you usually program fwrite() as follows:

```
if( (fwrite(buf, size, count, fp)) != count)
     fprintf(stderr, "Error writing to file.");
```

Here are some examples of using fwrite(). Click the Examples link.

Examples

To write a single type double variable x to a file:

```
fwrite(&x, sizeof(double), 1, fp);
```

To write an array data[ ] of 50 structures of type address to a file, you have two choices:

```
fwrite(data, sizeof(address), 50, fp);
```

or

```
fwrite(data, sizeof(data), 1, fp);
```

The first method writes the array as 50 elements, with each element having the size of a single type address structure. The second method treats the array as a single "element." The two methods accomplish exactly the same thing.

Example Program
After fread() is explained in the following section, a program demonstrating both commands is presented.

Description
The fread() library function reads a block of data from a binary-mode file into memory

Syntax
Its prototype, in STDIO.H, is

```
int fread(void *a-ptr, int el-size,
          int num, FILE *file-ptr);
```

a-ptr Argument
The argument `a-ptr` is a pointer to the region of memory that receives the data read from the file. As with fwrite(), the pointer type is void.

el-size and num Arguments
The argument `el-size` specifies the size, in bytes, of the individual data items being read, and `num` specifies the number of items to read, paralleling the arguments used by fwrite(). Again, the sizeof() operator is often used to provide the `el-size` argument.

file-ptr Argument
The argument `file-ptr` is (as always) the pointer to type FILE that was returned by fopen() when the file was opened.

Return Value
The fread() function returns the number of items read; this can be less than `num` if end-of-file was reached or an error occurred.

Example Program
The program in Listing 16.4 demonstrates the use of fwrite() and fread().

Using fread() Properly
When you save data with fwrite(), there's not much that can go wrong besides some type of disk error. With fread(), you need to be careful, however. As far as fread() is concerned, the data on the disk is just a sequence of bytes. The function has no way of knowing what it represents. Click the Example link.

Example

When writing programs, you must be sure that fread() is used properly, reading data into the appropriate types of variables and arrays. Notice that in Listing 16.4, all calls to fopen(), fwrite(), and fread() are checked to ensure that they worked correctly.

Listing 16.4: Using fwrite() and fread() for Direct File Access

```
Code  1: /* LIST1604.c: Day 16 Listing 16.4 */
      2: /* Direct file I/O with fwrite() and fread(). */
      3:
      4: #include <stdio.h>
      5: #include <stdlib.h>
      6:
      7: #define SIZE 20
      8:
      9: int main(void) {
```

```
 9: int main(void) {
10:   int count, array1[SIZE], array2[SIZE];
11:   FILE *fp;
12:
13:   /* Initialize array1[]. */
14:
15:   for (count = 0; count < SIZE; count++)
16:     array1[count] = 2 * count;
17:
18:   /* Open a binary mode file. */
19:
20:   if ((fp = fopen("direct.txt", "wb")) == NULL) {
21:     fprintf(stderr, "Error opening file.");
22:     exit(1);
23:   }
24:   /* Save array1[] to the file. */
25:
26:   if (fwrite(array1,sizeof(int),SIZE, fp) != SIZE){
27:     fprintf(stderr, "Error writing to file.");
28:     exit(1);
29:   }
30:
31:   fclose(fp);
32:
33:   /* Now open the same file for reading in */
34:   /* binary mode. */
35:
36:   if ((fp = fopen("direct.txt", "rb")) == NULL) {
37:     fprintf(stderr, "Error opening file.");
38:     exit(1);
39:   }
40:
41:   /* Read the data into array2[]. */
42:
43:   if (fread(array2,sizeof(int),SIZE, fp) != SIZE) {
44:     fprintf(stderr, "Error reading file.");
45:     exit(1);
46:   }
47:
48:   fclose(fp);
49:
50:   /* Now display both arrays to show they're */
51:   /* the same. */
52:
53:   for (count = 0; count < SIZE; count++)
54:     printf("%d\t%d\n", array1[count],
55:             array2[count]);
56:   return 0;
57: }
```

Output

```
0       0
2       2
4       4
6       6
8       8
10      10
12      12
14      14
16      16
18      18
20      20
22      22
24      24
26      26
28      28
30      30
32      32
34      34
36      36
38      38
```

Description Listing 16.4 demonstrates the use of the fwrite() and fread() functions. The program initializes an array on lines 15 and 16. It then uses fwrite() on line 26 to save the array to disk. The

	program uses fread() on line 43 to read the data into a different array. Notice that if statements on lines 36 – 39 and lines 43 – 46 perform error-checking by testing to make sure that the values returned by fread() and fwrite() correspond to the size of the arrays. Finally, the program displays both arrays on the screen to show that they now hold the same data (lines 53 – 55).

A block of 100 bytes could be 100 char variables, 50 int variables, 25 long variables, or 25 float variables. If you ask fread() to read that block into memory, it obediently does so. However, if the block was saved from an array of type int, and you retrieve it into an array of type float, no error occurs, but you get strange results.

Topic 2.6: File Buffering: Closing and Flushing Files

Closing Files with fclose()
When you're done using a file, you should close it with the fclose() function. You've seen fclose() used in all the programs presented so far in this unit. Its prototype is

```
int fclose(FILE *file-ptr);
```

The argument `file-ptr` is the FILE pointer associated with the stream; fclose() returns 0 on success or -1 on error. When you close a file, the file's buffer is flushed (written to the file).

Closing Files with fcloseall()
You also can close all open streams except the standard ones (stdin, stdout, stdprn, stderr, and stdaux) by using fcloseall(). Its prototype is

```
int fcloseall(void);
```

The function fcloseall() also flushes any stream buffers and returns the number of streams closed.

Caution: fcloseall() is not an ANSI-standard command. Click the Tip button.

DO open a file before trying to read or write to it.

DON'T assume that a file access is okay. Always check after doing a read, write, or open to ensure that the function worked.

DO use the sizeof() operator with the fwrite() and fread() functions.

DO close all files that you have opened.

DON'T use fcloseall() unless you have a reason to close all the streams.

Closing Streams Explicitly
When a program terminates (either by reaching the end of main() or executing the exit() function), all streams are automatically flushed and closed. However, it's a good idea to close streams explicitly—particularly those linked to disk files—as soon as you are finished using them. The reason has to do with stream buffers.

Buffers

When you create a stream linked to a disk file, a buffer is automatically created and associated with the stream. A *buffer* is a block of memory used for temporary storage of data being written to and read from the file.

Buffers are needed because disk drives are block-oriented devices, which means that they operate most efficiently when data is read and written in blocks of a certain size. The size of the ideal block differs depending on the specific hardware in use and is typically on the order of a few hundred to a thousand bytes. You don't need to be concerned about the exact block size, however.

Stream Buffers
The buffer associated with a file stream serves as an interface between the stream (which is character oriented) and the disk hardware (which is block oriented). As your program writes data to the stream, the data is saved in the buffer until the buffer is full, and then the entire contents of the buffer are written, as a block, to the disk. An analogous process occurs when reading data from a disk file. The creation and operation of the buffer is entirely automatic; you don't have to be concerned with it. (C does offer some functions for buffer manipulation, but they are beyond the scope of this course.)

Losing Data in the Buffer
In practical terms, this buffer operation means that during program execution, data that your program "wrote" to the disk might still be in the buffer, not on the disk. If your program "hangs up," if there's a power failure, or some other problem occurs, the data that's still in the buffer might be lost, and you won't know what is contained in the disk file.

Flushing Files with fflush() and flushall()
You can flush a stream's buffers without closing it by using the fflush() or flushall() library functions. Use fflush() when you want a file's buffer written to disk while still using the file. Use flushall() to flush the buffers of all open streams

Syntax
The prototypes of these functions are

```
int fflush(FILE *file-ptr);
```

and

```
int flushall(void);
```

Arguments
The argument `file-ptr` is the FILE pointer returned by fopen() when the file was opened. If a file was opened for writing, fflush() writes its buffer to disk. If the file was opened for reading, the buffer is cleared.

Return Values
The function fflush() returns 0 on success or EOF if an error occurred. The function flushall() returns the number of open streams.

Non-ANSI
Caution: flushall() is not an ANSI-standard command.

Topic 2.7: Sequential Versus Random File Access

The Position Indicator
Every open file has a file position indicator associated with it. The position indicator specifies where read and write operations take place in the file. The position is always given in terms of bytes from the beginning of the file. When a new file is opened, the position indicator is always at the beginning of the file, position 0. (Because the file is new with a length of 0, there's no other location to indicate.) When an existing file is opened, the position indicator is at the end of the file if the file was opened in append mode, or at the beginning of the file if the file was opened in any other mode.

Sequential Access

The file input/output functions covered earlier in this unit make use of the position indicator. Writing and reading operations occur at the location of the position indicator and update the position indicator as well. For example, if you open a file for reading and read in 10 bytes, you input the first 10 bytes in the file (the bytes at positions 0 through 9). After the read operation, the position indicator is at position 10, and the next read operation begins there. Thus, if you want to read all the data in a file *sequentially* or sequentially write data to a file, you don't need to be concerned about the position indicator, because the stream I/O functions take care of it automatically.

Random Access
When you need more control, use the C library functions that enable you to determine and change the value of the position indicator. By controlling the position indicator, you can perform *random* file access. Here "random" means you can read data from, or write data to, any position in a file without reading or writing all the preceding data.

The ftell() and rewind() functions control the position indicator.

The rewind() Function
To set the position indicator to the beginning of the file, use the library function rewind(). After calling rewind(), the position indicator for the file is set at the beginning of the file (byte 0). Use rewind() if you've read some data from a file and want to start reading from the beginning of the file again without closing and reopening the file.

Syntax
Its prototype, in STDIO.H, is

```
void rewind(FILE *file-ptr);
```

Argument
The argument `file-ptr` is the FILE pointer associated with the stream.

The ftell() Function
To determine the value of a file's position indicator, use ftell().

Syntax
This function's prototype, located in STDIO.H, reads

```
long ftell(FILE *file-ptr);
```

Argument
The argument `file-ptr` is the FILE pointer returned by fopen() when the file was opened.

Return Value
The function ftell() returns a long that gives the current file position in bytes from the start of the file (the first byte is at position 0). If an error occurs, ftell() returns -1L (a type long –1).

To get a feel for the operation of rewind() and ftell(), click the button to look at the program in Listing 16.5.

Listing 16.5: Using ftell() and rewind()

```
Code  1:  /* LIST1605.c: Day 16 Listing 16.5 */
      2:  /* Demonstrates ftell() and rewind(). */
      3:
      4:  #include <stdio.h>
      5:  #include <stdlib.h>
      6:
      7:  #define BUFLEN 6
      8:
      9:  char msg[] = "abcdefghijklmnopqrstuvwxyz";
     10:
     11:  int main(void) {
     12:    FILE *fp;
     13:    char buf[BUFLEN];

     14:
     15:    if ((fp = fopen("TEXT.TXT", "w")) == NULL) {
     16:      fprintf(stderr, "Error opening file.");
```

```
17:     exit(1);
18:   }
19:   if (fputs(msg, fp) == EOF) {
20:     fprintf(stderr, "Error writing to file.");
21:     exit(1);
22:   }
23:
24:   fclose(fp);
25:
26:   /* Now open the file for reading. */
27:
28:   if ((fp = fopen("TEXT.TXT", "r")) == NULL) {
29:     fprintf(stderr, "Error opening file.");
30:     exit(1);
31:   }
32:   printf("\nImmediately after opening, "
33:          "position = %ld", ftell(fp));
34:
35:   /* Read in 5 characters. */
36:
37:   fgets(buf, BUFLEN, fp);
38:   printf("\nAfter reading in %s, "
39:          "position = %ld",  buf, ftell(fp));
40:
41:   /* Read in the next 5 characters. */
42:
43:   fgets(buf, BUFLEN, fp);
44:   printf("\n\nThe next 5 characters are %s, "
46:          "and position now = %ld",
47:       buf, ftell(fp));
48:
49:   /* Rewind the stream. */
50:
51:   rewind(fp);
52:
53:   printf("\n\nAfter rewinding, the position "
54:          "is back at %ld",
55:       ftell(fp));
56:
57:   /* Read in 5 characters. */
58:
59:   fgets(buf, BUFLEN, fp);
60:   printf("\nand reading starts at the beginning "
61:          "again: %s", buf);
62:   fclose(fp);
63:   return 0;
64: }
```

Output

```
Immediately after opening, position = 0
After reading in abcde, position = 5

The next 5 characters are fghij, and position now = 10

After rewinding, the position is back at 0
and reading starts at the beginning again: abcde
```

Description

This program writes a string, msg, to a file called TEXT.TXT. Lines 15 – 18 open TEXT.TXT for writing and test to ensure that the open was successful.

Lines 19 – 22 write msg to the file using fputs() and again check to ensure that the write was successful. Line 24 closes the file with fclose(), completing the process of creating a file for the rest of the program to use.

Lines 28 – 31 open the file again, only this time for reading. Lines 32 – 33 print the return value of ftell(). Notice that this position is at the beginning of the file.

Line 37 performs a gets() to read 5 characters. The five characters and the new file position are printed on lines 38 – 39. Notice that ftell() returns the correct offset.

	Line 51 calls rewind() to put the pointer back to the beginning of the file, before lines 53 – 54 print the file position again. This should confirm for you that rewind() resets the position. An additional read on line 59 further confirms that the program is indeed back at the beginning of the file. Line 62 closes the file before ending the program.

Topic 2.7.1: The fseek() Function

Description
More precise control over a stream's position indicator is possible with the fseek() library function. By using fseek(), you can set the position indicator anywhere in the file.

Syntax
The function prototype, in STDIO.H, is

```
int fseek(FILE *file-ptr, long offset,
          int start-pos);
```

Arguments
The argument `file-ptr` is the FILE pointer associated with the file. The distance that the position indicator is to be moved is given by `offset` in bytes. The argument `start-pos` specifies the move's relative starting point. There can be three values for `start-pos`, with symbolic constants defined in IO.H, as shown here.

Return Value
The function fseek() returns 0 if the indicator was successfully moved or nonzero if an error occurred. The program in Listing 16.6 uses fseek() for random file access.

Table 16.2: Possible start-pos Values for fseek()

Constant	Value	Meaning
`SEEK_SET`	0	Move the indicator `offset` bytes from the beginning of the file.
`SEEK_CUR`	1	Move the indicator `offset` bytes from its current position.
`SEEK_END`	2	Move the indicator `offset` bytes from the end of the file.

Listing 16.6: Random File Access with fseek()

```
Code  1: /* LIST1606.c: Day 16 Listing 16.6 */
      2: /* Demonstrates random access with fseek(). */
      3:
      4: #include <stdio.h>
      5: #include <stdlib.h>
      6: #include <io.h>
      7:
      8: #define MAX 50
      9:
     10: int main(void) {
     11:   FILE *fp;
     12:   int data, count, array[MAX];
     13:   long offset;
     14:
     15:   /* Initialize the array. */
     16:
     17:   for (count = 0; count < MAX; count ++)
     18:     array[count] = count * 10;
     19:

     20:   /* Open a binary file for writing. */
     21:
     22:   if ((fp = fopen("RANDOM.DAT", "wb")) == NULL) {
```

```
23:     fprintf(stderr, "\nError opening file.");
24:     exit(1);
25:   }
26:
27:   /* Write the array to the file, then close it. */
28:
29:   if ((fwrite(array,sizeof(int), MAX, fp)) != MAX){
30:     fprintf(stderr, "Error writing data to file.");
31:     exit(1);
32:   }
33:
34:   fclose(fp);
35:
36:   /* Open the file for reading. */
37:
38:   if ((fp = fopen("RANDOM.DAT", "rb")) == NULL) {
39:     fprintf(stderr, "\nError opening file.");
40:     exit(1);
41:   }
42:
43:   /* Ask user which element to read. Input the */
44:   /* element and display it, quitting when -1 is */
45:   /* entered. */
46:
47:   while (1) {
48:     printf("\nEnter element to read, 0-%d, "
49:            "-1 to quit: ", MAX - 1);
50:       scanf("%ld", &offset);
51:
52:     if (offset < 0)
53:       break;
54:     else if (offset > MAX - 1)
55:       continue;
56:
57:     /* Move the position indicator to the */
58:     /* specified element. */
59:
60:     if ((fseek(fp, (offset * sizeof(int)),
61:       SEEK_SET)) != 0) {
62:       fprintf(stderr, "\nError using fseek().");
63:       exit(1);
64:     }
65:
66:     /* Read in a single integer. */
67:
68:     fread(&data, sizeof(int), 1, fp);
69:
70:     printf("\nElement %ld has value %d.",
71:            offset, data);
72:   }
73:   fclose(fp);
74:   return 0;
75: }
```

Output

```
Enter element to read, 0-49, -1 to quit: 5

Element 5 has value 50.
Enter element to read, 0-49, -1 to quit: 6

Element 6 has value 60.
Enter element to read, 0-49, -1 to quit: 49

Element 49 has value 490.
Enter element to read, 0-49, -1 to quit: 1

Element 1 has value 10.
Enter element to read, 0-49, -1 to quit: 0

Element 0 has value 0.
Enter element to read, 0-49, -1 to quit: -1
```

Description	Lines 15 – 34 are similar to the previous program. Lines 17 and 18 initialize an array called data with 50 type int values. Then the array is written to a binary file called RANDOM.DAT. You know it is binary because the file was opened with mode "wb" on line 22. Line 38 reopens the file in binary read mode before going into an infinite while loop. The while loop prompts users to enter the number of the array element that they wish to read. Notice that lines 52 – 55 check to see that the entered element is within the range of the file. Does C let you read an element that was beyond the end of the file? Yes. Like going beyond the end of an array with values, C also lets you read beyond the end of a file. If you do read beyond the end (or before the beginning), your results are unpredictable. It is always best to check what you are doing (as lines 52 – 55 do in this listing). After receiving the element to find, line 60 jumps to the appropriate offset with a call to fseek(). Because SEEK_SET is being used, the seek is done from the beginning of the file. Notice that the distance into the file is not just offset, but offset multiplied by the size of the elements being read. Line 68 then reads the value, and line 70 prints it.

Topic 2.8: Detecting the End of a File

Two Ways to Detect end-of-file
At times you know exactly how long a file is, so there's no need to be able to detect the file's end. For example, if you used fwrite() to save a 100-element integer array, you know the file is 200 bytes long. At other times, however, you don't know how long the file is, but you still want to read data from the file, starting at the beginning and proceeding to the end. There are two ways to detect end-of-file.

Using the EOF Character
When reading from a text-mode file character-by-character, you can look for the EOF character. The symbolic constant EOF is defined in STDIO.H as –1, a value never used by a "real" character. When a character input function reads EOF from a text-mode stream, you can be sure that you have reached the end of the file. For example, you could write:

```
while ((c = fgetc(fp)) != EOF)
```

Click the Tip button on the toolbar.

DO check your position within a file so that you do not read beyond the end or before the beginning of a file.

DO use either rewind() or fseek(file-ptr, SEEK_SET, 0) to reset the file position to the beginning of the file.

DO use feof() to check for the end of the file when working with binary files.

DON'T use EOF with binary files.

Using the feof() Function
With a binary-mode stream, you cannot detect the end-of-file by looking for –1 because a byte of data from a binary stream could have that value, which would result in premature end of input. Instead, you can use the library function feof() (which can be used for both binary- and text-mode files)

feof() Syntax
```
int feof(FILE *file-ptr);
```

Argument
The argument `file-ptr` is the FILE pointer returned by fopen() when the file was opened.

Return Value

Return Value

The function feof() returns 0 if the end of file `file-ptr` has not been reached, or nonzero if end-of-file has been reached. If a call to feof() detects end-of-file, no further read operations are permitted until a rewind() has been done, fseek() is called, or the file is closed and reopened.

Example Program

The program in Listing 16.7 demonstrates the use of feof(). When you are prompted for a filename, enter the name of any text file—one of your C source files, for example, or a header file, such as STDIO.H. The program reads the file one line at a time, displaying each line on stdout, until feof() detects end-of-file.

Listing 16.7: Using feof() to Detect the End of a File

Code

```
 1: /* LIST1607.c: Day 16 Listing 16.7 */
 2: /* Demonstrates detecting end-of-file. */
 3:
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6:
 7: #define BUFSIZE 100
 8:
 9: int main(void) {
10:   char buf[BUFSIZE];
11:   char filename[20];
12:   FILE *fp;
13:
14:   puts("Enter name of text file to display: ");
15:   gets(filename);
16:
17:   /* Open the file for reading. */
18:
19:   if ((fp = fopen(filename, "r")) == NULL) {
20:     fprintf(stderr, "\nError opening file.");
21:     exit(1);
22:   }
23:
24:   /* If end of file is not reached, read a line */
25:   /* and display it. */
26:
27:   while (!feof(fp)) {
28:     fgets(buf, BUFSIZE, fp);
29:     printf("%s", buf);
30:   }
31:   fclose(fp);
32:   return 0;
33: }
```

Output

```
Enter name of text file to display:
list1607.c
/* LIST1607.c: Day 16 Listing 16.7 */
/* Demonstrates detecting end-of-file. */

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

#define BUFSIZE 100

int main(void) {
  char buf[BUFSIZE];
  char filename[20];
  FILE *fp;

  puts("Enter name of text file to display: ");
  gets(filename);

  /* Open the file for reading. */


  if ((fp = fopen(filename, "r")) == NULL) {
    fprintf(stderr, "\nError opening file.");
    exit(1);
```

```
  }

  /* If end of file is not reached, read a line */
  /* and display it. */

  while (!feof(fp)) {
    fgets(buf, BUFSIZE, fp);
    printf("%s", buf);
  }
  fclose(fp);
  return 0;
}
```

Description The while loop in this program (lines 27 – 30) is typical of a while used in more complex programs that do sequential processing. As long as you are not at the end of the file, you execute the lines within the while statement (lines 28 and 29). When the feof() returns a nonzero value, the loop ends, the file is closed, and the program ends.

Topic 2.9: File Management Functions

File Management
The term *file management* refers to dealing with existing files—not reading from or writing to them, but erasing, renaming, and copying them. The C standard library contains functions for erasing and renaming files, and you also can write your own file-copying function.

Topic 2.9.1: Erasing a File

The remove() Function
To erase a file, you use the library function remove().

Syntax
Its prototype is in STDIO.H, as follows:

```
int remove(const char *filename);
```

Argument
The variable `*filename` is a pointer to the name of the file to be erased. (See the section on filenames earlier in this unit.) The specified file must not be open. If the file exists, it is erased (just as if you used the DEL command from the DOS prompt) and remove() returns 0.

Return Value
If the file does not exist, is write-only, or some other error occurs, remove() returns -1.

Example Program
The short program in Listing 16.8 demonstrates the use of remove(). Be careful—if you "remove" a file, it's gone forever.

Listing 16.8: Using the Function remove() to Delete a Disk File

```
Code  1: /* LIST1608.c: Day 16 Listing 16.8 */
      2: /* Demonstrates the remove() function. */
      3:
      4: #include <stdio.h>
      5:
      6: int main(void) {
      7:   char filename[80];
      8:
      9:   printf("Enter the filename to delete: ");
     10:   gets(filename);
     11:
```

```
     12:   if (remove(filename) == 0)
     13:     printf("The file %s has been deleted.",
     14:             filename);
     15:   else
     16:     fprintf(stderr, "Error deleting the file %s.",
     17:              filename);
     18:   return 0;
     19: }
```

Output

```
>list1608
Enter the filename to delete: *.bak
Error deleting the file *.bak.

>list1608
Enter the filename to delete: list1414.bak
The file list1414.bak has been deleted.
```

Description This program prompts the user on line 9 for the name of the file to be deleted. Line 12 then calls remove() to erase the entered file. If the return value is 0, the file was removed, and a message is displayed stating this. If the return value is not zero, an error occurred and the file was not removed.

Topic 2.9.2: Renaming a File

The rename() Function
The rename() function changes the name of an existing disk file.

Syntax
The function prototype is in STDIO.H as follows:

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

Arguments
The filenames pointed to by `oldname` and `newname` follow the rules given earlier in this unit. The only restriction is that both names must refer to the same disk drive; you cannot "rename" a file to a different disk drive.

Return Value
The function rename() returns 0 on success, or -1 if an error occurs. Errors can be caused by the following conditions (among others):

- The file oldname does not exist.
- A file with the name newname already exists.
- You try to rename to another disk.

Example Program
The program in Listing 16.9 demonstrates the use of rename().

Listing 16.9: Using rename() to Change the Name of a Disk File

```
Code   1: /* LIST1609.c: Day 16 Listing 16.9 */
       2: /* Using rename() to change a filename. */
       3:
       4: #include <stdio.h>
       5:
       6: int main(void) {
       7:   char oldname[80], newname[80];
       8:
       9:   printf("Enter current filename: ");
      10:   gets(oldname);
```

```
11:   printf("Enter new name for file: ");
12:   gets(newname);
13:
14:   if (rename(oldname, newname) == 0)
15:     printf("%s has been renamed %s.",
16:             oldname, newname);
17:   else
18:     fprintf(stderr, "An error has occurred "
19:              "renaming %s.", oldname);
20:   return 0;
21: }
```

Output

```
Enter current filename: list1609.exe
Enter new name for file: rname.exe
list1609.exe has been renamed rname.exe.
```

Description

Listing 16.9 shows how powerful C can be. With only 18 lines of code, this program replaces a DOS command, and it's a much more friendly function.
Line 9 prompts for the name of the file to be renamed. Line 11 prompts for the new filename. The call to the rename() function is wrapped in an if statement on line 14. The if statement checks to ensure that the renaming of the file occurred correctly. If so, line 15 prints an affirmative message, otherwise line 18 prints a message stating that there was an error.

Topic 2.9.3: Copying a File

No Library Function Available

It's frequently necessary to make a copy of a file—an exact duplicate with a different name (or with the same name but in a different drive or directory). In DOS, this is done with the COPY command. How do you copy a file in C? There's no library function available, so you need to write your own. This may sound a bit complicated to you, but it's really quite simple thanks to C's use of streams for input and output.

Writing Your Own Copy Function

Here's the approach you take in writing your own copy function:

Step	Action
1	Open the source file for reading in binary mode (using binary mode ensures that the function can copy all sorts of files, not just text files).
2	Open the destination file for writing in binary mode.
3	Read a character from the source file. Remember, when a file is first opened, the pointer is at the start of the file, so there's no need to position the file pointer explicitly.
4	Does the function feof() indicate that you've reached the end of the source file? If so, you're done and can close both files and return to the calling program.
5	If you haven't reached end-of-file, write the character to the destination file, and then loop back to step three.

Example Program

The program in Listing 16.10 contains a function copy_file() that is passed the names of the source and destination files, and then performs the copy operation as the previous scheme outlined. If there's an error opening either file, the function does not attempt the copy and returns -1 to the calling program. Once the copy operation is complete, the program closes both files and returns 0.

Listing 16.10: A Function that Copies a File

```
Code  1: /* LIST1610.c: Day 16 Listing 16.10 */
      2: /* Copying a file. */
      3:
```

```
 4: #include <stdio.h>
 5:
 6: int file_copy(char *oldname, char *newname);
 7:
 8: int main(void) {
 9:   char source[80], destination[80];
10:
11:   /* Get the source and destination names. */
12:
13:   printf("\nEnter source file: ");
14:   gets(source);
15:   printf("\nEnter destination file: ");
16:   gets(destination);
17:
18:   if (file_copy(source, destination) == 0)
19:     puts("Copy operation successful.");
20:   else
21:     fprintf(stderr, "Error during copy operation");
22:   return 0;
23: }
24:
25: int file_copy(char *oldname, char *newname) {
26:     FILE *fold, *fnew;
27:     int c;
28:
29:     /* Open the source file for reading in */
30:     /* binary mode. */
31:
32:     if ((fold = fopen(oldname, "rb")) == NULL)
33:         return -1;
34:
35:     /* Open the destination file for writing */
36:     /* in binary mode. */
37:
38:     if ((fnew = fopen(newname, "wb")) == NULL) {
39:       fclose(fold);
40:         return -1;
41:   }
42:
43:   /* Read one byte at a time from the source; if */
44:   /* end of file has not been reached, write the */
45:   /* byte to the destination. */
46:
47:   while (1) {
48:     c = fgetc(fold);
49:
50:     if (!feof(fold))
51:       fputc(c, fnew);
52:     else
53:       break;
54:   }
55:
56:   fclose(fnew);
57:   fclose(fold);
58:
59:   return 0;
60: }
```

Output

```
Enter source file: list1610.c

Enter destination file: tmpfile.c
       Copy operation successful
```

Description

The function copy_file() works perfectly well, permitting you to copy anything from a small text file to a huge program file. It does have limitations, however. If the destination file already exists, the function erases it without asking. A good programming exercise for you would be to modify copy_file() to check whether the destination file already exists, and then query the user whether the old file should be overwritten.

main() in Listing 16.10 should look very familiar. It is nearly identical to Listing 16.9 with the exception of line 18. Instead of rename(), this function uses copy(). Because C does not

	the exception of line 10. Instead of rename(), this function uses copy(). Because C does not have a copy function, lines 25 – 60 create a copy function. Lines 32 – 33 open the source file, fold, in binary read mode. Lines 38 – 41 open the destination file, fnew, in binary write mode. Notice line 39 closes the source file if there is an error opening the destination file. The while loop in lines 47 – 54 does the actual copying of the file. Line 48 gets a character from the source file, fold. Line 50 checks to see whether the end-of-file marker was read. If the end of the file has been reached, a break statement is executed in order to get out of the while loop. If the end of the file has not been reached, the character is written to the destination file, fnew. Lines 56 and 57 close the two files before returning to main().

Topic 2.10: Using Temporary Files

Temporary Files
Some programs make use of one or more temporary files during execution. A *temporary file* is a file that is opened by the program, used for some purpose during program execution, and then deleted before the program terminates.

Using tmpnam() to Create a Valid Filename
When you create a temporary file, you don't really care what its name is because it gets deleted. All that is necessary is that you use a name that is not already in use. The C standard library includes a function tmpnam() that creates a valid filename that does not conflict with any existing file. Click the Tip button.

DON'T remove a file that you may need again.

DON'T try to rename files across drives.

DON'T forget to remove temporary files that you create. They are not automatically deleted.

tmpnam() Syntax
Its prototype in STDIO.H reads as follows:

```
char *tmpnam(char *s);
```

Arguments and Return Value
The argument `s` must be a pointer to a buffer large enough to hold the filename. You also can pass a null pointer (NULL), in which case the temporary name is stored in a buffer internal to tmpnam(), and the function returns a pointer to that buffer.

The program in Listing 16.11 demonstrates both methods of using tmpnam() to create temporary filenames.

Listing 16.11: Using tmpnam() to Create Temporary Filenames

```
Code  1: /* LIST1611.c: Day 16 Listing 16.11 */
      2: /* Demonstration of temporary filenames. */
      3:
      4: #include <stdio.h>
      5:
      6: int main(void) {
      7:   char buffer[10], *c;
      8:
      9:   /* Get a temporary name in the defined buffer. */
     10:
```

```
10:
11:   tmpnam(buffer);
12:
13:   /* Get another name, this time in the */
14:   /* function's internal buffer. */
15:
16:   c = tmpnam(NULL);
17:
18:   /* Display the names. */
19:
20:   printf("Temporary name 1: %s", buffer);
21:   printf("\nTemporary name 2: %s", c);
22:   return 0;
23: }
```

Output

```
Temporary name 1: TMP1.$$$
Temporary name 2: TMP2.$$$
```

Description

This program only generates and prints the temporary names. Line 11 stores a temporary name in the character array, buffer. Line 16 assigns the character pointer to the name returned by tmpnam() to c. Your program would have to use the generated name to open the temporary file, and then delete the file before program execution terminates, as shown below.

```
char tempname[80];
FILE *tmpfile;
tmpnam(tempname);
tmpfile = fopen(tempname, "w");
/* Use appropriate mode */

fclose(tmpfile);
remove(tempname);
```

Topic 2.11: Day 16 Q&A

Questions & Answers
Here are some questions to help you review what you have learned in this unit.

Question 1
Can I use drives and paths with filenames when using erase(), rename(), fopen(), and the file functions?

Answer
Yes. You can use full filenames with paths and drives or just the filename by itself. If you use the filename by itself, the function looks for the file in the current directory. Remember when using backslashes, you need to use the escape sequences.

Question 2
Can I read beyond the end of a file?

Answer
Yes. You also can read before the beginning of a file. Results from such reads can be disastrous. Reading files is just like working with arrays. You are looking at offsets within memory. If you are using fseek(), you should check to make sure that you do not go beyond the end of the file.

Question 3
What happens if I don't close a file?

Answer
It is good programming practice to close any files that you open. By default, the file should be closed when the

program exits; however, you should never count on this. If the file isn't closed, you might not be able to access it later because the operating system will think that the file is already in use.

Question 4
How many files can I open at once?

Answer
This question cannot be answered with a simple number. The limitation on the number of files that can be opened is based on variables set within your operating system. On DOS systems, an environment variable called FILES determines the number of files that can be opened (this variable includes programs that are running, too). Consult your operating system manuals for more information.

Question 5
Can I read a file sequentially with random access functions?

Answer
When reading a file sequentially, there is no need to use such functions as fseek(). Because the file pointer is left at the last position it occupied, it is always where you want it for sequential reads. You can use fseek() to read a file sequentially; however, you gain nothing.

Topic 2.12: Day 16 Think About Its

Think About Its
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

Topic 2.13: Day 16 Try Its

Try Its
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Write the code to close all file streams.

Answer

```
fcloseall();
```

*** Exercise 2**
Show two different ways to reset the file position pointer to the beginning of the file.

Answer

`rewind(fp);` and `fseek(fp, 0, SEEK_SET);`

*** Exercise 3**
BUG BUSTERS: Is anything wrong with the following?

```
FILE *fp;
int c;
```

```
if ((fp = fopen(oldname, "rb")) == NULL)
    return -1;

while ((c = fgetc(fp)) != EOF)
    fprintf(stdout, "%c", c);

fclose(fp);
```

Answer

You cannot use the EOF check with a binary file. You should use the feof() function instead.

*** Exercise 4**
Write a program that displays a file to the screen.

Answer

Here is one possible answer.

```
/* EXER1604  Day 16 Exercise 4 */
/* Display a file to the screen */

#include <stdio.h>

#define BUFSIZE 100

char buf[BUFSIZE];
char filename[20];
FILE *fp;

int main(void)  {
  puts("Enter name of text file to display: ");
  gets(filename);

  /* Open the file for reading. */

  if ((fp = fopen(filename,"r")) == NULL)  {
    fprintf(stderr, "Error opening file.");
    exit(1);
  }

  /* If end of file not reached, read a line and display it. */

  while (!feof(fp))  {
    fgets(buf, BUFSIZE, fp);
    puts(buf);
  }
  fclose(fp);

  return 0;
}
```

*** Exercise 5**
Write a program that opens a file and prints it to the printer (stdprn). The program should print only 55 lines per page.

Answer

Here is one possible answer.

```
/* EXER1605  Day 16 Exercise 5 */
/* Display a file to the screen. */
/* MS-DOS Application Only */
/* Non-ANSI Application */

#include <stdio.h>

#define BUFSIZE 100

char buf[BUFSIZE];
char filename[20];
FILE *fp;
int lines, reading;

int main(void)  {

  puts("Enter name of text file to display: ");
  gets(filename);

  /* Open the file for reading. */

  if ((fp = fopen(filename,"r")) == NULL)  {
    fprintf(stderr, "Error opening file.");
    exit(1);
  }

  reading = 1;

/* While not at end of page, get a string. */

  while (reading)  {

    for (lines = 0; lines < 55; lines++)  {
      fgets(buf, BUFSIZE, fp);

/* While not at end of file, write the string. */

      if (!(feof(fp)))
        fputs(buf,stdprn);
      else
        reading = 0;
    }

/* At end of page, write a form feed character. */

    fputs("\f",stdprn);
  }
  fclose(fp);

  return 0;
}
```

*** Exercise 6**
Modify the program in exercise five to print headings on each page. The headings should contain the filename and the page number.

Answer

Here is one possible answer.

```
/* EXER1606  Day 16 Exercise 6
 * Display a file to the screen with headers on each
 * page. */

#include <stdio.h>

#define BUFSIZE 100

char buf[BUFSIZE];
char filename[20];
FILE *fp;
int lines, reading, pagenum, newpage;

int main(void)  {
  puts("Enter name of text file to display: ");
  gets(filename);

  /* Open the file for reading. */

  if ((fp = fopen(filename,"r")) == NULL)  {
    fprintf(stderr, "Error opening file.");
    exit(1);
  }

  reading = 1;
  pagenum = 1;
  newpage = 1;

/* While not at end of page, get a string. */

  while (reading)  {

    for (lines = 0; lines < 55; lines++)  {
      if (newpage)  {
  fprintf(stdprn,"FILE: %s  PAGE: %d\n\n\n",
          filename, pagenum++);
  newpage = 0;
      }
      fgets(buf, BUFSIZE, fp);

/* While not at end of file, write the string. */

      if (!(feof(fp)))
  fputs(buf,stdprn);
      else
  reading = 0;
    }

/* At end of page, write a form feed character. */

    fputs("\f",stdprn);
    newpage = 1;

  }
  fclose(fp);
}
```

*** Exercise 7**

Write a program that opens a file and counts the number of characters. The program should print the number of characters when finished.

Answer

Here is one possible answer.

```
/* EXER1607  Day 16 Exercise 7 */
/* Counts number of characters in file */
/* (including new line character). */

#include <stdio.h>

#define BUFSIZE 100

char c;
char filename[20];
int chars;
FILE *fp;


int main(void)  {
  puts("Enter name of text file to analyze: ");
  gets(filename);

  /* Open the file for reading. */

  if ((fp = fopen(filename,"r")) == NULL)  {
    fprintf(stderr, "Error opening file.");
    exit(1);
  }

  c = '\0';
  for (chars = 0; c != -1; chars++)
    c = fgetc(fp);


  fclose(fp);

  printf("Total number of characters: %d", chars-1);

  return 0;
}
```

*** Exercise 8**

Write a program that opens an existing text file and copies it to a new text file with all lowercase letters changed to uppercase, and all other characters unchanged.

Answer

Here is one possible answer.

```
/* EXER1608  Day 16 Exercise 8 */
/* Counts number of characters in file */
/* (including new line character) */

#include <stdio.h>

char c;
char filename[20];
char newfile[20] = {"outfile.da"};
int chars;
FILE *fp, *fnew;


int main(void)  {
  puts("Enter name of text file to copy: ");
  gets(filename);
```

```
  gets(filename);

  /* Open the file for reading. */

  if ((fp = fopen(filename,"r")) == NULL)  {
    fprintf(stderr, "Error opening old file.");
    exit(1);
  }

  if ((fnew = fopen(newfile,"w")) == NULL)  {
    fprintf(stderr, "Error opening new file.");
    exit(1);
  }

  /* Copy data, converting lowercase letters to uppercase. */

  while (1)  {
    c = fgetc(fp);
    if (!feof(fp))  {
      if (c <= 'z' && c >= 'a')
        c = c - 0x20;
      fputc(c,fnew);
    }
    else
      break;
  }
  fclose(fnew);

  fclose(fp);

  return 0;
}
```

*** Exercise**
Write a program that opens any disk file, reads it in 128 byte blocks, and displays the contents of each block on the screen in both hexadecimal and ASCII formats.

Answer

Here is one possible answer.

```
/* EXER1609  Day 16 Exercise 9 */
/* Display a file to the screen in */
/* hexadecimal and ASCII formats */

#include <stdio.h>

#define BUFSIZE 128

char buf[BUFSIZE];
char filename[20];
FILE *fp;
int x;

int main(void)  {
  puts("Enter name of text file to display: ");
  gets(filename);

  /* Open the file for reading. */

  if ((fp = fopen(filename,"rb")) == NULL)  {
    fprintf(stderr, "Error opening file.");
    exit(1);
  }
```

```
  ,

  /* If end of file not reached, */
  /* read a buffer and display it. */

  while (1)  {
    fgets(buf, BUFSIZE, fp);
    if (!feof(fp))  {
      printf("ASCII version:\n");
      puts(buf);
      printf("\n");

      printf("Hexadecimal version:\n");
      for (x = 0; x < BUFSIZE && buf[x] != '\0'; x++)  {
        printf("%02x",buf[x]);
      }
      printf("\n");
    }
    else
      break;
  }
  fclose(fp);

  return 0;
}
```

*** Exercise 1**
Write a function that opens a new temporary file with a specified mode. All temporary files created by this function should automatically be closed and deleted when the program terminates. (Hint: Use the atexit() library function.)

Answer

Here is one possible answer.

```
/* EXER1604  Day 16 Exercise 10 */
/* Open temporary files and close */
/* them automatically at termination */

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

#define NAMESIZE 12

void gettemp(void);
FILE *fptr;
struct filedata  {
  FILE *fp;
  char *name;
  struct filedata *newptr;
};
struct filedata *structp;
struct filedata *lastptr;
void closeit(void);
void storeptr(FILE *fptr, char *sptr);

int main(void)  {

  lastptr = 0;

  /* Get name of temporary file and open it. */
  gettemp();
  gettemp();
  gettemp();
  return 0;
}
```

```
}

void gettemp()  {
  char *ptr;
  FILE *fp;

  ptr = (char *)malloc(NAMESIZE);
  tmpnam(ptr);

  if ((fp = fopen(ptr, "w")) == NULL)  {
    printf("Error opening file.\n");
    exit(1);
  }
  else
    printf("%s was opened successfully\n", ptr);

  /* Save pointers to file stream and file name. */

  storeptr(fp,ptr);

  /* Register function to close */
  /* and delete file at termination. */

  atexit(closeit);

  return;
}

/* Build linked list to store file pointer and file name. */

void storeptr(FILE *fptr, char *sptr)  {

   struct filedata *oldptr;

   oldptr = lastptr;
   lastptr = malloc(sizeof(struct filedata));
   lastptr->fp = fptr;
   lastptr->name = sptr;
   lastptr->newptr = oldptr;
}

/* Retrieve file pointer and file name from linked list. */
/* Zero pointer indicates first file in list. */
/* Close file and delete it. */

void closeit(void)  {

  if (lastptr != 0)  {

    fclose(lastptr->fp);

    if (remove((lastptr->name)) == 0)
      printf("%s has been erased.\n", (lastptr->name));
    else
      printf("Error deleting %s", (lastptr->name));

    lastptr = lastptr->newptr;
  }
}
```

Topic 2.14: Day 16 Summary

Streams and Disk Files

In this unit, you learned how C programs can use disk files. C treats a disk file like a stream, a sequence of characters, just like the predefined streams you learned about on Day 14, "Working with the Screen, Printer, and Keyboard." A

stream associated with a disk file must be opened before it can be used, and it must be closed after use. A disk file stream can be opened in either text or binary mode.

Writing and Reading File Data
Once a disk file has been opened, you can read data from the file into your program, write data from the program to the file, or both. There are three general types of file I/O: formatted, character, and direct. Each type of I/O is best used for certain types of data storage and retrieval tasks.

Sequential Versus Random File Access
Each open disk file has a file position indicator associated with it. This indicator specifies the position in the file, measured as the number of bytes from the start of the file, where subsequent read and write operations occur. With some types of file access, the position indicator is updated automatically, and you don't have to be concerned with it. For random file access, the C standard library provides functions for manipulating the position indicator.

File Management Functions
Finally, C provides some rudimentary file management functions, enabling you to delete and rename disk files. In this unit, you developed your own function for copying a file.

Unit 3. Day 17 Manipulating Strings

Text data, which C stores in strings, is an important part of many programs. So far, you have learned how to input

and output strings and how your program stores them. C offers a variety of functions for other types of string manipulations as well.

Today, you learn

- How to determine the length of a string.
- How to copy and join strings.
- Functions that compare strings.
- How to search strings.
- How to convert strings.
- How to test characters.

Topic 3.1: String Length and Storage

The strlen() Function
You should remember from earlier units that in C programs, a string is a sequence of characters, with its beginning indicated by a pointer and its end marked by the null character \0. At times, you need to know the length of a string (the number of characters between the start and the end of the string). This is done with the library function strlen().

Syntax
Its prototype, in STRING.H, is

```
size_t strlen(char *s);
```

Return Type
You might be puzzling over the `size_t` return type. This is defined in STRING.H as unsigned, so the function strlen() returns an unsigned integer. The `size_t` type is used with many of the string functions. Just remember that it means unsigned.

Argument
The argument passed to strlen is a pointer `s` to the string the length of which you want to know.

Return Value
The function strlen() returns the number of characters between `s` and the next null character, not counting the null character.

Example Program
The program in Listing 17.1 demonstrates strlen().

Listing 17.1: Using the strlen() Function to Determine the Length of a String

```
Code  1: /* LIST1701.c: Day 17 Listing 17.1 */
      2: /* Using the strlen() function. */
      3:
      4: #include <stdio.h>
      5: #include <string.h>
      6:
      7: int main(void) {
      8:   size_t length;
      9:   char buf[80];
     10:
     11:   while (1) {
     12:     puts("\nEnter a line of text;"
     13:          " a blank line terminates.");
     14:     gets(buf);
     15:
     16:     length = strlen(buf);
     17:
```

```
18:     if (length != 0)
19:       printf("\nThat line is %u characters long.",
20:              length);
21:     else
22:       break;
23:   }
24:   return 0;
25: }
```

Output

```
Enter a line of text; a blank line terminates.
Just do it!

That line is 11 characters long.
Enter a line of text; a blank line terminates.
```

Description

This program does little more than demonstrate the use of strlen(). Lines 12 – 14 display a message and get a string called buf. Line 16 uses strlen() to assign the length of buf to the variable length. Line 18 checks whether the string was blank by checking for a length of zero. If the string wasn't blank, line 19 prints the string's size.

Topic 3.2: Copying Strings

Copying Strings

The C library has three functions for copying strings. Because of the way C handles strings, you cannot assign one string to another, as you can in other computer languages. You must copy the source string from its location in memory to the memory location of the destination string. The string-copying functions are strcpy(), strncpy(), and strdup(). All of the string-copying functions require the header file STRING.H.

Topic 3.2.1: The strcpy() Function

Description

The library function strcpy() copies an entire string to another memory location.

Syntax

Its prototype is

```
char *strcpy(char *destination, char *source);
```

Return Value

The function strcpy() copies the string (including the terminating null character \0) pointed to by `source` to the location pointed to by `destination`. The return value is a pointer to the new string, `destination`.

Allocating Space for the Destination String

When using strcpy(), you must allocate storage space for the destination string first. The function has no way of knowing whether `destination` points to allocated space; if space has not been allocated, the function overwrites strlen(source) bytes of memory, starting at `destination`. The use of strcpy() is illustrated in Listing 17.2.

Listing 17.2: Allocating Storage Space for the Destination String Before Using strcpy()

```
Code  1: /* LIST1702.c: Day 17 Listing 17.2 */
      2: /* Demonstrates strcpy(). */
      3:
      4: #include <stdio.h>
      5: #include <string.h>
      6: #include <stdlib.h>
      7:
      8: char source[] = "The source string.";
      9:
     10: int main(void) {
     11:   char dest1[80];
```

```
12:   char *dest2, *dest3;
13:
14:   printf("\nsource: %s", source);
15:
16:   /* Copy to dest1 is okay because dest1 points */
17:   /* to 80 bytes of allocated space. */
18:
19:   strcpy(dest1, source);
20:   printf("\ndest1: %s", dest1);
21:
22:   /* To copy to dest2 you must allocate space. */
23:
24:   dest2 = (char *)malloc(strlen(source) + 1);
25:   strcpy(dest2, source);
26:   printf("\ndest2: %s", dest2);
27:
28:   /* Copying without allocating destination */
29:   /* space is a no-no. The following could */
30:   /* cause serious problems. */
31:
32:   /* strcpy(dest3, source); */
33:   return 0;
34: }
```

Output

```
source: The source string.
dest1:  The source string.
dest2:  The source string.
```

Description

This program demonstrates copying strings to both character arrays such as dest1 (declared on line 11) and character pointers such as dest2 (declared along with dest3 on line 12). Line 14 prints the original source string. This string is then copied to dest1 with strcpy() on line 19. Line 25 copies source to dest2. Both dest1 and dest2 are printed to show that the function was successful. Notice that line 24 allocates the appropriate amount of space for dest2 with the malloc() function. If you copy a string to a character pointer that has not been allocated memory, you get unpredictable results.

Topic 3.2.2: The strncpy() Function

Description

The strncpy() function is similar to strcpy(), except that strncpy() enables you to specify how many characters to copy.

Syntax

Its prototype is

```
char *strncpy(char *destination, char *source,
              size_t num);
```

Arguments

The arguments `destination` and `source` are pointers to the destination and source strings. The function copies, at most, the first `num` characters of source to destination. If `source` is shorter than `num` characters, enough null characters are added at the end of source to make a total of `num` characters copied to destination. If `source` is longer than `num` characters, no terminating \0 is added to `destination`.

Return Value

The function's return value is destination.

The program in Listing 17.3 demonstrates the use of strncpy().

Listing 17.3: The strncpy Function

```
Code  1: /* LIST1703.c: Day 17 Listing 17.3 */
      2: /* Using the strncpy() function. */
```

```
 3:
 4:  #include <stdio.h>
 5:  #include <string.h>
 6:
 7:  char dest[] = ".........................";
 8:  char source[] = "abcdefghijklmnopqrstuvwxyz";
 9:
10:  int main(void) {
11:    size_t n;
12:
13:    while (1) {
14:      puts("Enter the number of characters to "
15:           "copy (1-26)");
16:      scanf("%d", &n);
17:
18:      if (n > 0 && n < 27)
19:        break;
20:    }
21:
22:    printf("\nBefore strncpy destination = %s",
23:            dest);
24:
25:    strncpy(dest, source, n);
26:
27:    printf("\nAfter strncpy destination = %s", dest);
28:    return 0;
29:  }
```

Output

```
Enter the number of characters to copy (1-26)
15

Before strncpy destination = .........................
After strncpy destination = abcdefghijklmno...........
```

Description

This program demonstrates the strncpy() function and an effective way to ensure that only correct information is entered. Lines 13 – 20 contain a while loop that prompts the user for a number from 1 – 26. The loop continues until a valid value is entered. Once a number from 1 – 26 is entered, line 22 prints the value of dest, line 25 copies the number of characters decided by the user, and line 27 prints the result.

Topic 3.2.3: The strdup() Function

Description

The library function strdup() is similar to strcpy(), except that strdup() performs its own memory allocation for the destination string with a call to malloc(). In effect, it does what you did in Listing 17.2, allocating space with malloc() and then calling strcpy().

Syntax

The prototype for strdup() is

```
char *strdup(char *source);
```

Argument

The argument `source` is a pointer to the source string.

Return Value

The function returns a pointer to the destination string—the space allocated by malloc()—or NULL if the needed memory could not be allocated.

Listing 17.4 demonstrates the use of strdup(). Note that strdup() is not an ANSI-standard function. It is included in the Microsoft, Borland, and Symantec C libraries, but might not be present (or might be different) in other C compilers.

Listing 17.4: Using strdup() to Copy a String with Automatic Memory Allocation

Code

```
 1: /* LIST1704.c: Day 17 Listing 17.4 */
 2: /* The strdup() function. */
 3:
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6: #include <string.h>
 7:
 8: char source[] = "The source string.";
 9:
10: int main(void) {
11:   char *dest;
12:
13:   if ((dest = _strdup(source)) == NULL) {
14:     fprintf(stderr, "Error allocating memory.");
15:     exit(1);
16:   }
17:
18:   printf("\nThe destination = %s", dest);
19:   return 0;
20: }
```

Output

```
The destination = The source string.
```

Description

In this listing, strdup() allocates the appropriate memory for dest. It then makes a copy of the passed string, source. Line 18 prints the duplicated string.

Topic 3.3: Concatenating Strings

Concatenating Strings

If you're not familiar with the term *concatenation,* you might be asking yourself, "What is it?" and "Is it legal?" Well, it means to join two strings—to tack one string onto the end of another—and in most states, it is legal. The C standard library contains two string concatenation functions, strcat() and strncat(), both of which require the header file STRING.H.

Topic 3.3.1: The strcat() Function

Syntax

The prototype of strcat() is

```
char *strcat(char *s1, char *s2);
```

Description

The function appends a copy of `s2` onto the end of `s1`, moving the terminating null character to the end of the new string. You must allocate enough space for `s1` to hold the resulting string.

Return Value

The return value of strcat() is a pointer to s1.

The program in Listing 17.5 demonstrates strcat().

Listing 17.5: Using strcat() to Concatenate Strings

Code

```
 1: /* LIST1705.c: Day 17 Listing 17.5 */
 2: /* The strcat() function. */
 3:
 4: #include <stdio.h>
 5: #include <string.h>
 6:
 7: char str1[27] = "a";
```

```
 8: char str2[2];
 9:
10: int main(void) {
11:   int n;
12:
13:   /* Put a null character at the end of str2[]. */
14:
15:   str2[1] = '\0';
16:
17:   for (n = 98; n < 123; n++) {
18:     str2[0] = n;
19:     strcat(str1, str2);
20:     puts(str1);
21:   }
22:   return 0;
23: }
```

Output

```
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

Description

The ASCII codes for the letters *b–z* are 98–122. This program uses these ASCII codes in its demonstration of strcat(). The for loop on lines 17 – 21 assigns these values in turn to str2[0]. Because str2[1] is already the null character (line 15), the effect is to assign the strings "b", "c", and so on to str2. Each of these strings is concatenated with str1 (line 19) and str1 displayed on the screen (line 20).

Topic 3.3.2: The strncat() Function

Description

The library function strncat() also performs string concatenation, but it enables you to specify how many characters of the source string are appended to the end of the destination string.

Syntax

The prototype is

```
char *strncat(char *s1, char *s2, size_t num);
```

Process

If `s2` contains more than num characters, the first num characters are appended to the end of `s1`. If `s2` contains fewer than num characters, all of `s2` is appended to the end of `s1`. In either case, a terminating null character is added. You must allocate enough space for `s1` to hold the resulting string.

Return Value
The function returns a pointer to `s1`.

The program in Listing 17.6 uses strncat() to produce the same output as Listing 17.5.

Listing 17.6: Using the strncat() Function to Concatenate Strings

Code

```
 1: /* LIST1706.c: Day 17 Listing 17.6 */
 2: /* The strncat() function. */
 3:
 4: #include <stdio.h>
 5: #include <string.h>
 6:
 7: char str2[] = "abcdefghijklmnopqrstuvwxyz";
 8:
 9: int main(void) {
10:   char str1[27];
11:   int n;
12:
13:   for (n = 1; n < 27; n++) {
14:     strcpy(str1, "");
15:     strncat(str1, str2, n);
16:     puts(str1);
17:   }
18:   return 0;
19: }
```

Output

```
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

Description
You might wonder about the purpose of line 14, `strcpy(str1, "");`. This line copies an empty string consisting of only a single null character to str1. The result is that the first character in str1, str1[0], is set equal to 0 (the null character). The same thing could have been accomplished with the statement `str1[0] = 0; or str1[0] = '\0';`.

Topic 3.4: Comparing Strings

Comparing Strings
Strings are compared to determine whether they are equal or unequal. If they are unequal, one string is "greater than" or "less than" the other. Determinations of "greater" and "less" are made with the ASCII codes of the characters. In the case of letters, this is equivalent to alphabetical order. The C library contains functions for three types of string

case of letters, this is equivalent to alphabetical order. The C library contains functions for three types of string comparisons: comparing two entire strings, comparing two strings without regard to the case of the characters, and comparing a certain number of characters in two strings. Because these three functions are not ANSI standard, not all compilers treat them the same way.

Topic 3.4.1: Comparing Two Strings

The strcmp() Function
The function strcmp() compares two strings, character by character.

Syntax
Its prototype is

```
int strcmp(char *s1, char *s2);
```

Arguments
The arguments `s1` and `s2` are pointers to the strings being compared.

Return Values
The function's return values are given here.

The program in Listing 17.7 demonstrates strcmp().

Table 17.1: The Values Returned by strcmp()

Return Value	Meaning
< 0	str1 is less than str2.
0	str1 is equal to str2.
> 0	str1 is greater than str2.

Listing 17.7: Using strcmp() to Compare Strings

```
Code  1: /* LIST1707.c: Day 17 Listing 17.7 */
      2: /* The strcmp() function. */
      3:
      4: #include <stdio.h>
      5: #include <string.h>
      6:
      7: int main(void) {
      8:   char str1[80], str2[80];
      9:   int x;
     10:
     11:   while (1) {
     12:
     13:     /* Input two strings. */
     14:     printf("\n\nInput the first string,"
     15:            " a blank to exit: ");
     16:     gets(str1);
     17:
     18:     if (strlen(str1) == 0)
     19:       break;
     20:
     21:     printf("\nInput the second string: ");
     22:     gets(str2);
     23:
     24:     /* Compare them and display the result. */

     25:
     26:     x = strcmp(str1, str2);
     27:
     28:     printf("\nstrcmp(%s,%s) returns %d",
```

```
     29:           str1, str2, x);
     30:   }
     31:   return 0;
     32: }
```

Output

```
Input the first string, a blank to exit: First string

Input the second string: Second string

strcmp(First string,Second string) returns -13

Input the first string, a blank to exit: test string

Input the second string: test string

strcmp(test string,test string) returns 0

Input the first string, a blank to exit: zebra

Input the second string: aardvark

strcmp(zebra,aardvark) returns 25

Input the first string, a blank to exit:
```

Description

The program in Listing 17.7 demonstrates strcmp(), prompting the user for two strings (lines 14 – 16, 21, and 22) and displaying the result returned by strcmp() on lines 28 – 29. Experiment with this program to get a feel for how strcmp() compares strings. Try entering two strings that are identical except for case, such as "Smith" and "SMITH". You learn that strcmp() is case-sensitive, meaning that the program considers upper- and lowercase letters different.

Topic 3.4.2: Comparing Two Strings — Ignoring Case

Compiler-Specific Functions
String comparison functions that are non-case-sensitive operate in the same format as the case-sensitive function strcmp(). Most compilers have their own non-case-sensitive comparison function. Click the Example link.

Example

Example Program
Modify line 27 in Listing 17.7 to use the appropriate compare function for your compiler.

Listing 17.7: Using strcmp() to Compare Strings

```
Code  1: /* LIST1707.c: Day 17 Listing 17.7 */
      2: /* The strcmp() function. */
      3:
      4: #include <stdio.h>
      5: #include <string.h>
      6:
      7: int main(void) {
      8:   char str1[80], str2[80];
      9:   int x;
     10:
     11:   while (1) {
     12:
     13:     /* Input two strings. */
     14:     printf("\n\nInput the first string,"
     15:            " a blank to exit: ");
     16:     gets(str1);
     17:
     18:     if (strlen(str1) == 0)
     19:       break;
```

```
19:       break;
20:
21:     printf("\nInput the second string: ");
22:     gets(str2);
23:
24:     /* Compare them and display the result. */
25:
26:     x = strcmp(str1, str2);
27:
28:     printf("\nstrcmp(%s,%s) returns %d",
29:            str1, str2, x);
30:   }
31:   return 0;
32: }
```

Output

```
Input the first string, a blank to exit: First string

Input the second string: Second string

strcmp(First string,Second string) returns -13

Input the first string, a blank to exit: test string

Input the second string: test string

strcmp(test string,test string) returns 0

Input the first string, a blank to exit: zebra

Input the second string: aardvark

strcmp(zebra,aardvark) returns 25

Input the first string, a blank to exit:
```

Description

The program in Listing 17.7 demonstrates strcmp(), prompting the user for two strings (lines 14 – 16, 21, and 22) and displaying the result returned by strcmp() on lines 28 – 29. Experiment with this program to get a feel for how strcmp() compares strings. Try entering two strings that are identical except for case, such as "Smith" and "SMITH". You learn that strcmp() is case-sensitive, meaning that the program considers upper- and lowercase letters different.

Symantec uses the library function strcmpl(). Microsoft uses a function called _stricmp(). Borland has two functions, strcmpi() and stricmp(). You need to check your library reference manual to determine which function is appropriate for your compiler. When you use a non-case-sensitive function, the strings "Smith" and "SMITH" compare as equal.

Topic 3.4.3: Comparing Partial Strings

The strncmp() Function
The library function strncmp() compares a specified number of characters of one string to another string.

Syntax
Its prototype is

```
int strncmp(char *s1, char *s2, size_t num);
```

Arguments and Return Value
The function strncmp() compares `num` characters of `s2` to `s1`. The comparison proceeds until num characters have been compared or the end of s1 has been reached. The method of comparison and return values are the same as for strcmp(). The comparison is case-sensitive.

The comparison is case sensitive.

Listing 17.8 demonstrates strncmp().

Listing 17.8: Comparing Parts of Strings with strncmp()

Code

```
 1: /* LIST1708.c: Day 17 Listing 17.8 */
 2: /* The strncmp() function. */
 3:
 4: #include <stdio.h>
 5: #include <string.h>
 6:
 7: char str1[] = "The first string.";
 8: char str2[] = "The second string.";
 9:
10: int main(void) {
11:   size_t n, x;
12:
13:   puts(str1);
14:   puts(str2);
15:
16:   while (1) {
17:
18:     puts("\n\nEnter number of characters to "
19:          "compare, 0 to exit.");
20:     scanf("%d", &n);
21:
22:     if (n <= 0)
23:       break;
24:
25:     x = strncmp(str1, str2, n);
26:
27:     printf("\nComparing %d characters, strncmp() "
28:            "returns %d.", n, x);
29:   }
30:   return 0;
31: }
```

Output

```
The first string.
The second string.
Enter number of characters to compare, 0 to exit.
3
Comparing 3 characters, strncmp() returns 0.
Enter number of characters to compare, 0 to exit.
6
Comparing 6 characters, strncmp() returns -13.
Enter number of characters to compare, 0 to exit.
0
```

Description

This program compares two strings defined on lines 7 and 8. Lines 13 and 14 print the strings to the screen so that the user can see what they are. The program executes a while loop on lines 16 – 29 so that multiple compares can be done. If the user asks to compare zero characters on lines 18 – 20, the program breaks on line 23; otherwise, a strncmp() executes on line 25 and the result is printed on lines 27 – 28.

Topic 3.4.4: Searching Strings

Searching Strings

The C library contains a number of functions that search strings. Searches determine whether one string occurs within another string and if so, where. There are six string searching functions, all of which require the header file STRING.H.

Topic 3.4.5: The strchr() Function

Description
The strchr() function finds the first occurrence of a specified character in a string.

Syntax
The prototype is

```
char *strchr(char *s, int c);
```

Return Value
The function strchr() searches s until the character c is found or the terminating null character is found. If c is found, a pointer to it is returned. If not, NULL is returned.

Obtaining the Position of the Found Character
When strchr() finds the character, it returns a pointer to that character. Knowing that s is a pointer to the first character in the string, you can obtain the position of the found character by subtracting s from the pointer value returned by strchr().

The program in Listing 17.9 illustrates this. Remember that the first character in a string is at position 0.

Listing 17.9: Using strchr() to Search a String for a Single Character

```
Code  1: /* LIST1709.c: Day 17 Listing 17.9 */
      2: /* Searching for a single character with */
      3: /* strchr(). */
      4:
      5: #include <stdio.h>
      6: #include <string.h>
      7:
      8: int main(void) {
      9:   char *loc, buf[80];
     10:   int ch;
     11:
     12:   /* Input the string and the character. */
     13:
     14:   printf("Enter the string to be searched: ");
     15:   gets(buf);
     16:   printf("Enter the character to search for: ");
     17:   ch = getchar();
     18:
     19:   /* Perform the search. */
     20:
     21:   loc = strchr(buf, ch);
     22:
     23:   if (loc == NULL)
     24:     printf("The character %c was not found.", ch);
     25:   else
     26:     printf("The character %c was found at "
     27:            "position %d.", ch, loc-buf);
     28:   return 0;
     29: }
```

Output

```
Enter the string to be searched: How now Brown Cow?
Enter the character to search for: C
The character C was found at position 14.
```

Description
This program uses strchr() on line 21 to search for a character within a string. strchr() returns a pointer to the location where the character is first found, or NULL if the character is not found. Line 23 checks whether the value of loc is NULL and prints an appropriate message.

Topic 3.4.6: The strrchr() Function

Description

The library function strrchr() is identical to strchr(), except that it searches a string for the last occurrence of a specified character.

Syntax
Its prototype is

```
char *strrchr(char *s, int c);
```

Return Value
The function strrchr() returns a pointer to the last occurrence of c in s and NULL if it finds no match.

To see how this function works, modify line 20 in Listing 17.9 to use strrchr() instead of strchr(). 9.

Listing 17.9: Using strchr() to Search a String for a Single Character

```
Code  1: /* LIST1709.c: Day 17 Listing 17.9 */
      2: /* Searching for a single character with */
      3: /* strchr(). */
      4:
      5: #include <stdio.h>
      6: #include <string.h>
      7:
      8: int main(void) {
      9:   char *loc, buf[80];
     10:   int ch;
     11:
     12:   /* Input the string and the character. */
     13:
     14:   printf("Enter the string to be searched: ");
     15:   gets(buf);
     16:   printf("Enter the character to search for: ");
     17:   ch = getchar();
     18:
     19:   /* Perform the search. */
     20:
     21:   loc = strchr(buf, ch);
     22:
     23:   if (loc == NULL)
     24:     printf("The character %c was not found.", ch);
     25:   else
     26:     printf("The character %c was found at "
     27:            "position %d.", ch, loc-buf);
     28:   return 0;
     29: }
```

Output

```
Enter the string to be searched: How now Brown Cow?
Enter the character to search for: C
The character C was found at position 14.
```

Description
This program uses strchr() on line 21 to search for a character within a string. strchr() returns a pointer to the location where the character is first found, or NULL if the character is not found. Line 23 checks whether the value of loc is NULL and prints an appropriate message.

Topic 3.4.7: The strcspn() Function

Description
The library function strcspn() searches one string for the first occurrence of any of the characters in a second string.

Syntax

Its prototype is

```
size_t strcspn(char *s1, char *s2);
```

Return Value
The function strcspn() starts searching at the first character of s1, looking for any of the characters contained in s2. If it finds a match, it returns the offset from the beginning of s1 where the matching character is located. If it finds no match, strcspn() returns the value of strlen(s1). This indicates that the first match was the null character terminating the string.

The program in Listing 17.10 shows you how to use strcspn().

Listing 17.10: Searching for a Set of Characters with strcspn()

Code

```
 1: /* LIST1710.c: Day 17 Listing 17.10 */
 2: /* Searching with strcspn(). */
 3:
 4: #include <stdio.h>
 5: #include <string.h>
 6:
 7: int main(void) {
 8:   char buf1[80], buf2[80];
 9:   size_t loc;
10:
11:   /* Input the strings. */
12:
13:   printf("Enter the string to be searched: ");
14:   gets(buf1);
15:   printf("Enter the string containing target "
16:          "characters: ");
17:   gets(buf2);
18:
19:   /* Perform the search. */
20:
21:   loc = strcspn(buf1, buf2);
22:
23:   if (loc == strlen(buf1))
24:     printf("No match was found.");
25:   else
26:     printf("The first match was found at "
27:            "position %d.", loc);
28:   return 0;
29: }
```

Output

```
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: Cow
The first match was found at position 1.
```

Description
This listing is similar to Listing 17.9. Instead of searching for the first occurrence of a single character, it searches for the first occurrence of any of the characters entered in the second string. The program calls strcspn() on line 21 with buf1 and buf2. If any of the characters in buf2 are in buf1, strcspn() returns the offset from the beginning of buf1 to the location of the first occurrence. Line 23 checks the return value to determine if it is NULL. If the value is NULL, no characters were found and an appropriate message is displayed on line 24. If a value was found, a message is displayed stating the character's position in the string.

Topic 3.4.8: The strspn() Function

Description
This function is related to the previous one, strcspn(), as the following paragraph explains.

Syntax
Its prototype is

```
size_t strspn(char *s1, char *s2);
```

Return Value

The function strspn() searches s1, comparing it character by character with the characters contained in s2. It returns the position of the first character in s1 that doesn't match a character in s2. In other words, strspn() returns the length of the initial segment of s1 that consists entirely of characters found in s2.

The program in Listing 17.11 demonstrates strspn().

Listing 17.11: Searching for the First Nonmatching Character with strspn()

Code

```
 1: /* LIST1711.c: Day 17 Listing 17.11 */
 2: /* Searching with strspn(). */
 3:
 4: #include <stdio.h>
 5: #include <string.h>
 6:
 7: int main(void) {
 8:   char buf1[80], buf2[80];
 9:   size_t loc;
10:
11:   /* Input the strings. */
12:
13:   printf("Enter the string to be searched: ");
14:   gets(buf1);
15:   printf("Enter the string containing target "
16:          "characters: ");
17:   gets(buf2);
18:
19:   /* Perform the search. */
20:
21:   loc = strspn(buf1, buf2);
22:
23:   if (loc == NULL)
24:     printf("No match was found.");
25:   else
26:     printf("Characters match up to position %d.",
27:            loc - 1);
28:   return 0;
29: }
```

Output

```
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: How now what?
Characters match up to position 7.
```

Description

This program is similar to the previous example, except it calls strspn() instead of strcspn() on line 21. The function returns the offset into buf1 where the first character not in buf2 is found. Lines 23 – 27 evaluate the return value and print an appropriate message.

Topic 3.4.9: The strpbrk() Function

Description

The library function strpbrk() is similar to strcspn(), searching one string for the first occurrence of any character contained in another string. It differs in that it doesn't include the terminating null characters in the search.

Syntax

The function prototype is

```
char *strpbrk(char *s1, char *s2);
```

Return Value

The function strpbrk() returns a pointer to the first character in s1 that matches any of the characters in s2. If it doesn't find a match, the function returns NULL. As previously explained for the function strchr(), you can obtain the offset of the first match in s1 by subtracting the pointer s1 from the pointer returned by strpbrk() (if it isn't NULL, of course). For example, replace strcspn() on line 20 of Listing 17.10 with strpbrk().

Topic 3.4.10: The strstr() Function

Description
The final and perhaps most useful of C's string searching functions is strstr(). This function searches for the first occurrence of one string within another.

Syntax
Its prototype is

```
char *strstr(char *s1, char *s2);
```

Return Value
The function strstr() returns a pointer to the first occurrence of s2 within s1. If it finds no match, the function returns NULL. If the length of s2 is zero, the function returns s1. When strstr() finds a match, you can obtain the offset of s2 within s1 by pointer subtraction, as explained earlier for strchr(). The matching procedure that strstr() uses is case-sensitive.

The program in Listing 17.12 demonstrates how to use strstr().

DO remember that for many of the string functions, there are equivalent functions that allow you to specify a number of characters to manipulate. The functions that allow specification of the number of characters are usually named strnxxx() where xxx is specific to the function.

DON'T forget that C is case-sensitive. "A" and "a" are different.

Listing 17.12: Using strstr() to Search for One String Within Another

Code

```
 1: /* LIST1712.c: Day 17 Listing 17.12 */
 2: /* Searching with strstr(). */
 3:
 4: #include <stdio.h>
 5: #include <string.h>
 6:
 7: int main(void) {
 8:   char *loc, buf1[80], buf2[80];
 9:
10:   /* Input the strings. */
11:
12:   printf("Enter the string to be searched: ");
13:   gets(buf1);
14:   printf("Enter the target string: ");
15:   gets(buf2);
16:
17:   /* Perform the search. */
18:
19:   loc = strstr(buf1, buf2);
20:
21:   if (loc == NULL)
22:     printf("No match was found.");
23:   else
24:     printf("%s was found at position %d.",
25:            buf2, loc - buf1);
26:   return 0;
27: }
```

Output

```
Enter the string to be searched: How now brown Cow?
Enter the target string: Cow

Cow was found at position 14.
```

Description
Here you are presented with an alternative way to search a string. This time you can search for a string within a string. Lines 12 – 15 prompt for two strings. Line 19 uses strstr() to search for the second string, buf2, within the first string, buf1. A pointer to the first occurrence is

	returned, or NULL if the string is not found. Lines 21 – 25 evaluate the returned value, loc, and print an appropriate message.

Topic 3.5: String Conversions

Case Converting Functions
Many C libraries contain two functions that change the case of characters within a string. These functions are not ANSI standard and therefore might differ or not even exist in your compiler.

Syntax
Their prototypes, in STRING.H, should be similar to

```
char *strlwr(char *s);
char *strupr(char *s);
```

Description
The function strlwr() converts all the letter characters in s from upper- to lowercase; strupr() does the reverse, converting all the characters in s to uppercase. Nonletter characters are not affected. Both functions return s. Note that neither function actually creates a new string but modifies the existing string in place.

Non-ANSI
These functions are a part of the Symantec C and Borland C libraries. In Microsoft C, the functions are preceded by an underscore (_strlwr() and _strupr()). You need to check the Library Reference for your compiler before using these functions. If portability is a concern, you should avoid using non-ANSI functions such as these.

The program in Listing 17.13 demonstrates these functions.

Listing 17.13: Converting the Case of Characters in a String with strlwr() and strupr()

Code

```
 1: /* LIST1713.c: Day 17 Listing 17.13 */
 2: /* The character conversion functions strlwr() */
 3: /* and strupr(). */
 4:
 5: #include <stdio.h>
 6: #include <string.h>
 7:
 8: int main(void) {
 9:   char buf[80];
10:
11:   while (1) {
12:     puts("Enter a line of text, a blank to exit.");
13:     gets(buf);
14:
15:     if (strlen(buf) == 0)
16:       break;
17:
18:     puts(_strlwr(buf));
19:     puts(_strupr(buf));
20:   }
21:   return 0;
22: }
```

Output

```
Enter a line of text, a blank to exit.
Bradley L. Jones
bradley l. jones
BRADLEY L. JONES
Enter a line of text, a blank to exit.
```

Description

This listing prompts for a string on line 12. It then checks to ensure that the string is not blank (line 15). Line 18 prints the string after converting it to lowercase. Line 19 prints the string in all uppercase.

Topic 3.6: Miscellaneous String Functions

Miscellaneous String Functions
This section covers a few string functions that don't fall into any other category. They all require the header file STRING.H.

Topic 3.6.1: TThe strrev() Function

Description
The function strrev() reverses the order of all the characters in a string. strrev() is not part of the ANSI C standard library.

Syntax
Its prototype is

```
char *strrev(char *s);
```

Return Value
The order of all characters in s is reversed, with the terminating null character remaining at the end. The function returns s.

Topic 3.6.2: The strset() and strnset() Functions

Changing Characters
Like the previous function, strrev(), strset(), and strnset() are not part of the ANSI C standard library. These functions change all characters (strset()) or a specified number of characters (strnset()) in a string to a specified character.

Syntax
The prototypes are

```
char *strset(char *s, int c);
char *strnset(char *s, int c, size_t num);
```

Description
The function strset() changes all the characters in s to c except the terminating null character. The function strnset() changes the first num characters of s to c. If num >= strlen(s), strnset() changes all the characters in s.

Non-ANSI
Although these functions are not ANSI standard, they are part of both the Symantec C and Borland C compilers. Microsoft supports these functions with an underscore preceding them (_strrev(), _strnset(), and _strnset()). You should check your Library Reference manual to determine whether your compiler supports these functions.

Listing 17.14 demonstrates both functions.

Listing 17.14: A Demonstration of strrev(), strnset(), and strset()

```
Code  1: /* LIST1714.c: Day 17 Listing 17.14 */
      2: /* strrev(), strset(), and strnset(). */
      3:
      4: #include <stdio.h>
      5: #include <string.h>
      6:
      7: char str[] = "This is the test string.";
      8:
      9: int main(void) {
     10:   printf("\nThe original string: %s", str);
```

```
11:   printf("\nCalling strrev(): %s", strrev(str));
12:   printf("\nCalling strrev() again: %s",
13:          strrev(str));
14:   printf("\nCalling strnset(): %s",
15:          strnset(str, '!', 5));
16:   printf("\nCalling strset(): %s",
17:          strset(str, '!'));
18:
19:   return 0;
20: }
```

Output

```
The original string: This is the test string.
Calling strrev(): .gnirts tset eht si sihT
Calling strrev() again: This is the test string.
Calling strnset(): !!!!!is the test string.
Calling strset(): !!!!!!!!!!!!!!!!!!!!!!!!
```

Description

This program demonstrates the three different string functions. The demonstrations are done by printing the value of a string, str. Line 10 prints the string normally. Line 11 prints the string after it has been reversed with strrev(). Lines 12 – 13 reverse it back to its original state. Lines 14 – 15 use the strnset() function to set the first five characters of str to exclamation marks. To finish the program, lines 16 – 17 change the entire string to exclamation marks.

Topic 3.7: String-to-Number Conversions

String-to-Number Conversions
There are times when you need to convert the string representation of a number to an actual numeric variable. For example, the string "123" can be converted to a type int variable with the value 123. There are three functions that convert a string to a number, explained in the following paragraphs, and their prototypes are in STDLIB.H.

Topic 3.7.1: The atoi() Function

Description
The library function atoi() converts a string to an integer.

Syntax
The prototype is

```
int atoi(char *ptr);
```

Return Value
The function atoi() converts the string pointed to by ptr to an integer. Besides digits, the string can contain leading whitespace and a + or – sign. Conversion starts at the beginning of the string and proceeds until a nonconvertible character (for example, a letter or punctuation mark) is encountered. The resulting integer is returned to the calling program. If it finds no convertible characters, atoi() returns zero.

Click here for a list of examples.

Table 17.2: String-to-Number Conversions with atoi()

String	Value Returned	Description
"157"	157	This example is straightforward.
"-1.6"	-1	You might be confused as to why the ".6" did not translate. Remember, this is a string-to-integer conversion.
"+50x"	50	The example is also straightforward; the function "understands" the plus sign and considers it a part of the number.

		consiaers it a part of the number.
"twelve"	0	The atoi() function cannot translate words; all it "sees" are characters. Because the string did not start with a number, atoi() returns 0.
"x506"	0	Because the string did not start with a number, atoi() returns 0.

Topic 3.7.2: The atol() Function

Description
The library function atol() works exactly like atoi(), except that it returns a type long.

Syntax
The function prototype is

```
long atol(char *ptr);
```

Return Value
The values returned by atol() would be the same as shown for atoi() in Table 17.2, except that each return value would be a type long instead of a type int.

Topic 3.7.3: The atof() Function

Description
The function atof() converts a string to a type double.

Syntax
The prototype is

```
double atof(char *s);
```

Arguments and Return Value
The argument s points to the string to be converted. This string can contain leading whitespace and a + or – character. The number can contain the digits 0–9, the decimal point, and the exponent indicator E or e. If there are no convertible characters, atof() returns zero.

Click here for a list of examples of using atof().

The program in Listing 17.15 enables you to enter your own strings for conversion.

Table 17.3: String-to-Number Conversions with atof()	
String	**Value Returned by atof()**
"12"	12.000000
"-0.123"	-0.123000
"123E+3"	123000.000000
"123.1e-5"	0.001231

Listing 17.15: Using atof() to Convert Strings to Type double Numeric Variables

```
Code  1: /* LIST1715.c: Day 17 Listing 17.15 */
      2: /* Demonstration of atof(). */
      3:
      4: #include <stdio.h>
      5: #include <stdlib.h>
```

```
 6: #include <string.h>
 7:
 8: int main(void) {
 9:   char buf[80];
10:   double d;
11:
12:   while (1) {
13:     printf("\nEnter the string to convert, "
14:            "blank to exit: ");
15:     gets(buf);
16:
17:     if (strlen(buf) == 0)
18:       break;
19:
20:     d = atof(buf);
21:
22:     printf("The converted value is %f.", d);
23:   }
24:   return 0;
25: }
```

Output

```
Enter the string to convert (blank to exit): 1009.12
The converted value is 1009.120000.
Enter the string to convert (blank to exit): abc
The converted value is 0.000000.
Enter the string to convert (blank to exit): 3
The converted value is 3.000000.
Enter the string to convert (blank to exit):
```

Description

This listing uses a while loop on lines 12 – 23 to enable you to keep running the program until you enter a blank line. Lines 13 – 15 prompt for the value. Line 17 checks whether a blank line was entered. If it was, the program breaks out of the while loop and ends. Line 20 calls atof(), converting the value entered, buf, to a type double, d. Line 22 prints the final result.

Topic 3.8: Character Test Functions

The Header File CTYPE.H

The header file CTYPE.H contains the prototypes for a number of functions that test characters, returning TRUE or FALSE depending on whether the character meets a certain condition. For example, is it a letter or is it a numeral? The isxxxx() functions are actually macros, defined in CTYPE.H. You learn about macros on Day 21, "Taking Advantage of Preprocessor Directives and More," at which time you might want to look at the definitions in CTYPE.H to see how they work. For now, you only need to see how they're used.

Syntax

The isxxxx() macros all have the same prototype.

```
int isxxxx(int c);
```

where c is the character being tested.

Return Value

The return value is TRUE (nonzero) if the condition is met or FALSE (zero) if it is not.

The isxxxx() Macros

Table 17.4 lists the complete set of isxxxx() macros.

You can do many interesting things with the character-test macros. One example is the function get_int() in Listing 17.16. This function inputs an integer from stdin and returns it as a type int variable. The function skips over leading whitespace and returns 0 if the first nonspace character is not a numeric character.

DON'T use non-ANSI functions if you plan on porting your application to other platforms.

DO take advantage of the string functions that are available.

DON'T confuse characters with numbers. It is easy to forget that '1' is not the same thing as 1.

Table 17.4: The isxxxx() Macros

Macro	Action
isalnum()	Returns TRUE if ch is a letter or a digit.
isalpha()	Returns TRUE if ch is a letter.
isascii()	Returns TRUE if ch is a standard ASCII character (between 0 and 127).
iscntrl()	Returns TRUE if ch is a control character.
isdigit()	Returns TRUE if ch is a digit.
isgraph()	Returns TRUE if ch is a printing character (other than a space).
islower()	Returns TRUE if ch is a lowercase letter.
isprint()	Returns TRUE if ch is a printing character (including a space).
ispunct()	Returns TRUE if ch is a punctuation character.
isspace()	Returns TRUE if ch is a whitespace character (space, tab, vertical tab, line feed, form feed, or carriage return).
isupper()	Returns TRUE if ch is an uppercase letter.
isxdigit()	Returns TRUE if ch is a hexadecimal digit (0–9, a–f, A–F).

Listing 17.16: Using the isxxxx() Macros to Implement a Function that Inputs an Integer

```
Code  1: /* LIST1716.c: Day 17 Listing 17.16 */
      2: /* Using character test macros to create an */
      3: /* integer input function. */
      4:
      5: #include <stdio.h>
      6: #include <ctype.h>
      7:
      8: int get_int(void);
      9:
     10: int main(void) {
     11:   int x;
     12:   x = get_int();
     13:
     14:   printf("You entered %d.", x);
     15:   return 0;
     16: }
     17:
     18: int get_int(void) {
     19:   int ch, i, sign = 1;
     20:
     21:   /* Skip over any leading whitespace. */
     22:
     23:   while (isspace(ch = getchar()))
     24:     ;
     25:
     26:   /* If the first character is non-numeric, */
     27:   /* unget the character and return 0. */
     28:
     29:   if (ch != '-' && ch != '+' && !isdigit(ch)
     30:       && ch != EOF) {
     31:     ungetc(ch, stdin);
     32:     return 0;
```

```
33:   }
34:
35:   /* If the first character is a minus sign, set */
36:   /* sign accordingly. */
37:
38:   if (ch == '-')
39:     sign = -1;
40:
41:   /* If the first character is a plus or minus */
42:   /* sign, get the next character. */
43:
44:   if (ch == '+' || ch == '-')
45:     ch = getchar();
46:
47:   /* Read characters until a nondigit is input. */
48:   /* Assign values, multiplied by proper power  */
49:   /* of 10, to i. */
50:
51:   for (i = 0; isdigit(ch); ch = getchar())
52:     i = 10 * i + (ch - '0');
53:
54:   /* Make result negative if sign is negative. */
55:
56:   i *= sign;
57:
58:   /* If EOF was not encountered, a nondigit */
59:   /* character must have been read in, so */
60:   /* unget it. */
61:
62:   if (ch != EOF)
63:     ungetc(ch, stdin);
64:
65:   /* Return the input value. */
66:
67:   return i;
68: }
```

Output

```
>list1716
-100
You entered -100.
>list1716
abc3.145
You entered 0.
>list1716
9 9 9
You entered 9.
>list1716
   2.5
You entered 2.
```

Description

This program uses the library function ungetc() on lines 31 and 63, which you learned about on Day 14, "Working with the Screen, Printer, and Keyboard." Remember that this function "ungets," or returns, a character to the specified stream. This "returned" character is the first one input the next time the program reads a character from that stream. This is necessary in case the function get_int() reads a nonnumeric character from stdin because you want to put that character back in case the program needs it later.

In this program, main() is simple. An integer variable, x, is declared (line 11), assigned the value of the get_int() function (line 12), and printed to the screen (line 14). The get_int() function makes up the rest of the program.

The get_int() function is not so simple. To remove leading whitespace that might be entered, line 23 loops with a while command. The isspace() macro tests a character, ch, gotten with the getchar() function. If ch is a space, another character is gotten, until a non-whitespace

character is received. Lines 29 – 30 check whether the character is one that can be used. Lines 29 – 30 could be read, "If the character gotten is not a negative sign, a plus sign, a digit, or the end of the file[s]." If this is true, ungetc() is used on line 31 to put the character back, and the function returns to main(). If the character is usable, execution continues.

	Lines 38 – 45 handle the sign of the number. Line 38 checks to see whether the character entered was a negative sign. If it was, a variable, sign, is set to –1. sign is used to make the final number either positive or negative (line 56). Because positive numbers are the default, once you have taken care of the negative sign, you are almost ready to continue. If a sign was entered, the program needs to get another character. Lines 44 and 45 take care of this. The heart of the function is the for loop on lines 51 – 52 that continues to get characters as long as the characters gotten are digits. Line 52 might be a little confusing at first. This line takes the individual character entered and turns it into a number. Subtracting the character '0' from your number changes a character number to a real number. (Remember the ASCII values.) Once the correct numerical value is obtained, the numbers are multiplied by the proper power of 10. The for loop continues until a nondigit number is entered. At that point, line 56 applies the sign to the number, making it complete. Before returning, the program needs to do a little cleanup. If the last number was not the end of file, it needs to be put back in case it is needed elsewhere. Line 63 does this before line 67 returns.

Topic 3.9: Day 17 Q&A

Questions & Answers
Here are some questions to help you review what you have learned in this unit.

Question 1
How do I know whether a function is ANSI compatible?

Answer
Most compilers have a Library Function Reference manual or section. This manual or section of a manual lists all the compiler's library functions and how to use them. Usually, the manual includes information on the compatibility of the function. Sometimes the descriptions state not only whether the function is ANSI compatible, but also whether it is compatible with DOS, UNIX, Windows, C++, or OS/2. (Most compilers only tell you what is relevant to their compiler.)

Question 2
Are all of the available string functions presented in this unit?

Answer
No. However, the string functions presented in this unit should cover virtually all of your needs. Consult your compiler's Library Reference to see what other functions are available.

Question 3
Does strcat() ignore trailing spaces when doing a concatenation?

Answer
No. strcat() looks at a space as just another character.

Question 4
Can I convert numbers to strings?

Answer
Yes. You can write a function similar to the one in Listing 17.16, or you can check your Library Reference for available functions. Some functions available include itoa(), ltoa(), and ultoa().

Topic 3.10: Day 17 Think About Its

Think About Its
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

The isxxxx() Macros	
Macro	**Action**
isalnum()	Returns TRUE if ch is a letter or a digit.
isalpha()	Returns TRUE if ch is a letter.
isascii()	Returns TRUE if ch is a standard ASCII character (between 0 and 127).
iscntrl()	Returns TRUE if ch is a control character.
isdigit()	Returns TRUE if ch is a digit.
isgraph()	Returns TRUE if ch is a printing character (other than a space).
islower()	Returns TRUE if ch is a lowercase letter.
isprint()	Returns TRUE if ch is a printing character (including a space).
ispunct()	Returns TRUE if ch is a punctuation character.
isspace()	Returns TRUE if ch is a whitespace character (space, tab, vertical tab, line feed, form feed, or carriage return).
isupper()	Returns TRUE if ch is an uppercase letter.
isxdigit()	Returns TRUE if ch is a hexadecimal digit (0–9, a–f, A–F).

The isxxxx() Macros	
Macro	**Action**
isalnum()	Returns TRUE if ch is a letter or a digit.
isalpha()	Returns TRUE if ch is a letter.
isascii()	Returns TRUE if ch is a standard ASCII character (between 0 and 127).
iscntrl()	Returns TRUE if ch is a control character.
isdigit()	Returns TRUE if ch is a digit.
isgraph()	Returns TRUE if ch is a printing character (other than a space).
islower()	Returns TRUE if ch is a lowercase letter.
isprint()	Returns TRUE if ch is a printing character (including a space).
ispunct()	Returns TRUE if ch is a punctuation character.
isspace()	Returns TRUE if ch is a whitespace character (space, tab, vertical tab, line feed, form feed, or carriage return).
isupper()	Returns TRUE if ch is an uppercase letter.
isxdigit()	Returns TRUE if ch is a hexadecimal digit (0–9, a–f, A–F).

Try Its
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
What values do the test ("isaxxxx()") functions return?

Answer

TRUE (1) or FALSE (0)

*** Exercise 2**
BUG BUSTERS: Is anything wrong with the following?

```
char *string1, string2;
string1 = "Hello World";
strcpy( string2, string1);
printf( "%s %s", string1, string2 );
```

Answer

string2 was not allocated space before it was used. There is no way of knowing where strcpy() copies the value of string1.

*** Exercise 3**
Write a program that prompts for the user's last name, first name, and middle name individually. Then store the name in a new string as first initial, period, space, middle initial, period, space, last name. For example, if "Bradley," "Lee," and "Jones" were entered, store "B. L. Jones." Display the new name to the screen.

Answer

```
/* EXER1703.C Day 17 Exercise 3 */
/* Redisplays user's name using initials for the first */
/* two names */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char buff[87];
char lname[80];
char fname[80];
char mname[80];

int main(void)  {

  printf("\nEnter your last name: ");
  gets(lname);

  printf("\nEnter your first name: ");
  gets(fname);

  printf("\nEnter your middle name: ");
  gets(mname);

  strncpy(buff,fname,1);
  strcat(buff,". ");
  if (mname[0] != '\0')  {
    strncat(buff,mname,1);
    strcat(buff,". ");
  }
  strcat(buff,lname);

  printf("\n%s", buff);
```

```
  return 0;
}
```

*** Exercise 4**
Write a program to prove the answers to these questions:

Using this table, which macros would return TRUE for var? `int var = 1;`

isascii() and iscntrl() both return TRUE; all others return FALSE.

Using Table 17.4, which macros would return TRUE for x? `char x = 65;`

65 is equivalent to the ASCII character 'A'. The following macros return TRUE: isalnum(), isalpha(), isascii(), isgraph(), isprint(), and isupper().

Answer

Table 17.4: The isxxxx() Macros	
Macro	**Action**
isalnum()	Returns TRUE if ch is a letter or a digit.
isalpha()	Returns TRUE if ch is a letter.
isascii()	Returns TRUE if ch is a standard ASCII character (between 0 and 127).
iscntrl()	Returns TRUE if ch is a control character.
isdigit()	Returns TRUE if ch is a digit.
isgraph()	Returns TRUE if ch is a printing character (other than a space).
islower()	Returns TRUE if ch is a lowercase letter.
isprint()	Returns TRUE if ch is a printing character (including a space).
ispunct()	Returns TRUE if ch is a punctuation character.
isspace()	Returns TRUE if ch is a whitespace character (space, tab, vertical tab, line feed, form feed, or carriage return).
isupper()	Returns TRUE if ch is an uppercase letter.
isxdigit()	Returns TRUE if ch is a hexadecimal digit (0–9, a–f, A–F).

```
/* EXER1704.C Day 17 Exercise 4 */
/* Verify  answers to questions */
#include <stdio.h>
#include <ctype.h>

int var = 1;
char x = 65;

int main(void)  {

  printf("ISALNUM for var is %s\n",
         isalnum(var) ? "True" : "False");
  printf("ISALNUM for x is %s\n",
         isalnum(x) ? "True" : "False");

  printf("ISALPHA for var is %s\n",
         isalpha(var) ? "True" : "False");
```

```
  printf("ISALPHA for x is %s\n",
         isalpha(x) ? "True" : "False");

  printf("ISASCII for var is %s\n",
         isascii(var) ? "True" : "False");
  printf("ISASCII for x is %s\n",
         isascii(x) ? "True" : "False");

  printf("ISCNTRL for var is %s\n",
         iscntrl(var) ? "True" : "False");
  printf("ISCNTRL for x is %s\n",
         iscntrl(x) ? "True" : "False");

  printf("ISDIGIT for var is %s\n",
         isdigit(var) ? "True" : "False");
  printf("ISDIGIT for x is %s\n",
         isdigit(x) ? "True" : "False");

  printf("ISGRAPH for var is %s\n",
         isgraph(var) ? "True" : "False");
  printf("ISGRAPH for x is %s\n",
         isgraph(x) ? "True" : "False");

  printf("ISLOWER for var is %s\n",
         islower(var) ? "True" : "False");
  printf("ISLOWER for x is %s\n",
         islower(x) ? "True" : "False");

  printf("ISPRINT for var is %s\n",
         isprint(var) ? "True" : "False");
  printf("ISPRINT for x is %s\n",
         isprint(x) ? "True" : "False");

  printf("ISPUNCT for var is %s\n",
         ispunct(var) ? "True" : "False");
  printf("ISPUNCT for x is %s\n",
         ispunct(x) ? "True" : "False");

  printf("ISSPACE for var is %s\n",
         isspace(var) ? "True" : "False");
  printf("ISSPACE for x is %s\n",
         isspace(x) ? "True" : "False");

  printf("ISUPPER for var is %s\n",
         isupper(var) ? "True" : "False");
  printf("ISUPPER for x is %s\n",
         isupper(x) ? "True" : "False");

  printf("ISXDIGIT for var is %s\n",
         isxdigit(var) ? "True" : "False");
  printf("ISXDIGIT for x is %s\n",
         isxdigit(x) ? "True" : "False");

  return 0;
}
```

*** Exercise 5**

The function strstr() finds the first occurrence of one string within another, and it is case-sensitive. Write a function that performs the same task without case-sensitivity.

Answer

```
/* EXER1705.C Day 17 Exercise 5 */
/* Find 1st occurrence of one string */
```

```
/* within another, without regard to case. */

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

int strstr2(char *str1, char *str2);
int x;

int main(void)  {

  x = strstr2("hij","ABCDEFGHIJKLMNOPQRSTUVWXYZ");

  printf("Match was found at location %d\n", x);

  return 0;
}

int strstr2(char *s1, char *s2)  {

  char *s3, *s4, *ptr;
  int x;

  /* Change s2 to uppercase. */
  if ((s4 = (char *)malloc(strlen(s2))) == NULL)  {
    printf("Insufficient memory");
    exit(1);
  }
  strcpy(s4,s2);
  for (x = 0; x < strlen(s2); x++)
    if (s2[x] >= 'a' && s2[x] <= 'z')
      s4[x] = s2[x] - 0x20;
    else
      s4[x] = s2[x];

  /* Change s1 to uppercase. */
  if ((s3 = (char *)malloc(strlen(s1))) == NULL)  {
    printf("Insufficient memory");
    exit(1);
  }
  strcpy(s3,s1);
  for (x = 0; x < strlen(s1); x++)
    if (s1[x] >= 'a' && s1[x] <= 'z')
      s3[x] = s1[x] - 0x20;
    else
      s3[x] = s1[x];

  /* Check for string. */
  ptr = strstr(s4,s3);
  return (ptr - s4);
}



/* EXER1704.C Day 17 Exercise 4 */
/* Verify  answers to questions */
#include <stdio.h>
#include <ctype.h>

int var = 1;
char x = 65;


int main(void)  {

  printf("ISALNUM for var is %s\n",
          isalnum(var) ? "True" : "False");
  printf("ISALNUM for x is %s\n",
```

```
            isalnum(x) ? "True" : "False");

    printf("ISALPHA for var is %s\n",
            isalpha(var) ? "True" : "False");
    printf("ISALPHA for x is %s\n",
            isalpha(x) ? "True" : "False");

    printf("ISASCII for var is %s\n",
            isascii(var) ? "True" : "False");
    printf("ISASCII for x is %s\n",
            isascii(x) ? "True" : "False");

    printf("ISCNTRL for var is %s\n",
            iscntrl(var) ? "True" : "False");
    printf("ISCNTRL for x is %s\n",
            iscntrl(x) ? "True" : "False");

    printf("ISDIGIT for var is %s\n",
            isdigit(var) ? "True" : "False");
    printf("ISDIGIT for x is %s\n",
            isdigit(x) ? "True" : "False");

    printf("ISGRAPH for var is %s\n",
            isgraph(var) ? "True" : "False");
    printf("ISGRAPH for x is %s\n",
            isgraph(x) ? "True" : "False");

    printf("ISLOWER for var is %s\n",
            islower(var) ? "True" : "False");
    printf("ISLOWER for x is %s\n",
            islower(x) ? "True" : "False");

    printf("ISPRINT for var is %s\n",
            isprint(var) ? "True" : "False");
    printf("ISPRINT for x is %s\n",
            isprint(x) ? "True" : "False");

    printf("ISPUNCT for var is %s\n",
            ispunct(var) ? "True" : "False");
    printf("ISPUNCT for x is %s\n",
            ispunct(x) ? "True" : "False");

    printf("ISSPACE for var is %s\n",
            isspace(var) ? "True" : "False");
    printf("ISSPACE for x is %s\n",
            isspace(x) ? "True" : "False");

    printf("ISUPPER for var is %s\n",
            isupper(var) ? "True" : "False");
    printf("ISUPPER for x is %s\n",
            isupper(x) ? "True" : "False");

    printf("ISXDIGIT for var is %s\n",
            isxdigit(var) ? "True" : "False");
    printf("ISXDIGIT for x is %s\n",
            isxdigit(x) ? "True" : "False");

    return 0;
}
```

*** Exercise 6**

Write a function that determines the number of times one string occurs within another.

Answer

```
/* EXER1706.C Day 17 Exercise 6 */
/* Find the number of times one string occurs within */
/* another. */

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

char s2[] = { "abcabcabcabcabc" };
char s1[] = { "abc" };
int count;
int occur(char *s1, char *s2);

int main(void)  {

  count = occur(s1,s2);
  printf("Number of occurrences are %d\n", count);

  return 0;
}

int occur(char *s1, char *s2)  {
  int count, x;
  char *ptr;

  count = 0;
  ptr = s2;
  for (x = 0; *ptr != '\0'; x++)  {

    if ((ptr = strstr(ptr, s1)) != NULL)  {

      count++;
      ptr++;

    }
    else
      break;
  }

  return count;
}
```

*** Exercise 7**

Write a program that searches a text file for occurrences of a user-specified target string, and then reports the line numbers where the target is found. For example, if you search one of your C source code files for the string "printf()", the program should list all the lines where the printf() function is called by the program.

Answer

```
/* EXER1707  Day 17 Exercise 7 */
/* Display occurrences of a specified string. */

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

#define BUFSIZE 100

char buf[BUFSIZE];
char string[80];
char filename[20];
FILE *fp;
int count;
```

```
int main(void)  {
  puts("Enter name of text file to search: ");
  gets(filename);

  /* Open the file for reading. */

  if ((fp = fopen(filename,"r")) == NULL)  {
    fprintf(stderr, "Error opening file.");
    exit(1);
  }

  printf("\nEnter string to be found: ");
  gets(string);

  count = 1;

  while(1)  {
    if (fgets(buf,BUFSIZE,fp) != NULL)  {
      if (strstr(buf,string) != NULL)
  printf("%d  %s\n", count, buf);
      count++;
    }
    else
      break;
  }

  fclose(fp);
  return 0;
}
```

*** Exercise 8**

Listing 17.16 demonstrates a function that inputs an integer from stdin. Write a function get_float() that inputs a floating-point value from stdin.

Answer

Listing 17.16: Using the isxxxx() Macros to Implement a Function that Inputs an Integer

```
Code   1: /* LIST1716.c: Day 17 Listing 17.16 */
       2: /* Using character test macros to create an */
       3: /* integer input function. */
       4:
       5: #include <stdio.h>
       6: #include <ctype.h>
       7:
       8: int get_int(void);
       9:
      10: int main(void) {
      11:   int x;
      12:   x = get_int();
      13:
      14:   printf("You entered %d.", x);
      15:   return 0;
      16: }
      17:
      18: int get_int(void) {
      19:   int ch, i, sign = 1;
      20:
      21:   /* Skip over any leading whitespace. */
      22:
      23:   while (isspace(ch = getchar()))
      24:     ;
      25:
      26:   /* If the first character is non-numeric, */
      27:   /* unget the character and return 0. */
      28:
```

```
  if (ch != '-' && ch != '+' && !isdigit(ch)
      && ch != EOF) {
    ungetc(ch, stdin);
    return 0;
  }

  /* If the first character is a minus sign, set */
  /* sign accordingly. */

  if (ch == '-')
    sign = -1;

  /* If the first character is a plus or minus */
  /* sign, get the next character. */

  if (ch == '+' || ch == '-')
    ch = getchar();

  /* Read characters until a nondigit is input. */
  /* Assign values, multiplied by proper power  */
  /* of 10, to i. */

  for (i = 0; isdigit(ch); ch = getchar())
    i = 10 * i + (ch - '0');

  /* Make result negative if sign is negative. */

  i *= sign;

  /* If EOF was not encountered, a nondigit */
  /* character must have been read in, so */
  /* unget it. */

  if (ch != EOF)
    ungetc(ch, stdin);

  /* Return the input value. */

  return i;
}
```

```
/* EXER178.C   Day 17 Exercise 8  */
/* Input a floating point value from stdin. */

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

float x;
char ch;
float get_float(void);

int main(void)  {

  printf("Enter a number: ");

  x = get_float();

  printf("You entered %f.\n", x);

  return 0;


}

float get_float(void)  {
  int ch, i, sign = 1;
```

```
    float f, mult;
    int x;

    /* Skip over any leading whitespace. */

    while (isspace(ch = getchar()))
      ;

    /* If the first character is non-numeric, unget */
    /* the character and return 0. */

    if (ch != '-' && ch != '+' && !isdigit(ch)
                  && ch != '.' && ch != EOF)  {
      ungetc(ch,stdin);
      return 0;
    }

    /* If the first character is a minus sign, */
    /* set sign accordingly */
    if (ch == '-')
      sign = -1;

    /* If the first character was a plus or minus sign, */
    /* get the next character. */

    if (ch == '+' || ch == '-')
      ch = getchar();

    /* Read characters until a nondigit is input. Assign */
    /* values, multiplied by proper power of 10, to i. */

    for (i = 0; isdigit(ch); ch = getchar())
      i = 10 * i + (ch - '0');

    f = 0;
    if (ch == '.')  {
      ch = getchar();
      for (x = 0; isdigit(ch); ch = getchar(), x++)
        f = (f * 10) + (ch - '0');

      for (mult = 10, --x; x > 0; x--)
        mult = mult * 10;
      f = (f / mult) + (float)(i);
    }
    else
      f = (float) i;

    /* Make result negative if sign is negative. */

    f *= sign;

    /* If EOF was not encountered, a nondigit character */
    /* must have been read in, so unget it. */

    if (ch != EOF)
      ungetc(ch,stdin);

    /* Return the input value. */

    return f;
  }
```

String Manipulations

This unit showed various ways you can manipulate strings. Using C standard library functions (and possibly compiler-specific functions as well), you can copy, concatenate, compare, and search strings. These are all needed tasks in most programming projects.

String Conversions
The standard library also contains functions for converting the case of characters in strings and for converting strings to numbers.

Character Testing
Finally, C provides a variety of character-test functions or more accurately, macros that perform a variety of tests on individual characters. By using these macros to test characters, you can create your own custom input functions.

Listing 17.16: Using the isxxxx() Macros to Implement a Function that Inputs an Integer

```
Code  1:  /* LIST1716.c: Day 17 Listing 17.16 */
      2:  /* Using character test macros to create an */
      3:  /* integer input function. */
      4:
      5:  #include <stdio.h>
      6:  #include <ctype.h>
      7:
      8:  int get_int(void);
      9:
     10:  int main(void) {
     11:    int x;
     12:    x = get_int();
     13:
     14:    printf("You entered %d.", x);
     15:    return 0;
     16:  }
     17:
     18:  int get_int(void) {
     19:    int ch, i, sign = 1;
     20:
     21:    /* Skip over any leading whitespace. */
     22:
     23:    while (isspace(ch = getchar()))
     24:      ;
     25:
     26:    /* If the first character is non-numeric, */
     27:    /* unget the character and return 0. */
     28:
     29:    if (ch != '-' && ch != '+' && !isdigit(ch)
     30:        && ch != EOF) {
     31:      ungetc(ch, stdin);
     32:      return 0;
     33:    }
     34:
     35:    /* If the first character is a minus sign, set */
     36:    /* sign accordingly. */
     37:
     38:    if (ch == '-')
     39:      sign = -1;
     40:
     41:    /* If the first character is a plus or minus */
     42:    /* sign, get the next character. */
     43:
     44:    if (ch == '+' || ch == '-')
     45:      ch = getchar();
     46:
     47:    /* Read characters until a nondigit is input. */
     48:    /* Assign values, multiplied by proper power  */
     49:    /* of 10, to i. */
     50:
     51:    for (i = 0; isdigit(ch); ch = getchar())
     52:      i = 10 * i + (ch - '0');
     53:
     54:    /* Make result negative if sign is negative. */
     55:
     56:    i *= sign;
     57:
     58:    /* If EOF was not encountered, a nondigit */
```

```
    59:   /* character must have been read in, so */
    60:   /* unget it. */
    61:
    62:   if (ch != EOF)
    63:     ungetc(ch, stdin);
    64:
    65:   /* Return the input value. */
    66:
    67:   return i;
    68: }
```

```
/* EXER178.C   Day 17 Exercise 8  */
/* Input a floating point value from stdin. */

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

float x;
char ch;
float get_float(void);

int main(void)  {

  printf("Enter a number: ");

  x = get_float();

  printf("You entered %f.\n", x);

  return 0;

}

float get_float(void)  {
  int ch, i, sign = 1;
  float f, mult;
  int x;

  /* Skip over any leading whitespace. */

  while (isspace(ch = getchar()))
    ;

  /* If the first character is non-numeric, unget */
  /* the character and return 0. */

  if (ch != '-' && ch != '+' && !isdigit(ch)
                && ch != '.' && ch != EOF)  {
    ungetc(ch,stdin);
    return 0;
  }

  /* If the first character is a minus sign, */
  /* set sign accordingly */
  if (ch == '-')
    sign = -1;

  /* If the first character was a plus or minus sign, */
  /* get the next character. */


  if (ch == '+' || ch == '-')
    ch = getchar();

  /* Read characters until a nondigit is input. Assign */
  /* values, multiplied by proper power of 10, to i. */
```

```
  /* values, multiplied by proper power of 10, to i. */

  for (i = 0; isdigit(ch); ch = getchar())
    i = 10 * i + (ch - '0');

  f = 0;
  if (ch == '.')  {
    ch = getchar();
    for (x = 0; isdigit(ch); ch = getchar(), x++)
      f = (f * 10) + (ch - '0');

    for (mult = 10, --x; x > 0; x--)
      mult = mult * 10;
    f = (f / mult) + (float)(i);
  }
  else
    f = (float) i;

  /* Make result negative if sign is negative. */

  f *= sign;

  /* If EOF was not encountered, a nondigit character */
  /* must have been read in, so unget it. */

  if (ch != EOF)
    ungetc(ch,stdin);

  /* Return the input value. */

  return f;
}
```

String Manipulations
This unit showed various ways you can manipulate strings. Using C standard library functions (and possibly compiler-specific functions as well), you can copy, concatenate, compare, and search strings. These are all needed tasks in most programming projects.

String Conversions
The standard library also contains functions for converting the case of characters in strings and for converting strings to numbers.

Character Testing
Finally, C provides a variety of character-test functions or more accurately, macros that perform a variety of tests on individual characters. By using these macros to test characters, you can create your own custom input functions.

Unit 4. Day 18 Getting More from Functions

As you should know by now, functions are central to C programming. In this unit you will examine additonal functions.

Today, you learn

- Pointers as arguments to functions.
- Type void pointers as arguments.
- How to use functions with a variable number of arguments.
- How to return a pointer from a function.

Some of these topics have been discussed earlier in the course, but this unit gives you more detailed information.

Topic 4.1: Passing Pointers to Functions

Passing by Value
The default method of passing an argument to a function is *by value*. Passing by value means the function is passed a copy of the argument's value.

Three Steps
This method has three steps:

Step	Action
1.	The argument expression is evaluated.
2.	The result is copied onto the *stack,* a temporary storage area in memory.
3.	The function retrieves the argument's value from the stack.

Click the Tip button on the toolbar.

DON'T pass large amounts of data by value if it is not necessary. You could run out of stack space.

DO pass variables by value if you don't want the original value altered.

DON'T forget that a variable passed by reference should be a pointer. Also, use the indirection operator (also known as the dereferencing operator) to dereference the variable in the function.

Schematic Representation
The procedure of passing an argument by value is illustrated in Figure 18.1 In this case, the argument is a simple type int variable, but the principle is the same for other variable types and more complex expressions.

Function Cannot Modify Original Variable
When a variable is passed to a function by value, the function has access to the variable's value but not to the original copy of the variable. As a result, the code in the function cannot modify the original variable. This is the main reason *by value* is the default method of passing arguments. Data outside a function is protected from inadvertent modification.

Allowable Data Types
Passing by value is possible with the basic data types (char, int, long, float, and double) and structures.

Passing by Reference
There is another way to pass an argument to a function, however; pass a pointer to the argument variable rather than the value of the variable itself. This method of passing an argument is called *passing by reference*.

Allowable Data Types
As you learned on Day 9, "Pointers," passing by reference is the only way to pass an array to a function; passing by value is not possible with arrays. With other data types, however, you can use either method. If your program uses large structures, passing them by value might cause your program to run out of stack space.

Advantage and Disadvantage
Aside from this consideration, passing an argument by reference instead of by value offers an advantage as well as a disadvantage:

- The advantage of passing by reference is that the function can modify the value of the argument variable.

- The disadvantage of passing by reference is that the function can modify the value of the argument variable.

"Help!" you might be hollering. "An advantage that's also a disadvantage?" Yes, it all depends on the specific situation. If your program requires a function to modify an argument variable, passing by reference is an advantage. If there is no such need, it is a disadvantage because of the possibility of inadvertent modifications.

Why Not Use Function's Return Value
You may be wondering why you don't use the function's return value to modify the argument variable. You can do this, of course, as shown in the following example. Click the Example link.

Example

Remember, however, that a function can return only a single value. By passing one or more arguments by reference, you allow a function to "return" more than one value to the calling program.

```
x = half(x);

int half(int y)
{
     return y/2;
}
```

Schematic Representation
Figure 18.2 illustrates passing by reference for a single argument.

The function used in Figure 18.2 is not a good example of something you would use *passing by reference* for in a real program, but it does illustrate the concept. When you pass by reference, you must ensure that the function definition and prototype reflect the fact that the argument passed to the function is a pointer. Within the body of the function, you must also use the indirection operator (also known as the dereferencing operator) to access the variable(s) passed by reference.

Example Program
The program in Listing 18.1 demonstrates passing by reference and the default passing by value. Its output clearly shows that a variable passed by value cannot be changed by the function, whereas a variable passed by reference can be changed. Of course, a function does not need to modify a variable passed by reference, but if the function doesn't need to, there is no reason to pass by reference.

Passing By Value and By Reference
You can write a function that receives some arguments by reference and others by value. Just remember to keep them straight inside the function, using the indirection operator (*) to dereference arguments passed by reference.

Listing 18.1: Passing by Value and Passing by Reference

```
Code  1: /* LIST1801.c: Day 18 Listing 18.1 */
      2: /* Passing arguments by value and by reference. */
      3:
```

```
 4: #include <stdio.h>
 5:
 6: void by_value(int a, int b, int c);
 7: void by_ref(int *a, int *b, int *c);
 8:
 9: int main(void) {
10:   int x = 2, y = 4, z = 6;
11:
12:   printf("\nBefore calling by_value(), x = %d,"
13:          " y = %d, z = %d.", x, y, z);
14:
15:   by_value(x, y, z);
16:
17:   printf("\nAfter calling by_value(), x = %d,"
18:          " y = %d, z = %d.", x, y, z);
19:
20:   by_ref(&x, &y, &z);
21:
22:   printf("\nAfter calling by_ref(), x = %d,"
23:          " y = %d, z = %d.", x, y, z);
24:   return 0;
25: }
26:
27: void by_value(int a, int b, int c) {
28:   a = 0;
29:   b = 0;
30:   c = 0;
31: }
32:
33: void by_ref(int *a, int *b, int *c) {
34:   *a = 0;
35:   *b = 0;
36:   *c = 0;
37: }
```

Output

```
Before calling by_value(), x = 2, y = 4, z = 6.
After calling by_value(), x = 2, y = 4, z = 6.
After calling by_ref(), x = 0, y = 0, z = 0.
```

Description

This program demonstrates the difference between passing variables by value and passing them by reference. Lines 6 and 7 contain prototypes for the two functions called in the program. Notice that line 6 describes three arguments of type int for the by_value() function, but by_ref() differs on line 7 because it requires three pointers to type int variables as arguments. The function headers for these two functions on lines 27 and 33 follow the same format as the prototypes. The bodies of the two functions are similar, but not the same. Both functions assign 0 to the three variables passed to them. In the by_value() function, 0 is assigned directly to the variables. In the by_ref() function, pointers are used, so the variables must be dereferenced.

Each function is called once by main(). First, the three variables to be passed are assigned values other than zero on line 10. Lines 12 – 13 print these values to the screen. Line 15 calls the first of the two functions, by_value(). Lines 17 – 18 print the three variables once again. Notice that they are not changed. The by_value() function received the variables by value and therefore could not change their original content. Line 20 calls by_ref(), and lines 22 – 23 print the values again. This time the values have all changed to 0. Passing the variables by reference gave by_ref() access to the actual contents of the variables.

Topic 4.2: Type void Pointers

Creating a Generic Pointer

You've seen the void keyword used to specify that a function either doesn't take arguments or return a value. The void keyword also can be used to create a "generic" pointer, a pointer that can point to any type of data object. Click the Example link and then the Tip button.

Example

DO cast a void pointer when you use the value it points to.

DON'T try to increment or decrement a void pointer.

The statement

```
void *x;
```

declares x as a generic pointer. x points to something; you just haven't yet specified *what*.

Usage
The most common use for type `void` pointers is in declaring function parameters. You might want to create a function that can handle different types of arguments: you can pass it a type int one time, a type float the next time, and so on. By declaring that the function takes a `void` pointer as an argument, you don't restrict it to accepting only a single data type. If you declare the function to take a `void` pointer as an argument, you can pass the function a pointer to anything. Click the Example link.

Example

Purpose of the Function
Here's a simple example: you want a function that accepts a numeric variable as an argument and divides it by 2, returning the answer in the argument variable. Thus, if the variable x holds the value 4, after a call to half(x) the variable x is equal to 2. Because you want to modify the argument, you pass it by reference.

Declare the Function to Take a void Pointer
Because you want to use the function with any of C's numeric data types, you declare the function to take a void pointer:

```
void half(void *x);
```

Now you can call the function, passing it any pointer as an argument.

Type Cast the void Pointer
There's one more thing necessary, however. Although you can pass a void pointer without knowing what data type it points to, you cannot yet dereference the pointer. Before the code in the function can do anything with the pointer, the data type must be known. This is done with a *type cast,* which is nothing more than a way of telling the program to "treat this void pointer as a pointer to type." If x is a void pointer, you would type cast it as follows:

```
(type *)x
```

where type is the appropriate data type.

Dereference the Pointer
Usually the typecasting is done within the same statement as the dereferencing. Therefore, to typecast and dereference the pointer—that is, to access the int that x points to—write

```
*(int *)x
```

Modify the Function Definition
Getting back to the original topic (passing a void pointer to a function), you can see that to use the pointer, the function must know the data type to which it points. In the case of the function you're writing to halve its argument, there are four possibilities for type: int, long, float, and double. In addition to the void pointer to the variable to be

halved, you must tell the function the type of variable to which the void pointer points. You can modify the function definition as follows:

```
void half(void *x, char type);
```

Based on the argument type, you can make the function cast the void pointer x to the appropriate type. Then the pointer can be dereferenced and the value of the pointed-to variable can be used.

Example Program

The final version of the function half() is shown in Listing 18.2.

Listing 18.2: Using a void Pointer to Pass Different Data Types to a Function

```
Code  1: /* LIST1802.c: Day 18 Listing 18.2 */
      2: /* Using type void pointers. */
      3:
      4: #include <stdio.h>
      5:
      6: void half(void *x, char type);
      7:
      8: int main(void) {
      9:   /* Initialize one variable of each type. */
     10:
     11:   int i = 20;
     12:   long l = 100000;
     13:   float f = 12.456;
     14:   double d = 123.044444;
     15:
     16:   /* Display their initial values. */
     17:
     18:   printf("\n%d", i);
     19:   printf("\n%ld", l);
     20:   printf("\n%f", f);
     21:   printf("\n%lf\n\n", d);
     22:
     23:   /* Call half() for each variable. */
     24:
     25:   half(&i, 'i');
     26:   half(&l, 'l');
     27:   half(&d, 'd');
     28:   half(&f, 'f');
     29:
     30:   /* Display their new values. */
     31:
     32:   printf("\n%d", i);
     33:   printf("\n%ld", l);
     34:   printf("\n%f", f);
     35:   printf("\n%lf\n\n", d);
     36:
     37:   return 0;
     38: }
     39:
     40: void half(void *x, char type) {
     41:   /* Depending on the value of type, case the */
     42:   /* pointer x appropriately and divide by 2. */
     43:
     44:   switch (type) {
     45:     case 'i': {
     46:       *((int *)x) /= 2;
     47:       break;
     48:       }
     49:     case 'l': {
     50:       *((long *)x) /= 2;
     51:       break;
     52:       }
     53:     case 'f': {
     54:       *((float *)x) /= 2;
     55:       break;
```

```
56:       }
57:     case 'd': {
58:       *((double *)x) /= 2;
59:       break;
60:       }
61:   }
62: }
```

Output

```
20
100000
12.456000
123.044444



10
50000
6.228000
61.522222
```

Description As implemented, the function half() on lines 40 – 62 includes no error checking (for example, if an invalid type argument is passed). However, in a real program you would not use a function to perform a task as simple as dividing a value by 2. This is an illustrative example only.

Passing the Type of the Pointed-To Variable
You might think the need to pass the type of the pointed-to variable makes the function less flexible. The function would be more general if it didn't need to know the type of the pointed-to data object, but that's not the way C works. You must always cast a void pointer to a specific type before you dereference it. By taking the preceding approach, you write only one function. If you do not make use of a void pointer, you would need to write four separate functions, one for each data type.

Functions Versus Macros
When you need a function that can deal with different data types, you often can write a *macro* to take the place of the function. The example just presented—in which the task performed by the function is relatively simple—would be a good candidate for a macro. (Day 21, "Taking Advantage of Preprocessor Directives and More," covers macros.)

Incrementing/ Decrementing a Type void Pointer
Note that type void pointers cannot be incremented or decremented.

Topic 4.3: Functions with Variable Numbers of Arguments

STDARG.H Header File
You have used several library functions, such as printf() and scanf(), that take a variable number of arguments. You can write your own functions that take a variable argument list. Programs that have functions with variable argument lists must include the header file STDARG.H.

Variable Argument Lists
When you declare a function that takes a variable argument list, you first list the fixed parameters, those that are always present (there must be at least one fixed parameter). You then include an *ellipsis* (...) at the end of the parameter list to indicate that zero or more additional arguments are passed to the function. During this discussion, please remember the distinction between a parameter and an argument, as explained on Day 5, "Functions: The Basics."

Number of Arguments
How does the function know how many arguments have been passed to it on a specific call? You tell it. One of the fixed parameters informs the function of the total number of arguments. For example, when using the printf() function, the number of conversion specifiers in the format string tells the function how many additional arguments to expect. More directly, a fixed argument can be the number of additional arguments. The example that follows uses this approach, but first you need to look at the tools that C provides for dealing with a variable argument list.

The Type of Each Argument
The function must also know the type of each argument in the variable list. In the case of printf(), the conversion specifiers indicate the type of each argument. In other cases—such as the following example—all arguments in the variable list are of the same type, so there's no problem. To create a function that accepts different types in the variable argument list, you must devise a method of passing information about the argument types.

Tools in STDARG.H
The tools for using a variable argument list are defined in STDARG.H. These tools are used within the function to retrieve the arguments in the variable list. They are as follows:

Tool	Description
va_list	A pointer data type.
va_start()	A macro used to initialize the argument list.
va_arg()	A macro used to retrieve each argument, in turn, from the variable list.
va_end()	A macro used to "clean up" when all arguments have been retrieved.

How the Macros Are Used
The way these macros are used is outlined here, followed by an example. Click the Use of Macros link.

Use of Macros

Example Program
Now for that example. The function average() in Listing 18.3 calculates the arithmetic average of a list of integers.

Strictly speaking, a function that accepts a variable number of arguments does not need to have a fixed parameter informing it of the number of arguments being passed. You could, for example, mark the end of the argument list with a special value not used elsewhere. This method places limitations on the arguments that can be passed, however, and is best avoided.

Listing 18.3: Using a Variable Argument List

```
Code  1: /* LIST1803.c: Day 18 Listing 18.3 */
      2: /* Functions with a variable argument list. */
      3:
      4: #include <stdio.h>
      5: #include <stdarg.h>
      6:
      7: float average(int num, ...);
      8:
      9: int main(void) {
     10:   float x;
     11:
     12:   x = average(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
     13:   printf("\nThe first average is %f.", x);
     14:
     15:   x = average(5, 121, 206, 76, 31, 5);
     16:   printf("\nThe second average is %f.", x);
     17:
     18:   return 0;
     19: }
     20:
     21: float average(int num, ...) {
     22:   /* Declare a variable of type va_list. */
     23:
     24:   va_list arg_ptr;
     25:   int count, total = 0;
     26:
     27:   /* Initialize the argument pointer. */
```

```
28:
29:   va_start(arg_ptr, num);
30:
31:   /*Retrieve each argument in the variable list. */
32:
33:   for (count = 0; count < num; count++)
34:     total += va_arg(arg_ptr, int);
35:
36:   /* Perform clean up. */
37:
38:   va_end(arg_ptr);
39:
40:   /* Divide the total by the number of values to */
41:   /* get the average. Cast the total to type */
42:   /* float so the value returned is type float. */
43:
44:   return ((float)total/num);
45: }
```

Output

```
The first average is 5.500000.
The second average is 87.800003.
```

Description

The function average() is first called on line 21. The first argument passed, the only fixed argument, specifies the number of values in the variable argument list. As an argument in the variable list is retrieved on lines 33 – 34, it is added to the variable total. Once all arguments have been retrieved, line 44 casts total as type float and then divides total by num to obtain the average.

Two other things should be pointed out in this listing. Line 29 calls va_start() to initialize the argument list. This must be done before the values are retrieved. Line 38 calls va_end() to "clean up" because the function is done with the values. Both of these functions should be used in your programs.

When the function is called, the function accesses its arguments as follows:

Step	Action
1.	Declare a pointer variable of type va_list. This pointer is used to access the individual arguments. It is common practice, although certainly not required, to call this variable arg_ptr.
2.	Call the macro va_start(), passing it the pointer arg_ptr as well as the name of the last fixed argument. The macro va_start() has no return value; it initializes the pointer arg_ptr to point at the first argument in the variable list.
3.	To retrieve each argument, call va_arg(), passing it the pointer arg_ptr and the data type of the next argument. The return value of va_arg() is the value of the next argument. If the function has received *n* arguments in the variable list, call va_arg() *n* times to retrieve the arguments in the order listed in the function call.
4.	When all of the arguments in the variable list have been retrieved, call va_end(), passing it the pointer arg_ptr. In some implementations, this macro performs no action; but in others, it performs necessary cleanup actions. You should get in the habit of calling va_end() in case you use a C implementation that requires it.

Topic 4.4: Functions that Return a Pointer

Functions that Return a Pointer

You have seen several functions from the C standard library that return a pointer to the calling program. You can write your own functions that return a pointer as well. As you might expect, the indirection operator (*) (also known as the

your own functions that return a pointer as well. As you might expect, the indirection operator (*) (also known as the dereferencing operator) is used in both the function declaration and the function definition.

Syntax
The general form is

```
type *func(parameter_list);
```

This statement declares a function func() that returns a pointer to type `type`. Click the Example link and then the Tip button.

Example

DO use all the elements described previously when writing functions with variable arguments. This is true even if your compiler does not require all the elements. The parts are va_list, va_start(), va_arg(), and va_end().

DON'T confuse pointers to functions with functions that return pointers.

Here are two concrete examples:

```
double *func1(parameter_list);
struct address *func2(parameter_list);
```

The first line declares a function that returns a pointer to type double. The second line declares a function that returns a pointer to type address (which you assume is a user-defined structure).

Pointers to Functions Versus Functions That Return Pointers
Don't confuse a *function that returns a pointer* and a *pointer to a function*. If you include an additional pair of parentheses in the declaration, you declare the latter. Click the Example link.

Example

Usage
Now that you have the declaration format straight, how do you use a function that returns a pointer? There's nothing special about such functions–you use them like any other function, assigning their return value to a variable of the appropriate type (in this case, a pointer). Because the function call is a C expression, you can use it anywhere a pointer of that type would be used.

```
double (*func)(...);    /* Pointer to a function that
                           returns a double. */

double *func(...);      /* Function that returns a
                           pointer to a double. */
```

Example Program
Listing 18.4 presents a simple example, a function that is passed two arguments and determines which is larger. In the listing, two different functions accomplish this task: one returning an int and the other returning a pointer to int.

Listing 18.4: Returning a Pointer from a Function

```
Code  1: /* LIST1804.c: Day 18 Listing 18.4 */
      2: /* Function that returns a pointer. */
      3:
      4: #include <stdio.h>
      5:
      6: int larger1(int x, int y);
```

```
 7: int *larger2(int *x, int *y);
 8:
 9: int main(void) {
10:   int a, b, bigger1, *bigger2;
11:
12:   printf("Enter two integer values: ");
13:   scanf("%d %d", &a, &b);
14:
15:   bigger1 = larger1(a, b);
16:   printf("\nThe larger value is %d.", bigger1);
17:
18:   bigger2 = larger2(&a, &b);
19:   printf("\nThe larger value is %d.", *bigger2);
20:
21:   return 0;
22: }
23:
24: int larger1(int x, int y) {
25:   if (y > x)
26:     return y;
27:   return x;
28: }
29:
30: int *larger2(int *x, int *y) {
31:   if (*y > *x)
32:     return y;
33:   return x;
34: }
```

Output

```
Enter two integer values: 1111 3000
The larger value is 3000.
The larger value is 3000.
```

Description

This is a relatively easy program to follow. Lines 6 and 7 contain the prototypes for the two functions. The first, larger1(), receives two int variables and returns an int. The second, larger2(), receives two pointers to int variables and returns a pointer to an int.

The main() function on lines 9 – 22 is straightforward. Line 10 declares four variables. a and b hold the two variables to be compared. bigger1 and bigger2 hold the return values from the larger1() and larger2() functions respectively. Notice that bigger2 is a pointer to an int and bigger1 is just an int.

Line 15 calls larger1() with the two ints, a and b. The value returned from the function is assigned to bigger1, which is printed on line 16. Line 18 calls larger2() with the address of the two ints. The value returned from larger2(), a pointer, is assigned to bigger2, also a pointer. This value is dereferenced and printed on the following line.

The two comparison functions are very similar. They both compare the two values. The larger value is returned. The difference between the functions is in larger2(). In this function, the values pointed to are compared on line 31. Then the pointer for the larger value's variable is returned. Notice that the dereference operator is used in the comparisons, but not in the return statements on lines 32 or 33.

In many cases, as in the previous example, it is equally feasible to write a function to return a value or a pointer. Which you select depends on the specifics of your program—mainly on how you intend to use the return value.

Topic 4.5: Day 18 Q&A

Questions & Answers

Here are some questions to help you review what you have learned in this unit.

Question 1
Is passing pointers a common practice in C programming?

Answer
Definitely! In many instances a function needs to change multiple variables; there are two ways this can be accomplished. The first is to declare and use global variables. The second is to pass pointers so that the function can modify the data directly. The first option is good only if nearly every function is going to use the variable.

Question 2
Is it better to modify a value by returning it or by passing a pointer to the actual data?

Answer
When you need to modify only one value with a function, usually it is best to return the value from the function rather than pass a pointer to the function. The logic behind this is simple. By not passing a pointer, you don't run the risk of changing any data that you did not intend to change, and you keep the function independent of the rest of the code.

Topic 4.6: Day 18 Think About Its

Think About Its
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

```
void squared(void *nbr) {
     *nbr *= *nbr;
}
```

```
float total( int num, ...) {
    int count, total = 0;
    for ( count = 0; count < num; count++ )
           total += va_arg( arg_ptr, int );
    return ( total );
}
```

Topic 4.7: Day 18 Try Its

Try Its
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Write the prototype for a function that returns an integer. It should take a pointer to a character array as its argument.

Answer

```
int function(char array[]);
```

*** Exercise 2**
Write a prototype for a function called numbers that takes three integer arguments. The integers should be passed by reference.

Answer

```
int numbers(int *nbr1, int *nbr2, int *nbr3);
```

*** Exercise 3**
Show how you would call the numbers function in exercise two with the three integers, int1, int2, and int3.

Answer

```
int number1 = 1, number2 = 2, number3 = 3;
numbers(&number1, &number2, &number3);
```

*** Exercise 4**
BUG BUSTERS: Is anything wrong with the following?

```
void squared(void *nbr) {
     *(int *)nbr *= *(int *)nbr;
}
```

Answer

Although the code might look confusing, it is perfectly correct. This function takes the value being pointed at by nbr and multiplies it by itself.

*** Exercise 5**
BUG BUSTERS: Is anything wrong with the following?

```
float total( int num, ...) {
    int count, total = 0;
    for ( count = 0; count < num; count++ )
           total += va_arg( arg_ptr, int );
    return ( total );
}
```

Answer

Listing 18.3: Using a Variable Argument List

```
Code  1:  /* LIST1803.c: Day 18 Listing 18.3 */
      2:  /* Functions with a variable argument list. */
      3:
      4:  #include <stdio.h>
      5:  #include <stdarg.h>
      6:
      7:  float average(int num, ...);
      8:
      9:  int main(void) {
     10:    float x;
     11:
     12:    x = average(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
     13:    printf("\nThe first average is %f.", x);
     14:
     15:    x = average(5, 121, 206, 76, 31, 5);
     16:    printf("\nThe second average is %f.", x);
     17:
```

```
17:
18:     return 0;
19:   }
20:
21:   float average(int num, ...) {
22:     /* Declare a variable of type va_list. */
23:
24:     va_list arg_ptr;
25:     int count, total = 0;
26:
27:     /* Initialize the argument pointer. */
28:
29:     va_start(arg_ptr, num);
30:
31:     /*Retrieve each argument in the variable list. */
32:
33:     for (count = 0; count < num; count++)
34:       total += va_arg(arg_ptr, int);
35:
36:     /* Perform clean up. */
37:
38:     va_end(arg_ptr);
39:
40:     /* Divide the total by the number of values to */
41:     /* get the average. Cast the total to type */
42:     /* float so the value returned is type float. */
43:
44:     return ((float)total/num);
45:   }
```

When using variable parameter lists, you should use all the macro tools. This includes va_list, va_start(), va_arg(), and va_end(). See Listing 18.3 for the correct way to use variable parameter lists. The corrected code would be:

```
float total( int num, ...) {
   int count, total = 0;
   va_list arg_ptr;
   va_start(arg_ptr, num);
   for ( count = 0; count < num; count++ )
      total += va_arg( arg_ptr, int );
   va_end(arg_ptr);
   return ( total );
}
```

*** Exercise 6**

Write a function that a) is passed a variable number of strings as arguments, b) concatenates the strings, in order, into one longer string, and c) returns a pointer to the new string to the calling program.

Answer

```
/* EXER1806.C Day 18 Exercise 6 */
/* Concatenate variable number of strings */

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


char *newstrng(int num, ...);
char *sptr;
```

```
int main(void)  {

  sptr = newstrng(5, "This ", "is ", "a ", "simple ", "test.");

  printf("Concatenated string: %s\n", sptr);

  return 0;
}

char *newstrng(int num, ...)  {

  va_list arg_ptr;
  int count;
  char *bptr, *bptr2, *s;
  char buff[80];

  /* Initialize the argument pointer */

  va_start(arg_ptr,num);
  s = &(buff[0]);

  /* Get first string and store it in allocated memory */
  /* Allow space for null character */

  if (num > 0) {
    s = va_arg(arg_ptr, char *);
    printf("Retrieved string: %s, space required: %d\n",
        s, strlen(s)+1);
    bptr = (char *)malloc(strlen(s) + 1);
    strcpy(bptr,s);
    num--;
  }

  /* Get remaining strings and concatenate them */

  for (count = 0; count < num; count++)  {
    s = va_arg(arg_ptr,char *);
    printf("Retrieved string: %s, new length: %d, "
       "old length: %d\n", s, strlen(s), strlen(bptr));
    bptr2 = (char *)malloc(strlen(s) + strlen(bptr) + 1);
    strcpy(bptr2,bptr);
    strcat(bptr2, s);
    free(bptr);
    bptr = bptr2;
  }

  return bptr;
}
```

*** Exercise 7**

Write a function that a) is passed an array of any numeric data type as an argument, b) finds the largest and smallest values in the array, and c) returns pointers to these values to the calling program. (Hint: You need some way to tell the function how many elements are in the array.)

Answer

```
/* EXER1807.C Day 18 Exercise 7 */
/* Function to pass largest & smallest values in an array */

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

```
void minmax(void *ptr, void *max, void *min,
            int num, char type);

int i_array[5] = { 1, 28, 55, 12, 19 };
int i_big, i_small;
double d_array[7] =
  { 7.5, .004, 305.876, 10.02, 5298.3, 3.234, .1 };
double d_big, d_small;

int main(void)  {

  minmax(i_array, &i_big, &i_small, 5, 'i');
  printf("Smallest number: %d\n", i_small);
  printf("Largest number: %d\n", i_big);
  minmax(d_array, &d_big, &d_small, 7, 'd');
  printf("Smallest number: %f\n", d_small);
  printf("Largest number: %f\n", d_big);
  return 0;
}

void minmax(void *ptr, void *max, void *min,
            int num, char type)  {
  int x;

  switch(type)  {

    case 'i' :
      printf("array of integers\n");

      /* assume the fist one is minumum and maximum */
      *(int *)min = *(int *)max = *(int *)ptr;

      /* compare with the rest of them */
      for (x = 1; x < num; x++, ((int *)ptr)++)  {

        if (*(int *)ptr > *(int *)max)
          *(int *)max = *(int *)ptr;

        if (*(int *)ptr < *(int *)min)
          *(int *)min = *(int *)ptr;
      }
      break;

    case 'd' :
      printf("array of doubles\n");

      *(double *)min = *(double *)max = *(double *)ptr;
      for (x = 1; x < num; x++, ((double *)ptr)++)  {

        if (*(double *)ptr > *(double *)max)
          *(double *)max = *(double *)ptr;

        if (*(double *)ptr < *(double *)min)
          *(double *)min = *(double *)ptr;
      }
      break;

    case 'f' :
      printf("array of floats\n");

      *(float *)min = *(float *)max = *(float *)ptr;

      for (x = 1; x < num; x++, ((float *)ptr)++)  {

        if (*(float *)ptr > *(float *)max)
         *(float *) max = *(float *)ptr;

        if (*(float *)ptr < *(float *)min)
          *(float *)min = *(float *)ptr;
```

```
            *(float *)min = *(float *)ptr;
        }
        break;

      case 'l' :
        printf("array of longs\n");

        *(long *)min = *(long *)max = *(long *)ptr;
        for (x = 1; x < num; x++, ((long *)ptr)++)  {

          if (*(long *)ptr > *(long *)max)
            *(long *)max = *(long *)ptr;

          if (*(long *)ptr < *(long *)min)
            *(long *)min = *(long *)ptr;
        }
        break;

    }
}
```

*** Exercise 8**

Write a function that accepts a string and a character. The function should look for the first occurrence of the character in the string and return a pointer to that location.

Answer

```
/* EXER1808  Day 18 Exercise 8 */
/* Search a string for a given character. */

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

#define BUFSIZE 100

char c;
char string[80];
char *loc;
char *lookup(char *ptr, char c);

int main(void)  {

  printf("Enter a string of characters: ");
  gets(string);
  printf("Enter a search character: ");
  c = getchar();

  loc = lookup(string,c);
  printf("Remainder of string: %s\n", loc);
  return 0;
}

/* Find character in string */

char *lookup(char *ptr, char c)  {
  int x, y;

  y = strlen(ptr);

  for (x = 0; x < y; x++, ptr++)
    if (*ptr == c)
      break;
```

```
  return (ptr);
}
```

Topic 4.8: Day 18 Summary

Passing Pointers to Functions
In this unit, you learned some additional things your C programs can do with functions. You learned the difference between passing arguments by value and by reference and how the latter technique allows a function to "return" more than one value to the calling program.

Type void Pointers
You also saw how the void type can be used to create a generic pointer that can point to any type of C data object. Type void pointers are most commonly used with functions that can be passed arguments not restricted to a single data type. Remember that a type void pointer must be cast to a specific type before you can dereference it.

Functions with Variable Numbers of Arguments
This unit also showed you how to use the macros defined in STDARG.H to write a function that accepts a variable number of arguments. Such functions provide considerable programming flexibility.

Functions that Return a Pointer
Finally, you saw how to write a function that returns a pointer.

Unit 5. Day 19 Exploring the Function Library

As you see throughout this course, much of C's power comes from the standard library functions. In this unit, you explore some of the functions that do not fit into the subject matter of other units.

Today, you learn:

- Mathematical functions.
- Functions that deal with time.
- Error-handling functions.
- Functions for searching and sorting data.

Topic 5.1: Mathematical Functions

Description
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 2pi radians.

Topic 5.1.1: Trigonometric Functions

acos() Function
```
double acos(double x);
```

The function acos() 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 <= pi.

asin() Function
```
double asin(double x);
```

The function asin() returns the arcsine of its argument. The argument must be in the range –1 <= x <= 1, and the return value is in the range –pi/2 <= asin <= pi/2.

atan() Function
```
double atan(double x);
```

The function atan() returns the arctangent of its argument. The return value is in the range –pi/2 <= atan <= pi/2.

atan2() Function
```
double atan2(double x, double y);
```

The function atan2() returns the arctangent of x/y. The value returned is in the range –pi <= atan2 <= pi.

cos() Function
```
double cos(double x);
```

The function cos() returns the cosine of its argument.

sin() Function
```
double sin(double x);
```

The function sin() returns the sine of its argument.

tan() Function
```
double tan(double x);
```

The function tan() returns the tangent of its argument.

Topic 5.1.2: Exponential and Logarithmic Functions

Topic 5.1.2: Exponential and Logarithmic Functions

exp() Function

```
double exp(double x);
```

The function exp() returns the natural exponent of its argument, that is, ex where e == 2.7182818284590452354.

log() Function

```
double log(double x);
```

The function log() returns the natural logarithm of its argument. The argument must be greater than 0.

log10() Function

```
double log10(double x);
```

The function log10() returns the base 10 logarithm of its argument. The argument must be greater than 0.

frexp() Function

```
double frexp(double x, int *y);
```

The function frexp() 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() Function

```
double ldexp(double x, int y);
```

The function ldexp() returns x * 2y.

Topic 5.1.3: Hyperbolic Functions

Hyperbolic Functions
The hyperbolic functions perform hyperbolic trigonometric calculations.

cosh() Function

```
double cosh(double x);
```

The function cosh() returns the hyperbolic cosine of its argument.

sinh() Function

```
double sinh(double x);
```

The function sinh() returns the hyperbolic sine of its argument.

tanh() Function

```
double tanh(double x);
```

The function tanh() returns the hyperbolic tangent of its argument.

Topic 5.1.4: Other Mathematical Functions

Other Mathematical Functions
This section lists some miscellaneous mathematical functions.

sqrt() Function

```
double sqrt(double x);
```

The function sqrt() returns the square root of its argument. The argument must be zero or greater.

ceil() Function

```
double ceil(double x);
```

The function ceil() 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.

fabs() Function

```
double fabs(double x);
```

The function fabs() returns the absolute value of a floating-point number.

floor() Function

```
double floor(double x);
```

The function floor() 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() Function

```
double modf(double x, double *i-ptr);
```

The function modf()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 *i-ptr.

pow() Function

```
double pow(double x, double y);
```

The function pow() returns xy. An error occurs if x == 0 and y <= 0, or if x < 0 and y is not an integer.

fmod() Function

```
double fmod(double x, double y);
```

The function fmod() returns the floating point remainder of x/y, with the same sign as x. The function returns 0 if x == 0.

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

Listing 19.1: Using the C Library Math Functions

```
Code  1: /* LIST1901.c: Day 19 Listing 19.1 */
      2: /* Demonstrate some math functions. */
      3:
      4: #include <stdio.h>
      5: #include <math.h>
      6:
      7: int main(void) {
      8:   double x;
      9:
     10:   printf("Enter a number: ");
     11:   scanf("%lf", &x);
     12:
     13:   printf("\n\nOriginal value: %lf", x);
     14:
     15:   printf("\nCeil: %lf", ceil(x));
     16:   printf("\nFloor: %lf", floor(x));
     17:   if (x >= 0)
     18:     printf("\nSquare root: %lf", sqrt(x));
     19:     else
     20:         printf("\nNegative number");
     21:
     22:     printf("\nCosine: %lf", cos(x));
     23:     return 0;
```

24:	}
Output	Enter a number: 100.95 Original value: 100.950000 Ceil: 101.000000 Floor: 100.000000 Square root: 10.047388 Cosine: 0.913482
Description	This listing uses just a few of the math functions. A value accepted on line 11 is printed after it is sent to four of the math functions, ceil(), floor(), sqrt(), and cos(). Notice that sqrt() is called only if the number is not negative. You cannot get the square root of a negative number. Any of the other math functions could be added to a program such as this to test their functionality.

Topic 5.2: Dealing with Time

TIME.H Header File
The C library contains several functions that allow your program to work with times. The function prototypes and the definition of the structure used by many of the time functions are in the header file TIME.H.

Topic 5.2.1: Representation of Time

Basic Method
The C time functions represent time 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.

A Second Method
The second method represents a time broken down into its components: year, month, day, etc. 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 0..59 */
       tm_min,      /* minutes 0..59 */
       tm_hour,     /* hour of day 0..23 */
       tm_mday,     /* day of month 1..31 */
       tm_mon,      /* month 0..11 */
       tm_year,     /* years since 1900 */
       tm_wday,     /* day of week, 0..6 (Sunday..Saturday) */
       tm_yday,     /* day of year, 0..365 */
       tm_isdst;    /* > 0 if daylight savings time */
                    /* == 0 if not DST */
                    /* < 0 if don't know */
};
```

Topic 5.2.2: The Time Functions

The Time Functions

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

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

Syntax
The prototype is

```
time_t time(time_t *t-ptr);
```

Remember, time_t is defined in TIME.H as a synonym for long.

Return Value
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 t-ptr.

Storing the Current Time in the Type time_t Variable
Thus, to store the current time in the type time_t variable now, you couldwrite

```
time_t now;

now = time(0);
```

You also could write

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

localtime() Function
Because knowing the number of seconds since January 1, 1970 is not very useful, time represented as a time_t value can be converted to a tm structure by the localtime() function. A tm structure contains day, month, year, and other time information in a format more appropriate for display and printing.

Syntax
The prototype of this function is

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

Return Value
This function returns a pointer to a static type tm structure, so you do not need to declare a type tm structure to use, but 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.

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

Syntax
The prototype is

```
time_t mktime(struct tm *t-ptr);
```

Return Value
The function returns the number of seconds between midnight, January 1, 1970, and the time represented by the type tm structure pointed to by t-ptr.

ctime() and asctime() Functions
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.

Syntax
Their prototypes are

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

Return Value
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 are called.

strftime() Function
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.

Syntax
The function prototype is

```
size_t strftime(char *s, size_t maximum-size,
                char *fmt, struct tm *t-ptr);
```

Arguments
The function takes the time in the type tm structure pointed to by t-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 maximum-size should specify the amount of space allocated at s.

Return Value
If the resulting string (including the terminating null character) has more than maximum-size characters, the function returns 0 and the string s is invalid. Otherwise, the function returns the number of characters written—strlen(s).

Conversion Specifiers
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	Is 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-99).
%d	Day of month as a decimal number 01–31.
%H	The hour (24-hour clock) as a decimal number 00–23.
%I	The hour (12-hour clock) as a decimal number 00–11.
%j	The day of the year as a decimal number 001–366.
%m	The month as a decimal number 01–12.
%M	The minute as a decimal number 00–59.
%p	AM or PM.
%S	The second as a decimal number 00–59.
%U	The week of the year as a decimal number 00–53; Sunday is considered the first day of the week.
%w	The weekday as a decimal number 0–6 (Sunday = 0)

%w	The weekday as a decimal number 0–6 (Sunday = 0).
%W	The week of the year as a decimal number 00–53; Monday is considered the first day of the week.
%x	The date representation (for example, 30-Jun-99).
%X	The time representation (for example, 10:41:50).
%y	The year, without century, as a decimal number 00–99.
%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 %.

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

Syntax
The prototype is

```
double difftime(time_t end-time,
                time_t start-time);
```

Return Value
The function subtracts start-time from end-time and returns the difference, the number of seconds between the two times.

Example Program
A common use for difftime() is to calculate elapsed time, as is demonstrated (along with other time operations) in Listing 19.2.

Listing 19.2: Using the C Library Time Functions

```
Code  1: /* LIST1902.c: Day 19 Listing 19.2 */
      2: /* Demonstrates the time functions. */
      3:
      4: #include <stdio.h>
      5: #include <time.h>
      6:
      7: int main(void) {
      8:   time_t start, finish, now;
      9:   struct tm *ptr;
     10:   char *c, buf1[80];
     11:   double duration;
     12:
     13:   /* Record time the program starts execution. */
     14:
     15:   start = time(0);
     16:
     17:   /* Record current time, using the alternate */
     18:   /* method of calling time(). */
     19:
     20:   time(&now);
     21:
     22:   /* Convert time_t value into type tm structure.*/
     23:
     24:   ptr = localtime(&now);
     25:
     26:   /* Create and display a formatted string */
     27:   /* containing the current time. */
     28:
     29:   c = asctime(ptr);
     30:   puts(c);
```

```
31:   getchar();
32:
33:   /* Now use the strftime() function to create */
34:   /* several different formatted versions of */
35:   /* the time. */
36:
37:   strftime(buf1, 80,
38:            "This is week %U of the year %Y", ptr);
39:   puts(buf1);
40:   getchar();
41:
42:   strftime(buf1, 80, "Today is %A, %x", ptr);
43:   puts(buf1);
44:   getchar();
45:
46:   strftime(buf1, 80,
47:            "It is %M minutes past hour %I.", ptr);
48:   puts(buf1);
49:   getchar();
50:
51:   /* Now get the current time and calculate */
52:   /* program duration. */
53:
54:   finish = time(0);
55:   duration = difftime(finish, start);
56:   printf("\nProgram execution time = %f seconds.",
57:          duration);
58:
59:   /* Also display program duration in hundreths */
60:   /* of seconds using clock(). */
61:
62:   printf("\nProgram execution time = "
63:          "%ld hundredths of sec.",
64:       clock());
65:   return 0;
66: }
```

Output

```
Sun Aug 23 19:54:25 1992
This is week 34 of the year 1992
Today is Sunday, Sun Aug 23, 1992
It is 54 minutes past hour 07.
Program execution time = 22.000000 seconds.
Program execution time = 401 hundredths of sec.
```

Description

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 5. Line 8 declares three variables of type time_t called 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 previously. 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 a character to be pressed.

Lines 37 – 44 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 once again on line 54. This records the time the program ends. Line 55 uses this ending time along with the starting time to calculate the program's duration. This value is printed on line 56. The program concludes by printing the

	program execution time from the clock() function.

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

Syntax
The prototype is

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

Use of Return Values
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.

Topic 5.2.3: Using the Time Functions

Example Program
The program in Listing 19.2 demonstrates how to use the C library time functions.

Listing 19.2: Using the C Library Time Functions

```
Code  1: /* LIST1902.c: Day 19 Listing 19.2 */
      2: /* Demonstrates the time functions. */
      3:
      4: #include <stdio.h>
      5: #include <time.h>
      6:
      7: int main(void) {
      8:   time_t start, finish, now;
      9:   struct tm *ptr;
     10:   char *c, buf1[80];
     11:   double duration;
     12:
     13:   /* Record time the program starts execution. */
     14:
     15:   start = time(0);
     16:
     17:   /* Record current time, using the alternate */
     18:   /* method of calling time(). */
     19:
     20:   time(&now);
     21:
     22:   /* Convert time_t value into type tm structure.*/
     23:
     24:   ptr = localtime(&now);
     25:
     26:   /* Create and display a formatted string */
     27:   /* containing the current time. */
     28:
     29:   c = asctime(ptr);
     30:   puts(c);
     31:   getchar();
     32:
     33:   /* Now use the strftime() function to create */
     34:   /* several different formatted versions of */
     35:   /* the time. */
     36:
     37:   strftime(buf1, 80,
     38:            "This is week %U of the year %Y", ptr);
     39:   puts(buf1);
     40:   getchar();
     41:
     42:   strftime(buf1, 80, "Today is %A, %x", ptr);
```

```
42:   strftime(buf1, 80, "Today is %A, %x", ptr);
43:   puts(buf1);
44:   getchar();
45:
46:   strftime(buf1, 80,
47:            "It is %M minutes past hour %I.", ptr);
48:   puts(buf1);
49:   getchar();
50:
51:   /* Now get the current time and calculate */
52:   /* program duration. */
53:
54:   finish = time(0);
55:   duration = difftime(finish, start);
56:   printf("\nProgram execution time = %f seconds.",
57:          duration);
58:
59:   /* Also display program duration in hundreths */
60:   /* of seconds using clock(). */
61:
62:   printf("\nProgram execution time = "
63:          "%ld hundredths of sec.",
64:       clock());
65:   return 0;
66: }
```

Output

```
Sun Aug 23 19:54:25 1992
This is week 34 of the year 1992
Today is Sunday, Sun Aug 23, 1992
It is 54 minutes past hour 07.
Program execution time = 22.000000 seconds.
Program execution time = 401 hundredths of sec.
```

Description

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 5 Line 8 declares three variables of type time_t called 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 previously. 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 a character to be pressed.

Lines 37 – 44 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 once again on line 54. This is the program-ending time. Line 55 uses this ending time along with the starting time to calculate the program's duration. This value is printed on line 56. The program concludes by printing the program execution time from the clock() function.

Topic 5.3: Error-Handling Functions

Error-Handling Functions

The C standard library contains a variety of functions and macros that assist you in dealing with program errors.

Topic 5.3.1: The assert() Function

Description
The macro assert() can diagnose program bugs. It is defined in ASSERT.H.

Syntax
Its prototype is

```
void assert(int exp);
```

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

Usage
How do you use assert()? It is most frequently used to track down program bugs (which are distinct from compilation errors). Click the Example link.

Example

Suppose a financial-analysis program you are writing gives incorrect answers occasionally. 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.

Example Program
To see the workings of assert(), run the program in 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 error message you see displayed is

```
Assertion failed: x, file list1903.c, line 13
```

Listing 19.3: Using the assert() Macro

```
Code  1: /* LIST1903.c: Day 19 Listing 19.3 */
      2: /* The assert() macro. */
      3:
      4: #include <stdio.h>
      5: #include <assert.h>
      6:
      7: int main(void) {
      8:   int x;
      9:
     10:   printf("\nEnter an integer value: ");
     11:   scanf("%d", &x);
     12:

     13:   assert(x);
     14:
     15:   printf("You entered %d.", x);
     16:   return 0;
     17: }
```

Output

```
>list1903
Enter an integer value: 10
```

	`You entered 10.`
	`>list1903`
	
	`Enter an integer value: 0`
	`Assertion failed: x, file list1903.c, line 13`
	`Abnormal program termination`

Description	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 (for "no debugging"). If the macro NDEBUG is not 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 3. Now, the program prints the value entered and terminates normally, even if you enter 0. Note that NDEBUG does not need to be defined as anything in particular, as long as it is included in a #define directive. You learn more about the #define directive on Day 21, "Taking Advantage of Preprocessor Directives and More."

Topic 5.3.2: The ERRNO.H Header File

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 following paragraphs.

errno
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.

Symbolic Constants
The file ERRNO.H also defines a group of symbolic constants for these errors. Table 19.2 gives some of the errno values for MS-DOS, the system error message for each value, and the value of each constant. Note that only the ERANGE and EDOM constants are specified in the ANSI standard.

Table 19.2: Some of the Symbolic Error Constants Defined in ERRNO.H

Constant	Meaning	Value
`E2BIG`	Argument list too long	7
`EACCES`	Permission denied	13
`EBADF`	Bad file number	9
`EDOM`	Math argument	33
`EEXIST`	File exists	17
`EMFILE`	Too many open files	24

ENOENT	No such file or directory	2
ENOEXEC	Exec format error	8
ENOMEM	Not enough memory	12
ERANGE	Result too large	34

Usage

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 is 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. After perror() is explained, use of errno is illustrated in Listing 19.4.

Listing 19.4: Using perror() and errno to Deal with Runtime Errors

```
Code  1: /* LIST1904.c: Day 19 Listing 19.4 */
      2: /* Demonstration of error handling with perror() */
      3: /* and errno. */
      4:
      5: #include <stdio.h>
      6: #include <stdlib.h>
      7: #include <errno.h>
      8:
      9: int main(void) {
     10:   FILE *fp;
     11:   char filename[80];
     12:
     13:   printf("Enter filename: ");
     14:   gets(filename);
     15:
     16:   if ((fp = fopen(filename, "r")) == NULL) {
     17:     perror("You goofed!");
     18:     printf("errno = %d.", errno);
     19:     exit(1);
     20:   }
     21:   else {
     22:     puts("File opened for reading.");
     23:     fclose(fp);
     24:   }
     25:   return 0;
     26: }
```

Output

```
>list1904
Enter file name: list1904.c
File opened for reading.

>list1904
Enter file name: notafile.xxx
You goofed!: No such file or directory
errno = 2.
```

Description

This program prints one of two messages based on whether a file can be opened for reading. Line 16 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 – 19 of the if loop execute. Line 17 calls the perror() function with the string `"You goofed!"`. This is followed by printing out the error number. The result of entering a file that does not exist is

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

Topic 5.3.3: The perror() Function

Description
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.

Syntax
The prototype, in STDIO.H, is

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

Argument
The argument s 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."

Usage
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 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 here. Click the Tip button on the toolbar.

Table 19.2: Some of the Symbolic Error Constants Defined in ERRNO.H

Constant	Meaning	Value
E2BIG	Argument list too long	7
EACCES	Permission denied	13
EBADF	Bad file number	9
EDOM	Math argument	33
EEXIST	File exists	17
EMFILE	Too many open files	24
ENOENT	No such file or directory	2
ENOEXEC	Exec format error	8
ENOMEM	Not enough memory	12
ERANGE	Result too large	34

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

DON'T include the ERRNO.H header file if you are not going to use the symbolic error constants described in Table 19.2.

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

Topic 5.4: Searching and Sorting

Description

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.

Topic 5.4.1: Searching with bsearch()

bsearch() Function
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.

Syntax
The prototype of bsearch() is in STDLIB.H:

```
void *bsearch(void *s-key, void *a-ptr,
      size_t num, size_t el-size,
      int (*compar)(void *e-1, void *e-2));
```

compar Argument
The final argument, compar, 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 criteria: 1) it is passed pointers to two data items, and 2) it returns a type int as follows:

```
< 0      element 1 is less than element 2.
0        element 1 = element 2.
> 0      element 1 is greater than element 2.
```

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

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

returns the value for el-size—the size (in bytes) of one array element. Because the expression `sizeof(array)` returns the size, in bytes, of the entire array, the statement

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

obtains the value of num, the number of elements in the array.

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

Step	Action
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.	Restrict the search to the appropriate half of the array, and return to step one.

How the Binary Search Algorithm Works
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 with only 14 comparisons. In general, a binary search requires at most $\log_2 n$ comparisons to search an array of n elements.

Topic 5.4.2: Sorting with qsort()

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

Syntax
The function prototype, defined in STDLIB.H, is

```
void qsort(void *a-ptr, size_t num,
     size_t el-size,
     int (*compar)(void *e-1, void *e-2));
```

Arguments
The argument a-ptr points at the first element in the array, num is the number of elements in the array, and el-size is the size (in bytes) of one array element. The argument compar 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 last section—you often use the same comparison function for both bsearch() and qsort().

Return Value
The function qsort() has no return value.

Topic 5.4.3: Searching and Sorting - Two Demonstrations

Sorting and Searching an Array of Values
The program in Listing 19.5 demonstrates the use of qsort() and bsearch(). The program sorts and searches an array of values.

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

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

Listing 19.5: Using the qsort() and bsearch() Functions with Values

```
Code  1: /* LIST1905.c: Day 19 Listing 19.5 */
      2: /* Using qsort() and bsearch() with values. */
      3:
      4: #include <stdio.h>
      5: #include <stdlib.h>
      6:
      7: #define MAX 20
      8:
      9: int intcmp(const void *v1, const void *v2);
     10:
     11: int main(void) {
     12:   int arr[MAX], count, key, *ptr;
     13:
     14:   /* Enter some integers from the user. */
     15:
     16:   printf("Enter %d integer values;"
     17:          " press Enter after each.\n", MAX);
```

```
18:
19:   for (count = 0; count < MAX; count++)
20:     scanf("%d", &arr[count]);
21:
22:   puts("Press Enter to sort the values.");
23:   getchar();
24:
25:   /* Sort the array into ascending order. */
26:
27:   qsort(arr, MAX, sizeof(arr[0]), intcmp);
28:
29:   /* Display the sorted array. */
30:
31:   for (count = 0; count < MAX; count++)
32:     printf("\narr[%d] = %d.", count, arr[count]);
33:
34:   puts("\nPress Enter to continue.");
35:   getchar();
36:
37:   /* Enter a search key. */
38:
39:   printf("Enter a value to search for: ");
40:   scanf("%d", &key);
41:
42:   /* Perform the search. */
43:
44:   ptr = (int *)bsearch(&key, arr, MAX,
45:              sizeof(arr[0]),intcmp);
46:
47:   if (ptr != NULL)
48:     printf("%d found at arr[%d].",
49:            key, (ptr - arr));
50:   else
51:     printf("%d not found.", key);
52:   return 0;
53: }
54:
55: int intcmp(const void *v1, const void *v2) {
56:   return (*(int *)v1 - *(int *)v2);
57: }
```

Output

```
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]
```

Description

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

Obtaining the values for the array on lines 19 and 20 is familiar code. Line 27 contains the call to qsort(), in order to get the array sorted. 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 55 – 57. 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 qsort()s. 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 is not found. On line 44, ptr is assigned the returned value of bsearch(). ptr is used in the if loop on lines 47 – 51 to print the status of the search.

Listing 19.6: Using qsort() and bsearch() with Strings

```
Code  1: /* LIST1906.c: Day 19 Listing 19.6 */
      2: /* Using qsort() and bsearch() with strings. */
      3:
      4: #include <stdio.h>
      5: #include <stdlib.h>
      6: #include <string.h>
      7:
      8: #define MAX 20
      9:
     10: int comp(const void *s1, const void *s2);
     11:
     12: int main(void) {
     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, "
     19:          "pressing Enter after each.\n", MAX);
     20:
     21:   for (count = 0; count < MAX; count++) {
     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 */
43:   /* pointer to the pointer to the search key. */
44:
45:   key = buf;
46:   key1 = &key;
47:   ptr = bsearch(key1, data, MAX,
48:                 sizeof(data[0]), comp);
49:
50:   if (ptr != NULL)
51:     printf("%s found.", buf);
52:   else
53:     printf("%s not found.", buf);
54:   return 0;
55: }
56:
57: int comp(const void *s1, const void *s2) {
58:   return (strcmp(*(char **)s1, *(char **)s2));
59: }
```

Output

```
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.
```

Description

A couple of points about Listing 19.6 bear mentioning. This program makes use of an array of pointers to strings, a technique you were introduced to on Day 15, "More on Pointers." As you saw in that unit, you can "sort" the strings by sorting the array of pointers. This method requires a modification in the comparison function, however. 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 based 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 (in this case buf) 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 but is a symbol that evaluates to the address of the array. Because of this, you cannot 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.

Topic 5.5: Day 19 Q&A

Questions & Answers
Here are some questions to help you review what you have learned in this unit.

Question 1
Why do nearly all of the math functions return doubles?

Answer
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, "Odds and Ends," you learn specifics on casting variables and variable promotion. These topics are also applicable to the precision obtained.

Question 2
Are bsearch() and qsort() the only ways in C to sort and search?

Answer
These two functions are provided; however, you do not have to use them. Many computer programming textbooks teach 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()

own. You can purchase specially written searching and sorting routines. The biggest benefits of bsearch() and qsort() are that they are already written and they are provided with any ANSI-compatible compiler.

Question 3
Do the math functions validate bad data?

Answer
Never assume that data entered is correct. Always validate data entered by a user. For example, if you pass a negative value to sqrt(), the function displays an error. If you are formatting the output, you probably don't want this error displayed as it is. For example, try removing the if statement in Listing 19.1 and entering a negative number.

Topic 5.6: Day 19 Think About Its

Think About Its
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int values[10], ctr, key, *ptr;

    printf("Enter values");
    for(ctr = 0; ctr < 10; ctr++)
        scanf("%d", &values[ctr]);

    qsort(values, 10, 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;
}
```

Topic 5.7: Day 19 Try Its

Try Its
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Write a call to bsearch(). The array to be searched is called names, the values are characters. The comparison function is called comp_names(). Assume all of the names are the same size.

Answer

```
bsearch(myname, names, (sizeof(names)/sizeof(names[0])),
        sizeof(names[0]), comp_names);
```

*** Exercise 2**

Modify Listing 19.1 so that the sqrt() function works with negative numbers. Do this by taking the absolute value of x.

Answer

Listing 19.1: Using the C Library Math Functions

```
Code  1:  /* LIST1901.c: Day 19 Listing 19.1 */
      2:  /* Demonstrate some math functions. */
      3:
      4:  #include <stdio.h>
      5:  #include <math.h>
      6:
      7:  int main(void) {
      8:    double x;
      9:
     10:    printf("Enter a number: ");
     11:    scanf("%lf", &x);
     12:
     13:    printf("\n\nOriginal value: %lf", x);
     14:
     15:    printf("\nCeil: %lf", ceil(x));
     16:    printf("\nFloor: %lf", floor(x));
     17:    if (x >= 0)
     18:      printf("\nSquare root: %lf", sqrt(x));
     19:      else
     20:          printf("\nNegative number");
     21:
     22:      printf("\nCosine: %lf", cos(x));
     23:      return 0;
     24:  }
```

```
/* EXER1902 Day 19 Exercise 2 */
/* Enable square root of negative numbers */

#include <stdio.h>
#include <math.h>

double x;
long lnum1, lnum2;

int main(void)  {

  printf("Enter a number: ");
  scanf("%lf", &x);

  printf("\n\nOriginal value: %lf", x);

  printf("\nCeil: %lf", ceil(x));
  printf("\nFloor: %lf", floor(x));
  if (x >= 0)
    printf("\nSquare root: %lf", sqrt(x));
  else  {
    lnum1 = labs((long) x);
    printf("\nSquare root: %lf", sqrt((double)lnum1));

  }

  printf("\nCosine: %lf", cos(x));

  return 0;
}
```

*** Exercise 3**

Write a program that consists of a menu that does various math functions. Use as many of the math functions as you can.

Answer

```
/* EXER1903.C Day 19 Exercise 3 */
/* Displays a menu which processes math commands. */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <float.h>
#include <math.h>

#define YES 1
#define NO  0
int menu(void);
double dprompt(char *s);
long lprompt(char *s);
void clear_kb(void);
double reply, second;
long lx;
char buf[80];
int cont = YES;
int num;

int main(void)  {
double answer;
    while (cont == YES)  {

      switch(menu())  {
      case 1:  {
  printf("\nEnter a real number:  ");
  scanf("%lf", &reply);
  answer = acos(reply);
  printf("Arccosine = %2.2f", answer);
  break;
      }
      case 2:  {
  printf("\nEnter a real number:  ");
  scanf("%lf", &reply);
  printf("Arcsine = %2.2f", asin(reply));
  break;
      }
      case 3:  {
  printf("\nEnter a real number:  ");
  scanf("%lf", &reply);
  printf("Arctangent = %2.2f", atan(reply));
  break;
      }
      case 4:  {
  printf("\nEnter a real number:  ");
  scanf("%lf", &reply);
  printf("\nEnter the second real number:  ");
  scanf("%lf", &second);
  printf("Arctangent of x/y = %2.2f",
          atan2(reply,second));
  break;
      }
      case 5:  {
  printf("\nEnter a real number:  ");
  scanf("%lf", &reply);
  printf("Cosine = %2.2f", cos(reply));
  break;
      }
```

```
        case 6:  {
      printf("\nEnter a real number:  ");
      scanf("%lf", &reply);
      printf("Sine = %2.2f", sin(reply));
      break;
          }
          case 7:  {
      printf("\nEnter a real number:  ");
      scanf("%lf", &reply);
      printf("Tangent = %2.2f", tan(reply));
      break;
          }
          case 8:  {
      printf("\nEnter a real number:  ");
      scanf("%lf", &reply);
      printf("Exponent = %2.2f", exp(reply));
      break;
          }
          case 9:  {
      printf("\nEnter a real number:  ");
      scanf("%lf", &reply);
      printf("Logarithm = %2.2f", log(reply));
      break;
          }
          case 10:  {
      printf("\nEnter a real number: ");
      scanf("%lf", &reply);
      printf("Base 10 Log = %2.2f", log10(reply));
      break;
          }
          case 11:  {
      printf("\nEnter a real number:  ");
      scanf("%lf", &reply);
      printf("\nEnter an integer:  ");
      scanf("%d", &num);
      printf("Normalized Fraction = %2.2f",
              frexp(reply, &num));
      break;
          }
          case 12:  {
      printf("\nEnter a real number:  ");
      scanf("%lf", &reply);
      printf("\nEnter an integer:  ");
      scanf("%d", &num);
      printf("Value %f * 2 to power of %d = %2.2f",
              reply,num,ldexp(reply,num));
      break;
          }

          case 13:  {
      cont = NO;
      printf("Exiting from menu.\n");
      exit(0);
          }
          default:  {
      printf("Invalid choice. Try again.");
          }
        }
        clear_kb();
      }
      return 0;
    }


    int menu(void)  {
    /* Displays a menu and inputs user's selection */
      char buff[80];

      puts("\nEnter the number of any math operation");
      puts("1 - Arccosine");
      puts("2 - Arcsine");
      puts("3 - Arctangent");
```

```
  puts("4 - Arctangent of x/y");
  puts("5 - Cosine");
  puts("6 - Sine");
  puts("7 - Tangent");
  puts("8 - Exponent");
  puts("9 - Logarithm");
  puts("10 - Base 10 Log");
  puts("11 - Normalized Fraction");
  puts("12 - Exponential Value x * 2 to power of y");
  puts("13 - Exit");

  gets(buff);

  return(atoi(buff));
}

double dprompt(char *message)  {
  char *dstr;
  double dnum;

  printf("%s", message);
  gets(dstr);
  if (strlen(dstr) > 0)
    dnum = atof(dstr);
  else
    dnum = 0;
  return dnum;
}

/* Function: clear_kb() */
/* Purpose: This function clears the keyboard of extra */
/* characters. Returns: Nothing */

void clear_kb(void) {
  char junk[80];
  gets(junk);
}
```

*** Exercise 4**

Write a function that causes the program to pause for approximately 5 seconds using the time functions learned in this unit.

Answer

```
/* EXER1904.C Day 19 Exercise 4 */
/* Pause using time functions */

#include <time.h>

int pause5()  {
  time_t start;

  start = time(0);
  while (difftime(time(0),start) < 5)
    ;
  return 0;
}
```

*** Exercise 5**

Add the assert() function to the program from exercise four. The program should print a message if a negative value is entered.

Answer

```
/* EXER1905 Day 19 Exercise 5 */
/* MS-DOS application */
/* Use assert() function if attempting to square negative */
/* numbers */

#include <stdio.h>
#include <math.h>
#include <assert.h>

double x;
long lnum1, lnum2;

int main(void)  {

  printf("Enter a number: ");
  scanf("%lf", &x);

  printf("\n\nOriginal value: %lf", x);

  printf("\nCeil: %lf", ceil(x));
  printf("\nFloor: %lf", floor(x));
  assert (x >= 0);
  printf("\nSquare root: %lf", sqrt(x));

  printf("\nCosine: %lf", cos(x));

  return 0;
}
```

*** Exercise 6**

Write a program that accepts 30 names and sorts them using qsort(). The program should print the sorted names.

Answer

```
/* EXER1906.C Day 19 Exercise 6 */
/* Sort 30 names and print them. */

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

#define MAX 30

int comp(const void *s1, const void *s2);
char *data[MAX], buf[80], *ptr, *key, **key1;
int count;

int main(void)  {

  /* Input a list of names. */


  printf("Enter %d names, pressing ENTER after each.\n",
          MAX);

  for (count = 0; count < MAX; count++)  {
    printf("Name %d: ", count + 1);
    gets(buf);
    data[count] = malloc(strlen(buf) + 1);
```

```
    strcpy(data[count], buf);
  }

  /* Sort the words. */

  qsort(data, MAX, sizeof(data[0]), comp);

  /* Display the sorted words. */

  for (count = 0; count < MAX; count++)
    printf("\n%d: %s", count + 1, data[count]);

  return 0;
}

int comp(const void *s1, const void *s2)  {

  return(strcmp(*(char **)s1, *(char **)s2));

}
```

*** Exercise 7**

Modify the program in exercise six so that if the user enters "QUIT", the program stops accepting input and sorts the entered values.

Answer

```
/* EXER1907.C Day 19 Exercise 7 */
/* Sort 30 names and print them. */

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

#define MAX 30

int comp(const void *s1, const void *s2);
char *data[MAX], buf[80], *ptr, *key, **key1;
int count, maxsize;

int main(void)  {

  /* Input a list of names. */

  printf("Enter %d names, pressing ENTER after each.\n",
         MAX);

  for (count = 0; count < MAX; count++)  {
    printf("Name %d: ", count + 1);
    gets(buf);

    /* End the loop if user enters 'quit'. */

   if (strcmp(buf,"QUIT") == 0 ||

      (strcmp(buf, "quit") == 0)) {
      break;
    }
    data[count] = malloc(strlen(buf) + 1);
    strcpy(data[count], buf);
  }
  maxsize = count;
```

```
  /* Sort the words. */

  qsort(data, maxsize, sizeof(data[0]), comp);

  /* Display the sorted words. */

  for (count = 0; count < maxsize; count++)
    printf("\n%d: %s", count + 1, data[count]);

  return 0;
}

/* Comparison function. */

int comp(const void *s1, const void *s2)  {

  return(strcmp(*(char **)s1, *(char **)s2));

}
```

*** Exercise 8**

Refer to Day 15, "More on Pointers," 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().

Answer

```
/* EXER198.C Day 19 Exercise 8 */
/* Compare brute force sort with qsort */

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

#define MAXLINES 100

int get_lines(char *lines[], char *arr[]);
void sort(char *p[], int n);
void prtstr(char *p[], int n, char *s);
int comp(const void *s1, const void *s2);
time_t time1, time2;
double bf_time, qs_time;
char *lines[MAXLINES];
char *arr[MAXLINES];
int numlines,x;

int main(void)  {

  /* Read in the lines from the keyboard. */

  numlines = get_lines(lines,arr);
  if (numlines < 0)  {
    puts("Memory allocation error");
    exit(-1);
  }

  if (numlines > 0)  {

    /* Determine time for brute force sort */

    time1 = time(0);
    sort(lines,numlines);
```

```
    time2 = time(0);
    prtstr(lines,numlines,"Results of Brute Force Sort");
    bf_time = difftime(time2,time1);
    printf("\nTotal time for sort: %f\n",bf_time);

    /* Determine time for qsort */

    time1 = time(0);
    qsort(arr,numlines, sizeof(arr[0]), comp);
    time2 = time(0);
    prtstr(arr,numlines,"\n\nResults of QSort");
    qs_time = difftime(time2,time1);
    printf("\nTotal time for sort: %f\n",qs_time);

    /* Report on difference */

    printf("Number of items sorted: %d\n",numlines);
    if (qs_time > bf_time)
      printf("\nQuick Sort time is longer\n");
    else
      if (bf_time > qs_time)
        printf("\nBrute Force Sort time is longer\n");
    else
      printf("\nSort times are approximately the same\n");

  }
  return 0;
}

int get_lines(char *lines[], char *ptr[])  {
  int n = 0;
  char buffer[80];

  puts("Enter one line at a time;"
       " enter a null line when done:\n");
  while((n < MAXLINES) && (gets(buffer) != 0) &&
        (buffer[0] != '\0'))  {
    if ((lines[n] = (char *)malloc(strlen(buffer)+1))
         == NULL)
      return -1;
    strcpy(lines[n],buffer);
    if ((ptr[n] = (char *)malloc(strlen(buffer)+1))
         == NULL)
      return -1;
    strcpy(ptr[n++],buffer);
  }
  return n;
}

void sort(char *p[], int n)  {
  int a, b;
  char *x;

  for (a = 1; a < n; a++)  {
    for (b = 0; b < (n - 1); b++)  {
      if (strcmp(p[b], p[b + 1]) > 0)  {
  x = p[b];
  p[b] = p[b + 1];
  p[b + 1] = x;
      }
    }
  }
}

void prtstr(char *p[], int n, char *s)  {
  int count;

  printf("%s\n", s);
```

```
  for (count = 0; count < n; count++)
    printf("\n%s ", p[count]);
}

int comp(const void *s1, const void *s2)  {

  return(strcmp(*(char **)s1, *(char **)s2));
}
```

Topic 5.8: Day 19 Summary

Exploring the Function Library
This unit explored 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.

Unit 6. Day 20 Odds and Ends

This unit covers some loose ends about C programming topics not discussed in earlier units.

Today, you learn

- Type conversions.
- Allocating and freeing memory storage.
- Command-line arguments.
- Operating on bits.
- Bit fields in structures.

Topic 6.1: Type Conversions

Automatic and Explicit Conversions
All of C's data objects have a specific type. A numeric variable can be an int or a float, a pointer can be a pointer to double or char, and so on. You often mix different types in expressions and statements. What happens in such cases? Sometimes C automatically handles the different types and you need not be concerned. At other times you must make an explicit conversion of one data type to another. You saw this in earlier units when you had to convert or *cast* a type void pointer to a specific type. The following sections cover C's automatic and explicit conversions.

Topic 6.1.1: Automatic Type Conversions

Automatic Type Conversions
As their name implies, automatic type conversions are performed automatically by the C compiler without any need for you to do anything. However, you should be aware of what is going on so that you can understand how C evaluates expressions.

Components with the Same Type
When a C expression is evaluated, the resulting value has a data type. If all components of the expression have the same type, the resulting type is the same. However, char and short data types are converted to int types before arithmetical evaluation can take place. An expression containing all char types or all short types results in a value of type int. Click the Example link.

Example

Components with Different Types
What if the components of an expression have different types? The expression has the same type as the most comprehensive component. From least comprehensive to most comprehensive, the numerical data types are char, int, long, float, and double. Click the Example link.

Example

If x and y are both type int, the expression

```
x + y
```

is type int also.

Thus, an expression containing an int and a char is type int, an expression containing a long and a float is type float, and so on.

Promotion
Within expressions, individual operands are *promoted* as necessary to match the associated operands in the expression. Operands are promoted, in pairs, for each binary operator in the expression. Promotion is not needed, of course, if both operands are the same type. If they're not, promotion follows these rules:

- If either operand is a double, the other operand is promoted to type double.
- If either operand is a float, the other operand is promoted to type float.
- If either operand is a long, the other operand is converted to type long.

Click the Example link.

Example

If x is an int and y is a float, evaluating the expression x/y causes x to be promoted to type float. This does not mean that the type of the variable x is changed. It means that a type float copy of x is created and used in the expression evaluation. The value of the expression is, as you just learned, type float.

Promotions with the Assignment Operator
Promotions also occur with the assignment operator. An expression on the right side of an assignment statement is always promoted to the type of the data object on the left side of the assignment operator.

"Demotion"
Note that this might cause a "demotion" rather than a promotion. Click the Example link.

Example

If f is a type float and i is a type int, in the assignment statement

```
f = i;
```

i is promoted to type float. In contrast, the assignment statement

```
i = f;
```

causes f to be demoted to type int. Its fractional part is lost on assignment to i. Remember that f itself is not changed at all; promotion affects only a copy of the value.

Floating-Point Types
When a floating point number is converted to an integer type, the fractional part is lost. When an integer type is converted to a floating-point type, the resulting floating-point value might not match the integer value. This is because the floating-point format used internally by the computer cannot accurately represent every possible integer number.

Topic 6.1.2: Explicit Conversions Using Type Casts

Casts
A *type cast* uses the cast operator explicitly to control type conversions in your program. A type cast consists of a type name, in parentheses, before an expression. Casts can be performed on arithmetic expressions and on pointers.

Casting
Casting an arithmetic expression tells the compiler to represent the value of the expression in a certain way. In effect, a cast is similar to a promotion, which was discussed earlier. However, a cast is under your control and not the compiler's. Click the Example link.

Example

Usage
When would you use a type cast with an arithmetic expression? The most common use is to avoid losing the fractional part of the answer in an integer division.

The program in Listing 20.1 illustrates this. You should compile and run the program.

DO use a cast to promote or demote variable values.

DON'T use a cast just to prevent a compiler warning. You might find that using a cast gets rid of a warning, but before removing the warning this way, be sure you understand why you are getting the warning.

Listing 20.1: An Integer Division Loses the Fractional Part of the Answer

Code

```
 1: /* LIST2001.c: Day 20 Listing 20.1 */
 2: /* Demonstrates how integer division loses the */
 3: /* fractional part of the answer. */
 4:
 5: #include <stdio.h>
 6:
 7: int main(void) {
 8:   int i1 = 100, i2 = 40;
 9:   float f1;
10:
11:   f1 = i1/i2;
12:
13:   printf("%lf", f1);
14:   return 0;
15: }
```

Output

```
2.000000
```

Description

The "answer" displayed by the program is 2.000000. But 100/40 evaluates to 2.5. What happened?

The expression `i1/i2` on line 11 contains two type int variables. By the rules explained earlier in this unit, the value of the expression is therefore type int itself. As such, it can represent only whole numbers, so the fractional part of the answer is lost.

You might think that assigning the result of `i1/i2` to a type float variable promotes it to type float. This is correct, but now it's too late; the fractional part of the answer is already gone.

To avoid this sort of inaccuracy, you must cast one of the type int variables to type float. If one of the variables is cast to type float, the previous rules tell you the other variable is promoted automatically to type float and the value of the expression is also type float. The fractional part of the answer is thus preserved. To demonstrate this, change line 11 in the source code so that the assignment statement reads `f1 = (float)i1/i2;` Then the program displays the correct answer.

For example, if i is a type int, the expression

```
(float)i
```

casts i to type float. In other words, the program makes an internal copy of the value of i in floating point format.

Casting Pointers

You have already been introduced to the casting of pointers. As you saw on Day 18, "Getting More from Functions," a type void pointer is a generic pointer; it can point to anything. Before you can use a void pointer, you must cast it to the proper type. Note that you do not need to cast a pointer to assign a value to it or to compare it with NULL. However, you must cast it before dereferencing it or performing pointer arithmetic with it. For more details on casting void pointers, review Day 18, "Getting More from Functions."

Topic 6.2: Allocating Memory Storage Space

Dynamic Memory Allocation

The C library contains functions for allocating memory storage space at runtime, a process called *dynamic memory allocation*.

Advantages Over Explicit Allocation

Advantages Over Explicit Allocation
This technique can have significant advantages over explicitly allocating memory in the program source code (such as declaring an array). With the latter method, you must know when you are writing the program exactly how much memory is needed. Dynamic memory allocation allows the program to react, while it is executing, to demands for memory, such as user input.

STDLIB.H
All the functions for handling dynamic memory allocation require the header file STDLIB.H.

Return Value
Note that all allocation functions return a type void pointer. Such a pointer must be cast to the appropriate type before being used.

Topic 6.2.1: malloc()

Description
In earlier units, you learned how to use the malloc() library function to allocate storage space for strings. The malloc() function is not limited to strings, of course; it can allocate space for any storage need. This function allocates memory by the byte.

Syntax
Recall that malloc()'s prototype is

```
void *malloc(size_t num);
```

Arguments and Return Value
The argument size_t is defined in STDLIB.H as unsigned. The malloc() function allocates num bytes of storage space and returns a pointer to the first byte. The function returns NULL if the requested storage space could not be allocated or if num == 0. Review the section on malloc() on Day 10, "Characters and Strings," if you're still a bit rusty on its operation.

Topic 6.2.2: calloc()

Description
The calloc() function also allocates memory, but rather than allocating a group of bytes as malloc() does, calloc() allocates a group of objects.

Syntax
The function prototype is

```
void *calloc(size_t num, size_t el-size);
```

Arguments and Return Value
Remember that size_t is a synonym for unsigned on most compilers. The argument num is the number of objects to allocate, and el-size is the size (in bytes) of each object. If allocation is successful, all the allocated memory is cleared (set to 0) and the function returns a pointer to the first byte. If allocation fails or if either num or el-size is 0, the function returns NULL.

The program in Listing 20.2 illustrates the use of calloc().

Listing 20.2: Using the calloc() Function to Allocate Memory Storage Space Dynamically

```
Code  1: /* LIST2002.c: Day 20 Listing 20.2 */
      2: /* Demonstrates using the calloc() function to */
      3: /* allocate memory storage space dynamically. */
      4:
      5: #include <stdlib.h>
      6: #include <stdio.h>
```

```
 6: #include <stdio.h>
 7:
 8: int main(void) {
 9:   unsigned num;
10:   int *ptr;
11:
12:   printf("Enter the number of type int"
13:          " to allocate: ");
14:   scanf("%d", &num);
15:
16:   ptr = calloc(num, sizeof(int));
17:
18:   if (ptr != NULL)
19:     puts("Memory allocation was successful.");
20:   else
21:     puts("Memory allocation failed.");
22:   return 0;
23: }
```

Output

```
>list2002
Enter the number of type int to allocate: 100
Memory allocation was successful.

>list2002
Enter the number of type int to allocate: 99999999
Memory allocation failed.
```

Description

This program prompts for a value on lines 12 – 14. The number determines how much space is allocated. The program attempts to allocate enough memory (line 16) to hold the specified number of int variables. If the allocation fails, the return value from calloc() is NULL; otherwise, it's a pointer to the allocated memory. In the case of this program, the return value from calloc() is placed in the int pointer, ptr. An if statement on lines 18 – 21, checks the status of the allocation based on ptr's value and prints an appropriate message.

Enter different values and see how much memory can be successfully allocated. The maximum amount depends, to some extent, on your system configuration. On some systems allocating space for 25,000 occurrences of type int is successful, whereas 30,000 fails.

Topic 6.2.3: Realloc()

Description
The realloc() function changes the size of a block of memory previously allocated with malloc() or calloc().

Syntax
The function prototype is

```
void *realloc(void *mem-ptr, size_t num);
```

Arguments and Return Value
The mem-ptr argument points to the original block of memory. The new desired size, in bytes, is specified by num.

Return Values for realloc()

- If sufficient space exists to expand the memory block pointed to by mem-ptr, the additional memory is allocated and the function returns mem-ptr.
- If sufficient space does not exist to expand the current block, a new block of the size for num is allocated and existing data is copied from the old block to the beginning of the new block. The old block is freed, and the function returns a pointer to the new block.
- If the mem-ptr argument is NULL, the function acts like malloc(), allocating a block of num bytes and returning a pointer to it.

Return Values for realloc()

- If the argument num is 0, the memory that mem-ptr points to is freed and the function returns NULL.
- If memory is insufficient for the reallocation (either expanding the old block or allocating a new one), the function returns NULL and the original block is unchanged.

The use of realloc() is demonstrated by the program in Listing 20.3.

Listing 20.3: Using realloc() to Increase the Size of a Block of Dynamically Allocated Memory

Code

```
 1: /* LIST2003.c: Day 20 Listing 20.3 */
 2: /* Using realloc() to change memory allocation. */
 3:
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6: #include <string.h>
 7:
 8: int main(void) {
 9:   char buf[80], *message;
10:
11:   /* Input a string. */
12:
13:   puts("Enter a line of text.");
14:   gets(buf);
15:
16:   /* Allocate the initial block and copy the */
17:   /* string to it. */
18:
19:   message = realloc(NULL, strlen(buf) + 1);
20:   strcpy(message, buf);
21:
22:   /* Display the message. */
23:
24:   puts(message);
25:
26:   /* Get another string from the user. */
27:
28:   puts("Enter another line of text.");
29:   gets(buf);
30:
31:   /* Increase the allocation, then concatenate */
32:   /* the string to it. */
33:
34:   message = realloc(message,
35:   (strlen(message) + strlen(buf) + 1));
36:   strcat(message, buf);
37:
38:   /* Display the new message. */
39:
40:   puts(message);
41:   return 0;
42: }
```

Output

```
Enter a line of text.
This is the first line of text.
This is the first line of text.
Enter another line of text.
This is the second line of text.

This is the first line of text.This is the second line of text.
```

Description

This program gets an input string on line 14. This string is read into an array of characters called buf. This value is then copied into a memory location pointed to by message (line 20). message was allocated using realloc() on line 19. realloc() was called even though there had not been a previous allocation. By passing NULL as the first parameter, realloc() knows that this is a first allocation.

	Line 29 gets a second string in the buf buffer. This string is concatenated to the string already held in message. Because message is just big enough to hold the first string, it needs to be reallocated to make room to hold both the first and second strings. This is exactly what line 34 does. The program concludes by printing out the final concatenated string.

Topic 6.2.4: Free()

Description
When you allocate memory with either malloc() or calloc(), it is taken from the dynamic memory pool available to your program. This pool is sometimes called the *heap* and it is finite. When your program finishes using a particular block of allocated memory, you might want to "deallocate," or free, the memory to make it available for future allocations. To free memory that was allocated dynamically, use free().

Syntax
Its prototype is

```
void free(void *mem-ptr);
```

Arguments
The free() function releases the memory pointed to by mem-ptr. This memory must have been allocated with malloc(), calloc(), or realloc(). If mem-ptr is NULL, free() does nothing. The free() function returns no value.

Listing 20.4 demonstrates the free() function.

DO free allocated memory when you are done with it.

DON'T assume that a call to malloc(), calloc(), or realloc() was successful. In other words, always check to see that the memory was indeed allocated.

Listing 20.4. Using free() to Release Previously Allocated Dynamic Memory

```
Code  1: /* LIST2004.c: Day 20 Listing 20.4 */
      2: /* Using free() to release allocated dynamic */
      3: /* memory. */
      4:
      5: #include <stdio.h>
      6: #include <stdlib.h>
      7: #include <string.h>
      8:
      9: #define BLOCKSIZE 30000
     10:
     11: int main(void) {
     12:   void *ptr1, *ptr2;
     13:
     14:   /* Allocate one block. */
     15:
     16:   ptr1 = malloc(BLOCKSIZE);
     17:
     18:   if (ptr1 != NULL)
     19:     printf("\nFirst allocation of %d bytes "
     20:            "successful.", BLOCKSIZE);
     21:   else {
     22:     printf("\nAttempt to allocate %d bytes "
     23:            "failed.", BLOCKSIZE);
     24:     exit(1);
     25:   }
     26:
     27:   /* Try to allocate another block. */
     28:
```

```
29:   ptr2 = malloc(BLOCKSIZE);
30:
31:   if (ptr2 != NULL) {
32:     /* If allocation successful, print message */
33:     /* and exit. */
34:
35:     printf("\nSecond allocation of %d bytes "
36:            "successful.", BLOCKSIZE);
37:     exit(0);
38:   }
39:
40:   /* If not successful, free the first block and */
41:   /* try again. */
42:
43:   printf("\nSecond attempt to allocate %d bytes "
44:          "failed.", BLOCKSIZE);
45:   free(ptr1);
46:   printf("\nFreeing first block.");
47:
48:   ptr2 = malloc(BLOCKSIZE);
49:
50:   if (ptr2 != NULL)
51:     printf("\nAfter free(), allocation of %d "
52:            "bytes successful.", BLOCKSIZE);
53:   return 0;
54: }
```

Output

```
First allocation of 30000 bytes successful.

Second allocation of 30000 bytes successful.
```

Description

This program tries to dynamically allocate two blocks of memory. It uses the defined constant BLOCKSIZE to determine how much to allocate. Line 16 does the first allocation using malloc(). Lines 18 – 25 check the status of the allocation by checking to see whether the return value was equal to NULL. A message is displayed stating the status of the allocation. If the allocation failed, the program exits.

Line 29 tries to allocate a second block of memory, again checking to see whether the allocation was successful (lines 31 – 38). If the second allocation was successful, a call to exit() ends the program. If it was not successful, a message states that the attempt to allocate memory failed. The first block is then freed with free() (line 45), and a new attempt is made to allocate the second block.

You might need to modify the value of the symbolic constant BLOCKSIZE. On some systems the value of 30000 produces the following program output:

```
First allocation of 30000 bytes successful.
Second attempt to allocate 30000 bytes failed.
Freeing first block.
After free(), allocation of 30000 bytes successful.
```

Topic 6.3: Using Command-Line Arguments

Command-Line Arguments

Your C program can access arguments passed to the program on the command line. This refers to information entered after the program name when you start the program. If starting a program from the C> prompt, you could type

```
C>progname arg1 arg2 ... argn
```

Passing Arguments to main()

The various arguments (arg1 and so on) can be retrieved by the program during execution. You can consider this information as arguments passed to the program's main() function. Command-line arguments can be retrieved only within main(). To do so, declare main() as follows:

within main(). To do so, declare main() as follows:

```
main(int argc, char *argv[]) {
  /* Statements go here */
}
```

`argc` and `argv` are described on the next page.

argc and argv
The following table describes argc and argv[], which are conventionally the command-line arguments for main(). Click the Tip button after reading the table.

Parameter	Description
`argc`	Is an integer giving the number of command-line arguments available. This value is always at least 1 because the program name is counted.
`argv[]`	Is an array of pointers to strings. The valid subscripts for this array are 0 through `argc - 1`. The pointer `argv[0]` points to the program name (including path information), `argv[1]` points to the first argument that follows the program name, and so on.

DO use argc and argv as the variable names for the command-line arguments for main(). Most C programmers are familiar with these names.

DON'T assume that program users have entered the command-line parameters. Check to be sure they did. If they did not, display a message stating how to use the program.

Arguments Including Whitespace
The command line is divided into discrete arguments by any whitespace. If you need to pass an argument that includes a space, enclose the entire argument in double quotation marks. Click the Example link.

Example

Required and Optional Arguments
Command-line arguments fall into two categories: some are required because the program can't operate without them, whereas others can be optional "flags" that instruct the program to act in a certain way. Click the Example link.

Example

Listing 20.5: Passing Command-Line Arguments to main()

Code

```
 1: /* LIST2005.c: Day 20 Listing 20.5 */
 2: /* Accessing command-line arguments. */
 3:
 4: #include <stdio.h>
 5:
 6: int main(int argc, char *argv[]) {
 7:   int count;
 8:
 9:   printf("Program name: %s\n", argv[0]);
10:
11:   if (argc > 1) {
12:     for (count = 1; count < argc; count++)
13:       printf("Argument %d: %s\n", count,
14:               argv[count]);
15:   }
16:   else
17:     puts("No command line arguments entered.");
18:   return 0;
19: }
```

Output

```
F:\BOOK\X>list2005
```

Output	E:\BOOK\X>list2005 Program name: E:\BOOK\X\LIST2005.EXE No command line arguments entered. E:\BOOK\X>list2005 first second 3 4 Program name: E:\BOOK\X\LIST2005.EXE Argument 1: first Argument 2: second Argument 3: 3 Argument 4: 4
Description	This program does no more than print the command-line parameters entered by the user. Notice that line 6 uses the argc and argv parameters shown previously. Line 9 prints the one command-line parameter that you always have, the program name. Notice this is argv[0]. Line 11 checks to see whether there is more than one command-line parameter. Why more than one and not more than zero? Because the program name is first. If there are additional arguments, a for loop prints each to the screen (lines 12 – 14). Otherwise, an appropriate message is printed (line 17).

If you enter

```
C>progname arg1 "arg two"
```

arg1 is the first argument (pointed to by argv[1]) and arg two is the second (pointed to by argv[2]).

The program in Listing 20.5 demonstrates how to access command-line arguments.

Imagine a program that sorts the data in a file. If you write the program to receive the input filename from the command line, the name is required information. If the user forgets to enter the input filename on the command line, the program must somehow deal with the situation. The program could also look for the argument /r, which signals a reverse-order sort. This argument is not required; the program looks for it and behaves one way if it's found, another way if not.

Topic 6.4: Operating on Bits

The Bitwise Operators
The C bitwise operators enable you to manipulate the individual bits of integer variables. These operators can be used only with integer types: char, int, and long.

Binary Notation
Before continuing with this section, you should be familiar with *binary notation,* the way the computer internally stores integers. If you need some review on binary notation, refer to the section on "Binary and Hexadecimal Notation," before you continue.

Usage
The bitwise operators are most frequently used when your C program interacts directly with your system's hardware, a topic beyond the scope of this course. They do have other uses, however, so you should become familiar with them.

Topic 6.4.1: The Shift Operators

<< and >> Operators
Two shift operators shift the bits in an integer variable by a specified number of positions: the << operator shifts bits left, and the >> operator shifts bits right. The syntax for these binary operators is

```
x << n
x >> n
```

Bit Replacement
Each operator shifts the bits in x by n positions in the specified direction. For a right shift, zeros are placed in the n high order bits of the variable; for a left shift, zeros are placed in the n low order bits of the variable. Click the Example link.

Example

Here are a few examples:

Binary 00001100 (decimal 12) right-shifted by 2 evaluates to binary 00000011 (decimal 3).
Binary 00001100 (decimal 12) left-shifted by 3 evaluates to binary 01100000 (decimal 96).
Binary 00001100 (decimal 12) right-shifted by 3 evaluates to binary 00000001 (decimal 1).
Binary 00110000 (decimal 48) left-shifted by 3 evaluates to binary 10000000 (decimal 128).

Multiplication and Division
Under certain circumstances, the shift operators can be used to multiply and divide a value by a power of 2. Left-shifting an integer by n places has the same effect as multiplying it by 2n, and right-shifting an integer has the same effect as dividing it by 2n. The results of a left-shift multiplication are accurate only if there is no overflow, that is, if no bits are "lost" by being shifted out of the high order positions. A right-shift division is an integer division, with any fractional part of the result lost.

The program in Listing 20.6 demonstrates the shift operators.

Listing 20.6: Using the Shift Operators

Code

```
 1: /* LIST2006.c: Day 20 Listing 20.6 */
 2: /* Demonstrating the shift operators. */
 3:
 4: #include <stdio.h>
 5:
 6: int main(void) {
 7:   unsigned char y, x = 255;
 8:   int count;
 9:
10:   printf("Decimal\t\tshift left by\tresult\n");
11:
12:   for (count = 1; count < 8; count++) {
13:     y = x << count;
14:     printf("%d\t\t%d\t\t%d\n", x, count, y);
15:   }
16:   return 0;
17: }
```

Output

```
Decimal         shift left by   result
255             1               254
255             2               252
255             3               248
255             4               240
255             5               224
255             6               192

255             7               128
```

Topic 6.4.2: The Bitwise Logical Operators

&, | and ^ Operators

Three bitwise logical operators are used to manipulate individual bits in an integer data type. These operators have names similar to the TRUE/FALSE logical operators that you learned about in earlier units, but their operations differ from each other. The bitwise logical operators are listed here. These are all binary operators, setting bits in the result to 1 or 0 depending on the bits in the operands.

Following are examples of how these operators work. Click the Example links.

Example of AND
Example of Inclusive OR
Example of Exclusive OR

Table 20.1: The Bitwise Logical Operators		
Operator	**Description**	**Action**
&	AND (also known as Bitwise AND)	Sets a bit in the result to 1 only if the corresponding bits in both operands are 1; otherwise, the bit is set to 0. The AND operator is used to turn off one or more bits in a value.
|	Inclusive OR (also known as Bitwise OR)	Sets a bit in the result to 0 only if the corresponding bits in both operands are 0; otherwise, the bit is set to 1. The OR operator is used to turn on one or more bits in a value.
^	Exclusive OR (also known as Bitwise XOR)	Sets a bit in the result to 1 if the corresponding bits in the operands are different (one 1, the other 0); otherwise, the bit is set to 0.

```
11110000
01010101
--------
01010000
```

```
11110000
01010101
--------
11110101
```

```
11110000
01010101
--------
10100101
```

Topic 6.4.3: The Complement Operator

The ~ Operator
The final bitwise operator is the complement operator, (~). This is a unary operator. It is also known as the bitwise inversion operator.

Action
Its action is to reverse every bit in its operand, changing all 0s to 1s and vice versa.

For example, ~254 (binary 11111110) evaluates to 1 (binary 00000001).

Topic 6.5: Bit Fields in Structures

Usage
The final bit-related topic is the use of *bit fields* in structures. On Day 11, "Structures," you learned how you can define your own data structures, customizing them to fit the data needs of your program. By using bit fields, you can accomplish even greater customization and save memory space as well.

Bit Fields
A bit field is a structure member that contains a specified number of bits. You can declare a bit field to contain 1 bit, 2 bits, or whatever number of bits are required to hold the data stored in the field. What advantage does this provide? Click the Example link to find out.

Example

Bit Field Values
Bit fields are not limited to YES/NO values. Click the Example link and then the Tip button.

Example

DO use defined constants YES and NO, or TRUE and FALSE, when working with bits. These are much easier to read and understand than 1 and 0.

DON'T define a bit field that takes 8 or 16 bits. These are the same as other available variables such as type char or int.

Suppose that you are programming an employee database program that keeps records on your company's employees. Many of the items of information the database stores are of the "YES" or "NO" variety, such as: "Is the employee enrolled in the dental plan?" or "Did the employee graduate from college?" Each piece of YES/NO information can be stored in a single bit, with 1 representing YES and 0 representing NO.

Using C's standard data types, the smallest type you could use in a structure is a type char. You could indeed use a type char structure member to hold YES/NO data, but seven of the char's eight bits would be wasted space. By using bit fields, you could store eight YES/NO values in a single char.

Continuing with this database example, imagine that your firm has three different health insurance plans. Your database needs to store data about the plan, if any, in which each employee is enrolled. You could represent no health insurance by 0 and the three plans by values of 1-3. A bit field containing two bits is sufficient because two binary bits can represent values of 0 through 3. Likewise, a bit field containing three bits could hold values in the range 0-7, four bits could hold values in the range 0-15, and so on.

Defining a Structure with Bit Fields
Bit fields are named and accessed like regular structure members. All bit fields have type unsigned int, and the size of the field (in bits) is specified by following the member name with a colon and the number of bits. Click the Example link.

Example

To define a structure with a 1-bit member named dental, another 1-bit member named college, and a 2-bit member named health, write the following:

```
struct emp_data {
    unsigned dental      : 1;
```

```
    unsigned college     : 1;
    unsigned health      : 2;
...
};
```

The ellipses (`...`) indicate space for other structure members, which can be bit fields or fields made up of regular data types. Note that bit fields must always be declared first, before other structure members.

Accessing the Bit Fields
To access the bit fields, use the structure member operator, just as you do with any structure member. Click the Example link.

Example

Your code is clearer, of course, if you use symbolic constants YES and NO with values of 1 and 0. In any case, you treat each bit field as a small, unsigned integer with the given number of bits. The range of values that can be assigned to a bit field with n bits is from 0 to 2n-1. If you try to assign an out-of-range value to a bit field, the compiler does not report an error, but you do get unpredictable results.

For the example, you can expand the structure definition to something more useful.

```
struct emp_data {
    unsigned dental      : 1;
    unsigned college     : 1;
    unsigned health      : 2;
    char fname[20];
    char lname[20];
    char ssnumber[10];
};
```

Then, declare an array of structures:

```
struct emp_data workers[100];
```

To assign values to the first array element, you write something like the following:

```
workers[0].dental = 1;
workers[0].college = 0;
workers[0].health = 2;
strcpy(workers[0].fname, "Mildred");
```

Topic 6.6: Day 20 Q&A

Questions & Answers
Here are some questions to help you review what you have learned in this unit.

Question 1
Do I really gain that much by using bit fields?

Answer
Yes, you can gain quite a bit with bit fields. (Pun intended!) Consider a circumstance similar to the example in this unit wherein a file contains information from a survey. People are asked to agree or disagree with the questions given. Assume you ask 100 questions of 10,000 people. If the answers are stored in characters as T or F, you need 10,000 times 100 bytes of storage (assuming a character is 1 byte). This is 1,000,000 bytes of storage. If you use bit fields instead, you need 100 divided by 8 bits for each record, or 13 bytes. This times the 10,000 people is 130,000 bytes of data. 130,000 is significantly less than 1 million.

Question 2
Why would I ever need to free memory?

Answer
When you first learn to use C, your programs are not very big. As your programs grow, their use of memory also grows. You should try to write your programs to use memory as efficiently as possible. When you are done with memory, you should release it. If you write programs that work on a multitasking environment, other applications might need memory that you aren't using.

Question 3
What happens if I reuse a string without calling realloc()?

Answer
You don't need to call realloc() if the string you are using was allocated enough room. Call realloc() when your current string is not big enough. Remember, the C compiler enables you to do almost anything. You can overwrite the original string with a bigger string. The problem is, you also overwrite whatever was after the string. This could be nothing, or it could be vital data. If you need a bigger allocated section of memory, call realloc().

Question 4
Is the following header also acceptable when using main() with command-line parameters?

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

Answer
You can probably answer this one on your own. This declaration uses a pointer to a character pointer instead of a pointer to a character array . Because an array is a pointer, this definition is virtually the same as the one presented in this unit. This declaration is also commonly used. (See Day 8, "Numeric Arrays," and Day 10, "Characters and Strings," for more details.)

Topic 6.7: Day 20 Think About Its

Think About Its
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

```
struct date{
    unsigned month : 4;
    unsigned day   : 5;
    unsigned year  : 7;
}
```

Sample 1

```
struct quiz_answers {
char student_name[15];
unsigned answer1   : 1;
unsigned answer2   : 1;
unsigned answer3   : 1;
unsigned answer4   : 1;
unsigned answer5   : 1;

     }
```

Sample 2

```
struct quiz_answers {
    unsigned answer1   : 1;
    unsigned answer2   : 1;
    unsigned answer3   : 1;
    unsigned answer4   : 1;
    unsigned answer5   : 1;
```

```
    char student_name[15];
}
```

Topic 6.8: Day 20 Try Its

Try Its
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Write a malloc() command that allocates memory for 1000 longs.

Answer

```
long *ptr;
ptr = malloc(1000 * sizeof(long));
```

*** Exercise 2**
Write a calloc() command that allocates memory for 1000 longs.

Answer

```
long *ptr;
ptr = calloc(1000, sizeof(long));
```

*** Exercise 3**
BUG BUSTERS: Is anything wrong with the following code?

```
void func() {
    int number1 = 100, number2 = 3;
    float answer;

    answer = number1 / number2;

    printf("%d/%d = %lf", number1, number2, answer)
}
```

Answer

There is nothing wrong with the given example; however, the result may not be the one expected. Because number1 and number2 are both integers, the result of their division will be an integer, thus losing the decimal places. In order to get the decimal places in the answer, you need to cast the expression to type float.

```
answer = (float) number1/number2;
```

*** Exercise 4**
Write a program that takes two filenames as command-line parameters. The program should copy the first file into the second file. (See Day 16, "Using Disk Files," if you need help working with files.)

Answer

```
/* EXER2004  Day 20 Exercise 4 */
/* MS-DOS application */
/* Copies a file into a second file, using command */
/* line arguments */

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

#define BUFSIZE 100

char buf[BUFSIZE];
char filename[20];
char copyname[20];
FILE *fpcopy;
FILE *fp;

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

  if (argc < 3)  {
    printf("\nNot enough arguments.\n");
    exit(1);
  }

  strcpy(filename,argv[1]);
  strcpy(copyname,argv[2]);

  /* Open the file for reading. */

  if ((fp = fopen(filename,"r")) == NULL)  {
    fprintf(stderr, "Error opening file.");
    exit(1);
  }

  /* Open the 2nd file for writing. */

  if ((fpcopy = fopen(copyname,"w")) == NULL)  {
    fprintf(stderr, "Error opening file.");
    exit(1);
  }

  /* Copy the 1st file into the 2nd */

  while (1)  {
    fgets(buf, BUFSIZE, fp);
    if (!feof(fp))
      fputs(buf,fpcopy);
    else
      break;
  }

  fclose(fpcopy);
  fclose(fp);

  return 0;
}
```

*** Exercise 5**

Write a function that returns the maximum number of bytes available for allocation at the moment. (Hint: The function calls both malloc() and free() multiple times.)

Answer

```
/* EXER2005  Day 20 Exercise 5 */
/* Determines maximum number of bytes available in memory */

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

#define BLOCKSIZE  30000


double howmuch()  {

  double numbytes;
  char *ptr;

  /* Allocate minimum amount of storage */
  if ((ptr = malloc(BLOCKSIZE)) == NULL)  {
    return(0);
  }

  numbytes = 0;

  /* Allocate additional blocks of storage until error */
  /* received */
  while (ptr != NULL)  {
    free(ptr);
    numbytes = numbytes + BLOCKSIZE;
    ptr = malloc((size_t)numbytes);
  }

  return (numbytes);
}
```

*** Exercise 6**

Write a program that accepts two or more floating point values as command-line arguments, and then displays their sum.

Answer

```
/* EXER2006  Day 20 Exercise 6 */
/* MS-DOS Application */
/* Accepts 2 floating point numbers and displays their sum */

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

double fnum1, fnum2;

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


  if (argc < 3)  {
    printf("\nNot enough arguments.\n");
    exit(1);
  }

  fnum1 = atof(argv[1]);
  fnum2 = atof(argv[2]);

  printf("Sum of the two values is %f\n", fnum1 + fnum2);
```

```
  return 0;
}
```

*** Exercise 7**
Write a program that copies the time from the tm structure to the bit field structure described in quiz question six.

Answer

```
/* EXER2007  Day 20 Exercise 7 */
/* Store day of week in a bit field structure. */

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

struct daynumbr  {
  unsigned day  :  3;
};
struct daynumbr datafld;
struct tm *ptr;
time_t now;

int main(void)  {

  now = time(0);
  ptr = localtime(&now);

  datafld.day = ptr->tm_wday + 1;

  printf("datafld.day contains %x\n", datafld.day);
  return 0;
}
```

*** Exercise 8**
Write a program that uses each of the bitwise logical operators. The program should apply the bitwise operator to a number and then reapply it to the result. You should observe the output.

Answer

```
/* EXER2008  Day 20 Exercise 8 */
/* Apply bitwise logical operators to number & result. */

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

unsigned char x = 0xde;
unsigned char y = 0x81;
int interim, final;


int main(void)  {

  printf("Number acted on: %x\n", x);
  printf("Number supplying the bits: %x\n", y);

  /*  AND */

  interim = x & y;
```

```
  printf("& Interim result: %x\n", interim);
  final = interim & y;
  printf("& Final result: %x\n", final);

  /* Inclusive OR */

  interim = x | y;
  printf("| Interim result: %x\n", interim);
  final = interim | y;
  printf("| Final result: %x\n", final);

  /* Exclusive OR */

  interim = x ^ y;
  printf("^ Interim result: %x\n", interim);
  final = interim ^ y;
  printf("^ Final result: %x\n", final);

  return 0;
}
```

*** Exercise**

Write a program that displays the binary value of a number. For instance, if 3 is entered, the program should display 00000011. You probably need to use the bitwise operators to accomplish this.

Answer

```
/* EXER209  Day 20 Exercise 9 */
/* Display binary representation of a number */

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

char chardisp[9] = { "00000000" };
int count, num, len, x;
unsigned char c;
char buffer[80];
unsigned char ckbit = 0x80;

int main(void)  {

  puts("Enter a number: ");
  gets(buffer);
  printf("\nBinary representation: ");

  num = atoi(buffer);
  len = strlen(buffer);

  /* Do translation for each character retrieved from stdin */
  for (x = 0; x < len; x++)  {
    c = (unsigned char)num;
    /* Shift right to see if a 1-bit is in each position */
    for (count = 0; count < 8; count++)  {
      if ((c & (ckbit >> count)) > 0)
        chardisp[count] = '1';
    }
    printf("%s ", chardisp);
  }
  return 0;
}
```

Topic 6.9: Day 20 Summary

Allocating Memory Storage Space
This unit covered a variety of C programming topics. You learned how to allocate, reallocate, and free memory at runtime, commands that give you flexibility in allocating storage space for program data.

Explicit Conversions Using Type Casts
You also saw how and when to use type casts with variables and pointers.

Using Command-Line Arguments
In addition, you saw how your program can use argc and argv[ ] to access DOS command-line arguments.

Bit Fields in Structures
This unit also covered the ways in which a C program can manipulate individual bits. The bitwise operators enable you to manipulate the individual bits in integer variables, and you can use bit fields in structures to maximize the efficiency of data storage.

Unit 7. Day 21 Taking Advantage of Preprocessor Directives and More

This final unit covers some additional features of the C compiler.

Today, you learn

- Programming with multiple source-code files.

- Using the C preprocessor.

Topic 7.1: Programming with Multiple Source Files

Modular Programming
Until now, all your C programs have consisted of a single source-code file, exclusive of header files. A single source-

Until now, all your C programs have consisted of a single source code file, exclusive of header files. A single source code file is often all you need, particularly for small programs, but you can also divide the source code for a single program among two or more files, a practice called *modular programming*. Why would you want to do this? The following sections explain.

Topic 7.1.1: Advantages of Modular Programming

The Primary Reason
The primary reason to use modular programming is closely related to structured programming and its reliance on functions. As you become a more experienced programmer, you develop more general-purpose functions that you can use, not only in the program for which they were originally written, but in other programs as well. For example, you might write a collection of general-purpose functions for displaying information on the screen. By keeping these functions in a separate file, you can use them again in different programs that also display information on the screen. When you write a program that consists of multiple source-code files, each source file is called a *module*.

Topic 7.1.2: Modular Programming Techniques

The Main Module
A C program can have only one main() function. The module that contains the main() function is called the *main module*.

Secondary Modules
Other modules are called *secondary modules*. A separate header file is usually associated with each secondary module (you learn why later in the unit).

Example Program
For now, look at a few simple examples illustrating the basics of multiple module programming. Listings 21.1, 21.2 and 21.3 show the main module, the secondary module, and the header file, respectively, for a program that inputs a number from the user and displays its square.

Listing 21.1: SQUARE.C, the Main Module

Code

```
 1: /* LIST2101.c: Day 21 Listing 21.1 */
 2: /* square.c, the main module. */
 3: /* Inputs a number and displays its square. */
 4:
 5: #include <stdio.h>
 6: #include "calc.h"
 7:
 8: int main(void) {
 9:   int x;
10:
11:   printf("Enter an integer value: ");
12:   scanf("%d", &x);
13:
14:   printf("\nThe square of %d is %ld.", x, sqr(x));
15:   return 0;
16: }
```

Output

```
Enter an integer value: 100

The square of 100 is 10000.
```

Description

The main module SQUARE.C contains the main() function. This module also includes the header file CALC.H.

Listing 21.2: CALC.C, the Secondary Module

Code

```
 1: /* LIST2102.c: Day 21 Listing 21.2 */
 2: /* calc.c, the secondary module. */
```

```
3: /* Module containing calculation functions. */
4:
5: #include "calc.h"
6:
7: long sqr(int x) {
8:   return ((long)x * x);
9: }
```

Description The secondary module file, CALC.C, contains the definition of the sqr() function. The #include directive is used to include the header file CALC.H. Note that the header filename is enclosed in quotation marks rather than angle brackets. (You learn the reason for this later in the unit.)

Listing 21.3: CALC.H, the Header File for CALC.C

```
Code 1: /* LIST2103.c: Day 21 Listing 21.3 */
     2: /* calc.h, the header file for calc.c. */
     3:
     4: long sqr(int x);
```

Description The header file, CALC.H, contains the prototype for the sqr() function in CALC.C. Because any module that uses sqr() needs to know sqr()'s prototype, the module must include CALC.H.

Compiling and Linking the Final Executable Program
After you use your editor to create these three files, how do you compile and link the final executable program? Your compiler controls this for you. At the command line, enter

```
tcc square.c calc.c
```

where tcc is your compiler's command. This directs the compiler's components to perform the following tasks.

Tasks
This directs the compiler's components to perform the following tasks.

Step	Action
1	Compile SQUARE.C, creating SQUARE.OBJ (or SQUARE.O on a UNIX system). If it encounters any errors, the compiler displays descriptive error messages.
2	Compile CALC.C, creating CALC.OBJ (or CALC.O on a UNIX system). Again, error messages appear if they are needed.
3	Link SQUARE.OBJ, CALC.OBJ, and any needed functions from the standard library to create the final executable program SQUARE.EXE.

Topic 7.1.3: Module Components

What to Put in Each File
As you can see, the mechanics of compiling and linking a multiple-module program are quite simple. The only real question is what to put in each file. The following paragraphs give you some general guidelines.

Contents of Secondary Module
The secondary module should contain general utility functions, that is, functions that you might want to use in other programs. A common practice is to create one secondary module for each type of function; for example, KEYBOARD.C for your keyboard functions, SCREEN.C for your screen display functions, and so on. To compile and link more than two modules, list all source files on the command line.

```
tcc mainmod.c screen.c keyboard.c
```

Contents of Main Module
The main module should contain main(), of course, and any other functions that are program-specific (meaning they have no general utility).

Contents of Header File
There is usually one header file for each secondary module. Each file has the same name as the associated module, with the .H extension. In the header file, put

- Prototypes for functions in the secondary module.
- #define directives for any symbolic constants and macros used in the module.
- Definitions of any structures or external variables used in the module.

Because this header file might be included in more than one source file, you want to prevent portions of it from compiling more than once. You can do this by using the preprocessor directives for conditional compilation (discussed later in the unit).

Topic 7.1.4: External Variables and Modular Programming

Data Visibility
In many cases, the only data communication between the main module and the secondary module is through arguments passed to and returned from the functions. In this case, you don't need to take special steps regarding data visibility, but what about an external variable that needs to be visible in both modules?

The extern Keyword
Recall from Day 12, "Variable Scope," that an external variable is one declared outside of any function. An external variable is visible throughout the entire source code file in which it is declared. It is not, however, automatically visible in other modules. To make it visible, you must declare the variable in each module, using the extern keyword. Click the Example link.

Example

All extern variables have static duration and are visible to all functions in the module.

If you have an external variable declared in the main module as

```
float interest_rate;
```

you make interest_rate visible in a secondary module by including the following declaration in that module (outside of any function):

```
extern float interest_rate;
```

The extern keyword tells the compiler that the original declaration of interest_rate (the one that set aside storage space for it) is located elsewhere, but that the variable should be made visible in this module.

Usage
Figure 21.1 illustrates the use of the extern keyword in a multiple-module program. In Figure 21.1, the variable x is visible throughout all three modules. In contrast, y is visible only in the main module and secondary module 1.

Topic 7.1.5: Using .OBJ Files

Object Files
After you've written and thoroughly debugged a secondary module, you don't need to recompile it every time you use it in a program. Once you have the object file for the module code, all you need to do is link it with each program that uses the functions in the module.

When you compile a program, the compiler creates an object file that has the same name as the C source code file along with the .OBJ extension.

Object File: KEYBOARD.OBJ
Say, for example, you are developing a module called KEYBOARD.C and compiling it, along with the main module DATABASE.C, with the following command:

```
tcc database.c keyboard.c
```

The KEYBOARD.OBJ file is also on your disk. Click the Tip button.

DON'T try to compile multiple source files together if more than one module contains a main() function. You can have only one main().

DO create generic functions in their own source files. This way they can be linked into any other programs that need them.

DON'T always use the C source files when compiling multiple files together. If you compile a source file into an object file, recompile only when the file changes. This saves a great deal of time.

Linking KEYBOARD.OBJ
Once you know the functions in KEYBOARD.C work properly, you can stop compiling it every time you recompile DATABASE.C (or any other program that uses it), instead linking the existing object file. To do this, use this command:

```
tcc database.c keyboard.obj
```

The compiler then compiles DATABASE.C and links the resulting object file DATABASE.OBJ with KEYBOARD.OBJ to create the final executable file DATABASE.EXE.

Recompiling
This saves time because the compiler doesn't have to recompile the code in KEYBOARD.C. However, if you modify the code in KEYBOARD.C, you must recompile it. In addition, if you modify a header file, you must recompile all the modules that use it.

Topic 7.2: The C Preprocessor

Preprocessor
The *preprocessor* is a part of all C compiler packages. When you compile a C program, the preprocessor is the first compiler component that processes your program. In most C compilers, the preprocessor is part of the compiler program. When you run the compiler, it automatically runs the preprocessor.

Preprocessor Directives
The preprocessor changes your source code based on instructions, or *preprocessor directives,* in the source code. The output of the preprocessor is a modified source-code file that is then used as the input for the next compilation step. Normally, you never see this file because it is deleted automatically by the compiler after it is used. However, later in the unit, you learn how to look at this intermediate file. First, you need to look at the preprocessor directives, all of which begin with the # symbol.

Topic 7.2.1: The #define Preprocessor Directive

Usage
The #define directive has two uses: creating *symbolic constants* and creating macros.

Creating Substitution Macros
You learned about substitution macros (also known as object-like macros) on Day 3, "Numeric Variables and Constants," although the term used to describe them was *symbolic constants*. You create a *substitution macro* by using #define to replace text with other text. Click the Example link.

Example

Usage
The most frequent use for substitution macros is to create symbolic constants, explained on Day 3, "Numeric Variables and Constants." Click the Example link.

Example

To replace `text1` with `text2`, you write

```
#define text1 text2
```

This directive causes the compiler to go through the entire source code file, replacing every occurrence of `text1` with `text2`. The only exception occurs if `text1` is found within double quotation marks, in which case the compiler makes no change.

If your program contains the lines

```
#define MAX 1000

x = y * MAX;
z = MAX - 12;
```

after preprocessing, the source code reads

```
x = y * 1000;
z = 1000 - 12;
```

The effect is the same as using your editor's search-and-replace feature in order to change every occurrence of MAX to 1000.

Other Uses
Note that #define is not limited to creating symbolic numeric constants. You could write, for example,

```
#define ZINGBOFFLE printf


ZINGBOFFLE("Hello, world.");
```

although there is little reason to do so.

Terminology
You should also be aware that some authors refer to symbolic constants defined with #define as being *macros* themselves. (Symbolic constants are also called manifest constants.) However, in this course the word *macro* is reserved for the type of construction described next.

Function Macros Accept Arguments
You can also use the #define directive to create function macros. A *function macro* is a type of shorthand, using something simple to represent something more complicated. The reason for the "function" name is that this type of macro can accept arguments, just like a real C function does. One advantage of function macros is that their arguments are not type-sensitive. Therefore, you can pass any numeric variable type to a function macro that expects a numeric argument. Click the Example link.

Example

Macro Definition
The following preprocessor directive defines a macro named HALFOF that takes a parameter named value:

```
#define HALFOF(value) ((value)/2)
```

Macro Expansion
Whenever the preprocessor encounters the text `HALFOF(value)` in the source code, it replaces it with the definition text and inserts the argument as needed.

If the source code line is...	Then it is replaced by...
`result = HALFOF(10);`	`result = ((10)/2);`
`printf("%f",` `  HALFOF(x[1]+y[2]));`	`printf("%f",` `  ((x[1]+y[2])/2));`

Rules for Macro Parameters

- A macro can have more than one parameter, and each parameter can be used more than once in the replacement text. Click the Examples link.

  Examples
- A macro can have as many parameters as needed, but all of the parameters in the list must be used in the substitution string. Click to see an example.

  Example of Invalid Macro Definition
- When you invoke the macro, you must pass the correct number of arguments to the macro.

The following macro, which calculates the average of five values, has five parameters:

```
#define AVG5(v, w, x, y, z) (((v)+(w)+(x)+(y)+(z))/5)
```

This macro, in which the ternary operator determines the larger of two values, uses each of its parameters twice.

```
#define LARGER(x, y) ((x) > (y) ? (x) : (y))
```

The following macro definition is invalid because the parameter z is not used in the substitution string:

```
#define ADD(x, y, z) ((x) + (y))
```

Syntax
Placement of Opening Parenthesis

Placement of Opening Parenthesis
When you write a macro definition, the opening parenthesis must immediately follow the macro name; there can be no whitespace. The opening parenthesis tells the preprocessor that a function macro is being defined, and not a simple symbolic constant type substitution. Click to see an example.

Example of Wrong Placement

Parameters Enclosed in Parentheses
In the substitution string, each parameter is enclosed in parentheses. This is necessary to avoid unwanted side effects when passing expressions as arguments to the macro.

Look at the following definition:

```
#define SUM (x, y, z) ((x)+(y)+(z))
```

Because of the space between SUM and `(`, the preprocessor treats this like a simple substitution macro. Every occurrence of SUM in the source code is replaced with `(x, y, z) ((x)+(y)+(z))`, clearly not what you wanted.

Look at the following example of a macro defined without parentheses. If you invoke the macro with a simple variable as an argument, there's no problem. What if you pass an expression as an argument? The resulting macro expansion does not give the proper result. If you use parentheses, you can avoid the problem.

	Incorrect: Without Parentheses	**Correct: With Parentheses**
Macro Definition	`#define SQUARE(x) x*x`	`#define SQUARE(x) (x)*(x)`
Macro Invocation	`result = SQUARE(x + y);`	`result = SQUARE(x + y);`
Macro Expansion	`result = x + y * x + y;`	`result = (x + y)*(x + y);`

Stringizing Operator
The *stringizing* operator # (sometimes called the *string literal* operator) gives additional flexibility in macro definitions. When a macro parameter is preceded by # in the substitution string, the argument is converted into a quoted string when the macro is expanded. Click the Example link.

Example

Arguments with Special Characters
The conversion performed by the stringizing operator takes special characters into account. If a character in the argument normally requires an escape character, # precedes it with a backslash. Click the Example link.

Example

If you define a macro as

```
#define OUT(x) printf(#x)
```

and you invoke it with the statement

```
OUT(Hello Mom);
```

it expands to

```
printf("Hello Mom");
```

In the previous example, the macro was defined as

```
#define OUT(x) printf(#x)
```

If the invocation is

```
OUT("Hello Mom");
```

it expands to

```
printf("\"Hello Mom\"");
```

Example Program
You can see a demonstration of the # operator in Listing 21.4.

Listing 21.4: Using the # Operator in Macro Expansion

Code

```
 1: /* LIST2104.c: Day 21 Listing 21.4 */
 2: /* Demonstrates # operator in macro expansion. */
 3:
 4: #include <stdio.h>
 5:
 6: #define OUT(x) printf(#x " is equal to %d.", x)
 7:
 8: int main(void) {
 9:   int value = 123;
10:
11:   OUT(value);
12:   return 0;
13: }
```

Output

```
value is equal to 123.
```

Description

By using the # operator on line 6, the call to the macro expands with the variable name value as a quoted string passed to the printf() function. After expansion, the macro OUT looks like this:

```
printf("value" " is equal to %d.",  value );
```

Concatenation Operator
The *concatenation* operator ## (sometimes called the *token-pasting* or *merging* operator) concatenates, or joins, two strings in the macro expansion. It does not include quotation marks or special treatment of escape characters. Its main use is to create sequences of C source code. Click the Example link.

Example

You can see that by using the ## operator, you determine which function is called. You have actually modified the C source code.

If you define and invoke a macro as

```
#define CHOP(x) func ## x
salad = CHOP(3)(q, w);
```

the macro invoked in the second line is expanded to

```
salad = func3 (q, w);
```

Which Should You Use?
You have seen that function macros can be used in place of real functions, at least in situations where the resulting code is relatively short. Function macros can extend beyond one line but usually become impractical beyond a few lines. When you can use either a function or a macro, which should you use? It's a trade-off between program size and program speed.

Program Size
A macro's definition is expanded into the code each time the macro is encountered in the source code. If your program invokes a macro 100 times, 100 copies of the expanded macro code are in the final program. In contrast, a function's code exists only as a single copy. Therefore, in terms of program size, the advantage goes to a true function. Click the Tip button.

DO use #defines, especially for symbolic constants. Symbolic constants make your code much easier to read. Examples of things to put into defined constants are colors, TRUE/FALSE, YES/NO, the keyboard keys, and maximum values. Symbolic constants are used throughout this course.

DON'T overuse macro functions. Use them where needed, but be sure they are a better choice than a normal function.

Program Speed
When a program calls a function, a certain amount of processing overhead is required to pass execution to the function code and then return execution to the calling program. There is no processing overhead in "calling" a macro because the code is right there in the program. In terms of speed, a function macro has the advantage.

Importance
These size/speed considerations are not usually of much concern to the beginning programmer. Only with large, time-critical applications do they become important.

Precompiling a Program
At times, you may want to see what your expanded macros look like, particularly when they are not working properly. To see the expanded macros, you instruct the compiler to create a listing file that includes macro expansion after the compiler's first pass of the code. You might not be able to do this within an Integrated Programmer Interface (IDE); you might have to work from the command prompt. Most compilers have a flag that should be set during compilation. This flag is passed as a command-line parameter to the compiler. Click the Example link.

Example

The Precompiled File
The compiler makes the first pass on your source code and produces a temporary file. This file is an ASCII text file containing the preprocessed source code. All header files are included, #define macros are expanded, and other preprocessor directives are carried out. You can use your editor to view this file and see what your expanded macros look like.

To precompile a program called PROGRAM.C with the Symantec compiler, you enter

```
ztc1 -e -l program.c
```

On a UNIX system, you enter

```
cc -E program.c
```

Consult your compiler manual to determine what flag you need to add to get the precompiled file.

Topic 7.2.2: The #include Directive

Including Header Files
You have already learned how to use the #include preprocessor directive to include header files in your program. When it encounters an #include directive, the preprocessor reads the specified file and inserts it at the location of the directive.

Nesting #include Directives
You cannot use the * or ? wildcards to read in a group of files with one #include directive. However, you can nest #include directives. That is, an included file can contain #include directives, which can contain #include directives. Most compilers limit the number of levels deep that you can nest, but you can usually nest up to ten levels.

Enclosing the Filename in Angle Brackets
There are two ways to specify the filename for an #include directive. If the filename is enclosed in angle brackets, such as `#include <stdio.h>` (as you have seen throughout this course), the preprocessor first looks for the file in the *standard* directory (also known as the standard include directory). If the file is not found, or there is no standard directory specified, the preprocessor looks for the file in the current directory.

The Standard Directory
"What is the standard directory?" you might be asking. In DOS, it's the directory or directories specified by the DOS INCLUDE environment variable. Your DOS documentation contains complete information on the DOS environment. In brief, however, you set an environment variable with a SET command (usually, but not necessarily, in your AUTOEXEC.BAT file). Most compilers automatically set the INCLUDE variable in the AUTOEXEC.BAT file when the compiler is installed.

Enclosing the Filename in Double Quotation Marks
The second method of specifying the file to be included is enclosing the filename in double quotation marks: `#include "myfile.h"`. In this case, the preprocessor does not search the standard directories, but looks instead in the directory containing the source code file being compiled. Generally speaking, header files that you write should be kept in the same directory as the C source code files, and they are included by using double quotation marks. The standard directory is reserved for header files supplied with your compiler.

Topic 7.2.3: Using #if, #elif, #else, and #endif

Conditional Compilation
There are four preprocessor directives that control conditional compilation. The term *conditional compilation* means that blocks of C source code are compiled only if certain conditions are met.

#if Versus if Statement
In many ways the #if family of preprocessor directives operates like the C language's if statement. The difference is that if controls whether certain statements are executed, whereas #if controls whether they are compiled.

Syntax
The structure of an #if block is as follows:

```
#if constant-expression-1
   statement-list-1
#elif constant-expression-2
   statement-list-2
...
#elif constant-expression-n
   statement-list-n
#else
   default_statement-list
#endif
```

Parameters

The test expression #if uses can be almost any expression that evaluates to a constant. You can't use the sizeof() operator, type casts, or the float type. For the most part, you use #if to test symbolic constants created with the #define directive.

Each statement-list consists of one or more C statements of any type, including preprocessor directives. They do not need to be enclosed in braces, although they can be.

The #if and #endif directives are required, but #elif and #else are optional. You can have as many #elif directives as you want, but only one #else.

Process
When the compiler reaches an #if directive, it tests the associated condition. If it evaluates as TRUE (non-zero), the statements following the #if are compiled. If it evaluates as FALSE (zero), the compiler tests, in order, the conditions associated with each #elif directive. The statements associated with the first TRUE #elif are compiled. If none of the conditions evaluates as TRUE, the statements following the #else directive are compiled.

What Is Passed to the Compiler
Note that, at most, a single block of statements within the #if...#endif construction is compiled. If the compiler finds no #else directive, it might not compile any statements.

Usage
The possible uses for these conditional compilation directives are limited only by your imagination. Click the Example link.

Example

Here's one example. Suppose you're writing a program that uses a great deal of country-specific information. This information is contained in a header file for each country. When you compile the program for use in different countries, you can use an #if...#endif construction as follows:

```
#if ENGLAND == 1
   #include "england.h"
#elif FRANCE == 1
   #include "france.h"
#elif ITALY == 1
   #include "italy.h"
#else
   #include "usa.h"
#endif
```

Then, by using #define to define the appropriate symbolic constant, you can control which header file is included during compilation.

Topic 7.2.4: Using #if...#endif to Help Debug

Including Conditional Debugging Code
Another common use for #if...#endif is to include conditional debugging code in the program. Click the Example link.

Example

The defined() Operator
The defined() operator is useful when you write conditional compilation directives. This operator tests to see whether a particular name is defined. Thus, the expression

```
defined( NAME )
```

evaluates as TRUE or FALSE depending on whether or not NAME is defined. By using defined() you can control compilation, based on previous definitions, without regard to the specific value of a name. Click the Example link.

Example

You could define a DEBUG symbolic constant set to either 1 or 0. Throughout the program you can insert debugging code as follows:

```
#if DEBUG == 1

   debugging code here

#endif
```

During program development, if you define DEBUG as 1, the debugging code is included to help track down any bugs. Once the program is working properly, you can redefine DEBUG as 0 and recompile the program without the debugging code.

Referring to the previous debugging code example, you could rewrite the #if...#endif section as follows:

```
#if defined( DEBUG )

   debugging code here

#endif
```

! defined()
You can also use defined() to assign a definition to a name only if it has not been previously defined. Use the NOT operator, !, as follows:

```
#if !defined( TRUE ) /*if TRUE is not defined.*/
   #define TRUE 1
#endif
```

The Specified Name
Notice the defined() operator does not require that a name be defined as anything in particular. For example, after the program line

```
#define RED
```

the name RED is defined, but not as *being* anything. Even so, the expression define (RED) still evaluates as TRUE. Of course, occurrences of RED in the source code are removed and not replaced with anything, so you must use caution.

Topic 7.2.5: Avoiding Multiple Inclusions of Header Files

Preventing Multiple Includes
As programs grow, or as you use header files more often, you run a risk of accidentally including a header file more than once. This can cause the compiler to balk in confusion. Using the directives that you've learned, you can easily avoid this problem.

5.

Listing 21.5: Using Preprocessor Directives with Header Files	
Code 1:	/* LIST2105.c: Day 21 Listing 21.5 */

```
 2: /* PROG.H - A header file with a check */
 3: /* to prevent multiple includes! */
 4:
 5: #if defined(PROG_H)
 6: /* The file has been included already. */
 7: #else
 8: #define PROG_H
 9:
10: /* Header file information goes here... */
11:
12: #endif
```

Description	Examine what this header file does. On line 5, it checks whether PROG_H is defined. Notice that PROG_H is similar to the name of the header file. If PROG_H is defined, a comment is included on line 6, and the program looks for the #endif at the end of the header file. This means that nothing more is done. How does PROG_H get defined? It is defined on line 8. The first time this header is included, the preprocessor checks whether PROG_H is defined. It won't be, so control goes to the #else statement. The first thing done after the #else is to define PROG_H so that any other inclusions of this file skip the body of the file. Lines 9 – 11 could contain any number of commands or declarations.

Topic 7.2.6: The #undef Directive

Removing the Definition
The #undef directive is the opposite of #define—it removes the definition from a name. Click the Example link.

Example

#undef Used with #define or #if
You can use #undef and #define to create a name that is defined only in parts of your source code. You can use this in combination with the #if directive, as previously explained, for more control over conditional compilations.

Here's an example:

```
#define DEBUG 1

/* In this section of the program, occurrences of
 * DEBUG are replaced with 1 and the expression
 * defined(DEBUG) evaluates to TRUE.
 */

#undef DEBUG

/* In this section of the program, occurrences of
 * DEBUG are not replaced and the expression
 * defined(DEBUG) evaluates to FALSE.
 */
```

Topic 7.3: Predefined Macros

Predefined Macros
Most compilers have a number of predefined macros. The most useful of these are _ _DATE_ _, _ _TIME_ _, _ _LINE_ _, and _ _FILE_ _. Notice that each of these is preceded and followed by double underscores. This is done

to prevent you from redefining them. Click the Tip button.

DO use the _ _LINE_ _ and _ _FILE_ _ macros to make your error messages more helpful.

DON'T forget the #endif when using the #if statement.

DO put parentheses around the value to be passed to a macro. This prevents errors. For example:

```
#define CUBE(x)   (x)*(x)*(x)
instead of:
#define CUBE(x)   x*x*x
```

Process
These macros work just like the macros described earlier in this unit. When the precompiler comes across one of these macros, it replaces the macro with the macro's code.

_ _DATE_ _ and _ _TIME_ _
_ _DATE_ _ and _ _TIME_ _ are replaced with the current date and time. This is the date on and time at which the source file is precompiled, which is useful information for controlling versions.

_ _LINE_ _ and _ _FILE_ _
The other two macros are even more valuable. _ _LINE_ _ is replaced by the current source-file line number. _ _FILE_ _ is replaced with the current source-filename. These two macros are best used when trying to debug a program or deal with errors. Click the Example link.

Example

Consider the following printf() statement:

```
31:
32: printf( "Program %s: (%d) Error opening file ",
           _ _FILE _ _, _ _LINE _ _);
33:
```

If these lines were part of a program called MYPROG.C, this would print:

```
Program MYPROG.C: (32) Error opening file
```

This might not seem important at this point, but as your programs grow and spread across multiple source files, finding errors becomes more difficult. Using _ _LINE _ _ and _ _FILE _ _ makes debugging a great deal easier.

Topic 7.4: Day 21 Q&A

Questions & Answers
Here are some questions to help you review what you have learned in this unit.

Question 1
When compiling multiple files, how does the compiler know which filename to use for the executable file?

Answer
You might think the compiler uses the name of the file containing the main() function; however, this is not usually the case. When compiling from the command line, the first file listed is used to determine the name.

Question 2
Do header files need to have a .H extension?

Do header files need to have a .H extension?

Answer
No. You can give a header file any name you want. It is standard practice to use the .H extension.

Question 3
When including header files, can I use an explicit path?

Answer
Yes. If you want to state the path where a file to be included is, you can. In such a case, you put the name of the include file between quotation marks.

Question 4
Are all the predefined macros and preprocessor directives in this unit?

Answer
No. The predefined macros and directives presented in this unit are ones common to most compilers. However, most compilers also have additional macros and constants.

Topic 7.5: Day 21 Think About Its

Think About Its
Now, answer the following questions and test your knowledge of the concepts presented in this unit.

Topic 7.6: Day 21 Try Its

Try Its
Now, take some time to perform the following exercises. They will provide you with experience in using what you've learned.

*** Exercise 1**
Use your compiler to compile multiple source files into a single executable file. (You can use Listings 21.1, 21.2 and 21.3 or your own listings.)

Listing 21.1: SQUARE.C, the Main Module

```
Code  1:  /* LIST2101.c: Day 21 Listing 21.1 */
      2:  /* square.c, the main module. */
      3:  /* Inputs a number and displays its square. */
      4:
      5:  #include <stdio.h>
      6:  #include "calc.h"
      7:
      8:  int main(void) {
      9:    int x;
     10:
     11:    printf("Enter an integer value: ");
     12:    scanf("%d", &x);
     13:
     14:    printf("\nThe square of %d is %ld.", x, sqr(x));
     15:    return 0;
     16:  }
```

Listing 21.2: CALC.C, the Secondary Module

```
Code  1:  /* LIST2102.c: Day 21 Listing 21.2 */
      2:  /* calc.c, the secondary module. */
      3:  /* Module containing calculation functions. */
```

```
      4:
      5:    #include "calc.h"
      6:
      7:    long sqr(int x) {
      8:      return ((long)x * x);
      9:    }
```

Listing 21.3: CALC.H, the Header File for CALC.C

```
Code  1:    /* LIST2103.c: Day 21 Listing 21.3 */
      2:    /* calc.h, the header file for calc.c. */
      3:
      4:    long sqr(int x);
```

*** Exercise 2**

Write an error routine that receives an error number, line number, and module name. The routine should print a formatted error message and then exit the program. Use the predefined macros for the line number and module name (Pass the line number and module name from the location where the error occurs). A possible example for a formatted error could be

```
module.c (Line ##): Error number ##
```

Answer

```
/* EXER2102  Day 21 Exercise 2 */
/* Display binary representation of a number */

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

void err_rtn(int errno, int lineno, char *pgmname);

int main(void)  {

  err_rtn(16, __LINE__, __FILE__);

  return 0;
}

void err_rtn(int errno, int lineno, char *pgmname)  {

  printf("%s (Line %d): Error number %d\n", pgmname,          lineno, errno);
  exit(1);
}
```

*** Exercise 3**

Modify the previous exercise to make the error more descriptive. Create a text file with your editor that contains an

error number and message. Call this file ERRORS.TXT. It could contain information such as the following:

```
1     Error number 1
2     Error number 2
90    Error opening file
100   Error reading file
```

Have your error routine search this file and display the appropriate error message based on a number passed to it.

Answer

Answer

```
/* EXER2103  Day 21 Exercise 3 */
/* Display error information including error message */

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

void err_rtn(int errno, int lineno, char *pgmname);

int main(void)  {
  err_rtn(90, __LINE__, __FILE__);
  return 0;
}

/* Error routine to search errors.txt for error message. */

void err_rtn(int errno, int lineno, char *pgmname)  {
#define BUFSIZE 80
  FILE *fp;
  char filename[] = "errors.txt";
  char buff[BUFSIZE], numstr[4];
  int x, num;

  /* Open error message file */

  if ((fp = fopen(filename,"r")) != NULL)  {
    while (1)  {
      fgets(buff,BUFSIZE,fp);
      if (feof(fp))
        break;
      for (x = 0;!isspace(buff[x]); x++)
        numstr[x] = buff[x];

      /* Look for matching error number in record. */

      num = atoi(numstr);
      if (num == errno)  {
        for (x = x + 1;isspace(buff[x]); x++)
          ;
  /* Print error information with message. */

        printf("%s (Line %d): %s\n", pgmname, lineno,
               &buff[x]);
        exit(1);
      }
    }
  }

  /* Print error information without message. */

  printf("%s (Line %d): Error number %d\n", pgmname,
         lineno, errno);
  exit(1);
}
```

*** Exercise 4**

Some header files might be included more than once when you are writing a modular program. Use preprocessor directives to write the skeleton of a header file that compiles only the first time it is encountered during compilation.

Answer

```
/* EXER2104.C Day 21 Exercise 4 */
/* A header file with a check to prevent multiple includes. */

#if defined(PROG_H)
/* The file has been included already. */
#else
#define PROG_H

/* Header file information goes here... */


#endif
```

*** Exercise 5**
This is the last exercise of the course, and its content is up to you. Select a programming task of interest to you that also meets a real need you have. For example, you could write programs to catalog your compact disk collection, keep track of your checkbook, or calculate financial figures related to a planned house purchase. There's no substitute for tackling a real-world programming problem in order to sharpen your programming skills and help you remember all the things you learned in this course.

Topic 7.7: Day 21 Summary

Programming with Multiple Source-Code Files
This unit covered some of the more advanced programming tools available with C compilers. You've learned to write a program that has source code divided among multiple files or modules. This practice, called modular programming, makes it easy to reuse general-purpose functions in more than one program.

Using the C Preprocessor
You saw how you can use preprocessor directives to create function macros, for conditional compilation, and other tasks. Finally, you saw that the compiler provides some function macros for you.

Unit 8. Week 3 in Review

8. Week 3 in Review

Week 3 in Review
You have finished your third and final week of learning how to program in C. You started the week covering such advanced topics as pointers and disk files. In the middle of the week, you saw just a few of the many functions contained in most C compilers' libraries of functions. You ended your week by discovering the odds and ends needed to get the most from your compiler and the C language. Listing R3.1 should pull together many of these topics.

More Information
The numbers to the left of the line numbers indicate the day (e.g. D02 = Day 2) that covers the concept presented on that line. If you are confused by the line, refer to the referenced day for more information.

Listing R3.1: Week Three Review Listing

```
Code  1: /* Program: week3.c                                */
      2: /* A program to enter names and phone numbers.   */
      3: /* This information is written to a disk file    */
      4: /* specified with a command-line parameter.      */
      5:
      6: /* Included files. */
      7:
      8: #include <conio.h>
```

```
      8: #include <conio.h>
      9: #include <stdlib.h>
     10: #include <stdio.h>
D19  11: #include <time.h>
D18  12: #include <string.h>
     13:
     14: /* Defined constants. */
     15:
     16: #define YES 1
     17: #define NO  0
     18: #define REC_LENGTH 54
     19:
     20: /* Variables. */
     21:
     22: struct record {
     23:   char fname[15+1];       /* First name + NULL   */
     24:   char lname[20+1];       /* Last name + NULL    */
     25:   char mname[10+1];       /* Middle name + NULL  */
     26:   char phone[9+1];        /* Phone number + NULL */
     27: } rec;
     28:
     29: /* Function prototypes. */
     30:
     31: void display_usage(char *filename);
     32: int display_menu(void);
     33: void get_data(FILE *fp, char *progname,
     34:                         char *filename);
     35: void display_report(FILE *fp);
     36: void clear_kb(void);
     37: int continue_function(void);
     38: int look_up(FILE *fp);
     39:
     40: /* Start of program. */
     41:
D20  42: int main(int argc, char *argv[])  {
D20  43:   FILE *fp;
     44:   int cont = YES;
     45:
D20  46:   if (argc < 2)  {
     47:     display_usage(argv[0]);
     48:     exit(1);
     49:   }
     50:
D16  51:   if ((fp = fopen(argv[1], "a+")) == NULL)  {
     52:     /* Open file. */
     53:
     54:     fprintf(stderr, "%s(%d) -- Error opening file"
     55:             " %s", argv[0], __LINE__, argv[1]);
     56:     exit(1);
     57:   }
     58:
     59:   while (cont == YES)  {
     60:     switch(display_menu())  {
     61:       case '1':
D18  62:         get_data(fp, argv[0], argv[1]);
     63:         clear_kb();
     64:         break;
     65:       case '2':
     66:         display_report(fp);
     67:         clear_kb();
     68:         break;
     69:       case '3':
     70:         look_up(fp);
     71:         break;
     72:       case '4':
     73:         printf("\n\nThank you for using this "
     74:                "program!");
     75:         cont = NO;
     76:         break;
     77:       default:
     78:         printf("\n\nInvalid choice, Please select"
     79:                " 1 to 4!");
```

```
     80:          break;
     81:        }
     82:      }
     83:      fclose(fp);             /* Close file. */
D16  84:      return 0;
     85:  }
     86:
     87:  /* Function: display_menu() */
     88:
     89:  int display_menu(void)  {
     90:    printf("\n");
     91:    printf("\n     MENU");
     92:    printf("\n   ========\n");
     93:    printf("\n1. Enter names");
     94:    printf("\n2. Display report");
     95:    printf("\n3. Look up number");
     96:    printf("\n4. Quit");
     97:    printf("\n\nEnter Selection ==> ");
     98:
     99:    return(getchar());
    100:  }
    101:
    102:  /* Function: get_data()    */
    103:
    104:  void get_data(FILE *fp, char *progname,
    105:                          char *filename)  {
    106:    int cont = YES;
    107:
    108:    while (cont == YES) {
    109:      clear_kb();
    110:      printf("\n\nPlease enter information:");
    111:
    112:      printf("\n\nEnter first name: ");
    113:      gets(rec.fname);
    114:
    115:      printf("\n\nEnter middle name: ");
    116:      gets(rec.mname);
    117:
    118:      printf("\nEnter last name: ");
    119:      gets(rec.lname);
    120:
    121:      printf("\nEnter phone in 123-4567 format: ");
    122:      gets(rec.phone);
    123:
D16 124:      if (fseek(fp, 0, SEEK_END) == 0)
    125:      if (fwrite(&rec, 1, sizeof(rec), fp) !=
    126:          sizeof(rec))  {
    127:        fprintf(stderr, "%s(%d) -- Error writing to"
D21 128:               " file %s", progname, __LINE__,
    129:                filename);
    130:        exit(2);
    131:      }
    132:      cont = continue_function();
    133:    }
    134:  }
    135:
    136:  /* Function: display_report()                    */
    137:  /* Purpose: To print out the formatted names of */
    138:  /* people in the file.                           */
    139:

    140:  void display_report(FILE *fp)  {
    141:    time_t rtime;
    142:    int num_of_recs = 0;
    143:
    144:    time(&rtime);
    145:
    146:    fprintf(stdout, "\n\nRun Time: %s",
D19 147:           ctime(&rtime));
    148:    fprintf(stdout, "\nPhone number report\n");
    149:
D16 150:    if (fseek(fp, 0, SEEK SET) == 0)  {
```

```
    151:     fread(&rec, 1, sizeof(rec), fp);
    152:     while (!feof(fp))  {
    153:       fprintf(stdout, "\n\t%s, %s %c %s",
    154:               rec.lname, rec.fname, rec.mname[0],
    155:               rec.phone);
    156:       num_of_recs++;
    157:       fread(&rec, 1, sizeof(rec), fp);
    158:       }
    159:       fprintf(stdout, "\n\nTotal number of "
    160:              "records: %d", num_of_recs);
    161:       fprintf(stdout, "\n\n*** End of Report ***");
    162:   }
    163:   else
    164:     fprintf(stderr, "\n\n*** ERROR WITH REPORT "
    165:            "***\n");
    166: }
    167:
    168: /* Function: continue_function() */
    169:
    170: int continue_function(void)  {
    171:   char ch;
    172:
    173:   do  {
    174:     printf("\n\nDo you wish to continue? "
    175:            "(Y)es/(N)o: ");
    176:     ch = getchar();
    177:   } while (strchr("NnYy", ch) == NULL);
    178:
D17 179:   if (ch == 'n' || ch == 'N')
    180:     return(NO);
    181:   else
    182:     return(YES);
    183: }
    184: /* Function: display_usage */
    185:
    186: void display_usage(char *filename)
    187: {
    188:   printf("\n\nUSAGE: %s filename", filename);
    189:   printf("\n\n       where filename is a file to "
    190:          "store people\'s names");
    191:   printf("\n         and phone numbers.\n\n");
    192: }
    193:
    194: /* Function: look_up */
    195:
    196: int look_up(FILE *fp)  {
    197:   char tmp_lname[20+1];
    198:   int ctr = 0;
    199:
    200:   clear_kb();
    201:   fprintf(stdout, "\n\nPlease enter last name to"
    202:          " be found: ");
    203:   gets(tmp_lname);
    204:
    205:   if (strlen(tmp_lname) != 0)  {
D17 206:     if (fseek(fp, 0, SEEK_SET) == 0)  {
D16 207:       fread(&rec, 1, sizeof(rec), fp);
    208:       while (!feof(fp))  {
    209:         if (strcmp(rec.lname, tmp_lname) == 0)  {
    210:         /* If matched. */

D17 211:
    212:         fprintf(stdout, "\n%s %s %s - %s",
    213:                rec.fname, rec.mname, rec.lname,
    214:                rec.phone);
    215:         ctr++;
    216:       }
    217:       fread(&rec, 1, sizeof(rec), fp);
    218:       }
    219:     }
    220:     fprintf(stdout, "\n\n%d names matched.", ctr);
    221:   }
```

```
222:   }
223:   else  {
224:     fprintf(stdout, "\nNo name entered.");
225:   }
226:   return(ctr);
227: }
228:
229: /* Function: clear_kb()                          */
230: /* Purpose: This function clears the keyboard   */
231: /* of extra characters. Returns: Nothing        */
232:
233: void clear_kb(void)  {
234:   char junk[80];
235:   gets(junk);
236: }
```

Description

You might consider this an extremely long program; however, it barely does what it needs to do. This program is similar to the programs presented in the reviews of week one and week two. A few of the data items tracked in the week two review program have been dropped. This program enables the user to enter information for people. The information to be tracked is a first name, last name, middle name and phone number. The major difference you should notice in this program is that there is no limit to the number of people that can be entered into the program. This is because a disk file is used.

This program enables the user to specify the name of a file to be used with the program. main() starts on line 42 with the argc and argv arguments required to get the command-line parameters. You saw this on Day 20, "Odds and Ends." Line 46 checks the value of argc to see how many parameters were entered on the command line. If argc is less than 2, only one parameter was entered (the command to run the program). Because a filename was not provided, display_usage() is called with argv[0] as an argument. argv[0], the first parameter entered on the command line, is the name of the program.

The display_usage() function is on lines 186 – 192. Whenever you write a program that takes command-line arguments, it is a good idea to include a function similar to display_usage() that shows how to use the program. Why doesn't the function just write the name of the program instead of using the variable filename? The answer is simple. By using a variable, you don't have to worry if the user renames the program; the usage description is always accurate.

Be aware that a majority of the new concepts in this program come from Day 16, "Using Disk Files." Line 43 declares a file pointer called fp that is used throughout the program to access the file provided on the command line. Line 51 tries to open this file with a mode of "a+" (argv[1] is the second item listed on the command line). The "a+" mode is used because you want to be able to add to the file and read any records that already exist. If the open fails, lines 54 – 55 display an error message before line 56 exits the program. Notice that the error message contains descriptive information. Also notice that __LINE__, covered on Day 21, "Taking Advantage of Preprocessor Directives and More," indicates the line number where the error occurred. Once the file is opened, a menu is presented. When the user selects the option to exit the program, line 83 closes the file with fclose() before the program returns control to the operating system.

In the get_data() function, there are a few significant changes. Line 104 contains the function header. The function now accepts three pointers. The first pointer is the most important; it is

the handle for the file to be written to. Lines 108 – 133 contain a while loop that continues to get data until the user wants to quit. Lines 110 – 122 prompt for the data in the same format that the review program from week two did. Line 124 calls fseek() to set the pointer in the disk file to the end to write the new information. Notice that this program does not do anything if the seek fails. A good program would handle such a failure. Line 125 writes the data to the disk file with a call to fwrite().

The report presented in this program also has changed. One change typical of most "real world" reports is the addition of the current date and time to the top of the report. On line

141, the variable rtime is declared. This variable is passed to time() and then displayed using the ctime() function. These time functions were presented on Day 19, "Exploring the Function Library."

Before the program can start printing the records in the file, it needs to reposition the file pointer back to the beginning of the file. This is done on line 150 with another call to fseek(). Once the file pointer is positioned, records can be read. Line 151 does the first read. If the read is successful, the program begins a while loop that continues until the end of the file is reached (when feof() returns a nonzero value). If the end of the file hasn't been reached, lines 153 – 155 print the information, line 156 counts the record, and line 157 tries to read the next record. You should notice that functions are used without checking their return values. To protect the program against errors, the function calls should contain checks to ensure that no errors occurred.

The continue_function() contains one small modification. Line 177 has been changed to use the strchr() function. This function makes the line of code easier to understand.

The look_up() function is new. Lines 196 – 226 contain the function which searches the disk file for all records with a given last name. Lines 201 – 202 prompt for the name to be found and store it in a local variable called tmp_lname. If tmp_lname is not blank (line 205), the file pointer is set to the beginning of the file. Each record is then read. Using strcmp() (line 209), the record's last name is compared to tmp_lname. If the names match, the record is printed (lines 212 – 214). This continues until the end of the file is reached. Again, you should notice that not all the functions had their return values checked. You always should check return values.

You should be able to make modifications to this program to create your own files that can store any information. Using the functions you learned in the third week along with the other functions that your library has should enable you to create nearly any program you want.

Unit 9. Reference

Topic 9.1: C Reserved Words

C Reserved Words
The following identifiers are reserved C keywords. They should not be used for any other purpose in a C program. They are allowed, of course, within double quotation marks.

Keyword	Description
`asm`	A C keyword that denotes inline assembly language code.
`auto`	The default storage class.
`break`	A C command that exits for, while, switch, and do...while statements unconditionally.
`case`	A C command used within the switch statement.
`char`	The simplest C data type.
`const`	A C data qualifier that prevents a variable from being changed. Some compilers

const	A C data modifier that prevents a variable from being changed. See volatile.
continue	A C command that resets a for, while, or do...while statement to the next iteration.
default	A C command used within the switch statement to catch any instances not specified with a case statement.
do	A C looping command used in conjunction with the while statement. The loop will always execute at least once.
double	A C data type that can hold double-precision floating-point values.
else	A statement signaling alternative statements to be executed when an if statement evaluates to FALSE.
enum	A C data type that allows variables to be declared that accept only certain values.
extern	A C data modifier indicating that a variable will be declared in another area of the program.
float	A C data type used for floating-point numbers.
for	A C looping command that contains initialization, incrementation, and conditional sections.
goto	A C command that causes a jump to a predefined label.
if	A C command used to change program flow based on a TRUE/FALSE decision.
int	A C data type used to hold integer values.
long	A C data type used to hold larger integer values than int.
register	A storage modifier that specifies that a variable should be stored in a register, if possible.
return	A C command that causes program flow to exit from the current function and return to the calling function. It also can be used to return a single value.
short	A C data type that is used to hold integers. It is not commonly used, and is the same size as an int on most computers.
signed	A C modifier that is used to signify that a variable can have both positive and negative values.
sizeof	A C operator that returns the size (number of bytes) of the item.
static	A C modifier that is used to signify that the compiler should retain the variable's value.
struct	A C keyword used to combine C variables of any data types into a group.
switch	A C command used to change program flow into a multitude of directions. Used in conjunction with the case statement.
typedef	A C modifier used to create new names for existing variable and function types.
union	A C keyword used to allow multiple variables to share the same memory space.
unsigned	A C modifier that is used to signify that a variable will only contain positive values. See signed.
void	A C keyword used to signify either that a function does not return anything or that a pointer being used is considered generic or able to point to any data type.
volatile	A C modifier that signifies that a variable can be changed. See const.
while	A C looping statement that executes a section of code as long as a condition remains TRUE.

Topic 9.2: Binary and Hexadecimal Notation

Binary and Hexadecimal Notation

Binary and hexadecimal numeric notations are frequently used in the world of computers, so it's a good idea to be familiar with them. All number notation systems use a certain base. For the binary system, the base is 2, and for the hexadecimal system, it is 16.

Base 10

Base 10

To understand what *base* means, consider the familiar decimal notation system, which uses a base of 10. Base 10 requires 10 different digits, 0–9. In a decimal number, each successive digit (starting right and moving to the left) indicates a successively increasing power of 10. The rightmost digit specifies 10 to the 0 power, the second digit specifies 10 to the 1 power, the third digit specifies 10 to the 2 power, and so on. Because any number to the 0 power equals 1 and any number to the 1 power equals itself, you have

```
first digit:    10^0 = ones
second digit:   10^1= tens
third digit:    10^2 = hundreds
...
nth digit:      10^(n-1)
```

Click the Example link.

Example

You can break down the decimal number 382 as follows:

```
382 (decimal)
___---------- 2 x 10^0 = 2 x 1      =   2
__---------- 8 x 10^1 = 8 x 10     =  80
_---------- 3 x 10^2 = 3 x 100    = 300
                                    ------
                                sum = 382
```

Base 16

The hexadecimal system works in the same way except that it uses powers of 16. Because the base is 16, the hexadecimal system requires 16 digits. It uses the regular digits 0–9, and then represents the decimal values 10–15 by the letters A–F. Click to see some hex/decimal equivalents:

Hex/Decimal Equivalents

Because some hex numbers (those without letters) look like decimal numbers, they are written with the prefix 0X in order to distinguish them. Click the Example link.

Example

Hexadecimal Values	**Decimal Values**
9	9
A	10
F	15
10	16
1F	31

The following is a three-digit hex number broken down into its decimal components:

```
2DA (hex)
  |--------------  10 x 16^0 = 10 x 1   =  10 (decimal)
 |---------------- 13 x 16^1 = 13 x 16  = 208
|----------------  2 x 16^2 =  2 x 256 = 512
                                       ------
                                          730
```

Base 2
Binary is a base 2 system, and as such requires only two digits, 0 and 1. Each place in a binary number represents a power of 2. Click the Example link.

Example

As you can see, binary notation requires many more digits than either decimal or hexadecimal to represent a given value. It is useful, however, because the way a binary number represents values is very similar to the way a computer stores integer values in memory.

Breaking down a binary number gives you the following:

```
10010111 (binary)
       |---------- 1 x 2^0 = 1 x 1      =   1
      |----------- 1 x 2^1 = 1 x 2      =   2
     |------------ 1 x 2^2 = 1 x 4      =   4
    |-------------   x 2^3 = 0 x 8      =   0
   |--------------- 1 x 2^4 = 1 x 16     =  16
  |----------------   x 2^5 = 0 x 32     =   0
 |-----------------   x 2^6 = 0 x 64     =   0
|------------------ 1 x 2^7 = 1 x 128    = 128
                                        ------
                                           151
```