Click to Print

Welcome to C in 21 Days: Week 2

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 8 Numeric Arrays

Arrays are a type of data storage that you often use in C programs. You had a brief introduction to arrays on Day 6, "Basic Program Control." Here we'll look at arrays in more detail.

Today, you learn

- What an array is.
- The definition of single and multidimensional numeric arrays.
- How to declare and initialize arrays.

Topic 1.1: What Is an Array?

Arrays and Array Elements
An *array* is a collection of data storage locations, each having the same data type and the same name. Each storage location in an array is called an *array element*.

Usage
Why do you need arrays in your programs? This question can be answered with an example. Click the Example link and then the Figure link.

Example

Figure 8.1 illustrates the difference between using individual variables and an array.

If you are keeping track of your business expenses for 1994, and filing your receipts by month, you could have a separate folder for each month's receipts, but it would be more convenient to have a single folder with twelve compartments.

Extend this example to computer programming. Imagine that you are designing a program to keep track of your business expense totals. The program could declare twelve separate variables, one for each month's expense total. This approach is analogous to having twelve separate folders for your receipts. Good programming practice, however, would utilize an array with twelve elements, storing each month's total in the corresponding array element. This approach is comparable to filing your receipts in a single folder with twelve compartments.

Topic 1.1.1: Single-Dimensional Arrays

Single-Dimensional Arrays
A *single-dimensional* array is an array that has only a single subscript. A *subscript* is a number in brackets following an array's name. This number can identify the number of individual elements in the array. All of C's data types can be used for arrays. C array elements are always numbered starting at 0. Click the Example link.

Example

For the business expenses program, you could use the program line

```
float expenses[12];
```

to declare an array of type float. The array is named expenses, and contains twelve elements. Each of the twelve elements is the exact equivalent of a single float variable. The twelve elements of expenses are numbered 0–11.

Placement of Array Declarations
When you declare an array, the compiler sets aside a block of memory large enough to hold the entire array. Individual array elements are stored in sequential memory locations, as illustrated in Figure 8.2.

The location of array declarations in your source code is important. As with nonarray variables, the declaration's location affects how your program can use the array. The effect of a declaration's location is covered in more detail on Day 12, "Variable Scope." For now, place your array declarations with other variable declarations, just before the start of main(). Click the Tip button on the toolbar.

DON'T forget that array subscripts start at element 0.

DO use arrays instead of creating several variables that store the same thing. (For example, if you want to store total sales for each month of the year, create an array with twelve elements to hold sales, rather than creating a sales variable for each month.)

Use of Array Elements
An array element can be used in your program anywhere a nonarray variable of the same type can be used. Individual elements of the array are accessed by using the array name followed by the element subscript enclosed in square brackets. Click the Example link.

Example

The statement

```
expenses[1] = 89.95;
```

stores the value 89.95 in the second array element. (Remember, the first array element is expenses[0], not expenses[1].) Likewise, the statement

```
expenses[10] = expenses[11];
```

assigns the value that is stored in array element expenses[11] to array element expenses[10].

Allowable Subscripts

When you use arrays, keep the element numbering scheme in mind: In an array of *n* elements, the allowable subscripts range from 0 to *n*–1. If you use the subscript value *n*, you may get program errors. The C compiler does not recognize whether your program uses an array subscript that is out of bounds. Your program compiles and links, but out-of-range subscripts generally produce erroneous results. Sometimes you might want to treat an array of *n* elements as if its elements were numbered 1–*n*. Click the Example link.

Example

Types of Array Subscripts

When you refer to an array element, the array subscript can be a literal constant, as in `expenses[1]`. Your programs may, however, frequently use a subscript that is a C integer variable or expression, or even another array element. Click the Example link for a detailed example.

Example

Here are some examples of using variables or expressions in the subscript of an array reference:

```
/* Declare an array with 100 float elements. */
float expenses[100];
/* Declare another array with 10 int elements. */
int a[10];
int i=5

/* Additional programming staments here.*/

/* Assign the value 5 to 3rd element in a[]. */
a[2]=5

/* Assign 100 to 6th element in expenses[]. */
expenses[i] = 100;

/* Assign 50 to 6th element in the array. */
```

```
expenses[2 + 3] = 50;

/* Assign 100 again to 6th element in array. */
expenses[a[2]] = 100;
```

That last example may need an explanation. Say, for instance, you have an integer array named a[ ] and that the value 8 is stored in element a[2]. Then, writing

```
expenses[a[2]]
```

has the same effect as writing

```
expenses[8];
```

For instance, in the previous example of a program to keep track of business expense totals, January's expense total would be stored in expenses[0], February's in expenses[1], and so on. A more natural method might be to store January's expense total in expenses[1], February's in expenses[2], and so on. The simplest way to do this is to declare the array with one more element than needed, and ignore element 0. In this case, you would declare the array

```
float expenses[13];
```

You also could store some related data in element 0 (the yearly expense total, perhaps).

Example Program

The program EXPENSES.C in Listing 8.1 demonstrates the use of an array. This is a simple program with no real practical use; it's for demonstration purposes only.

Listing 8.1: EXPENSES.C

```
Code  1: /* LIST0801.c: Day 8 Listing 8.1 */
      2: /* EXPENSES.C — Demonstrates use of an array */
      3:
      4: #include <stdio.h>
      5:
      6: /* Declare an array to hold expenses, and a */
      7: /* counter variable. */
      8:
      9: float expenses[13];
     10: int count;
     11:
```

```
12: int main(void) {
13:   /* Input data from keyboard into array. */
14:
15:   for (count = 1; count < 13; count++) {
16:     printf("Enter expenses for month %d: ", count);
17:     scanf("%f", &expenses[count]);
18:   }
19:
20:   /* Print array contents. */
21:
22:   for (count = 1; count < 13; count++) {
23:     printf("\nMonth %d = $%.2f", count,
24:             expenses[count]);
25:   }
26:   return 0;
27: }
```

Output

```
Enter expenses for month 1: 100
Enter expenses for month 2: 200.12
Enter expenses for month 3: 150.50
Enter expenses for month 4: 300
Enter expenses for month 5: 100.50
Enter expenses for month 6: 34.25
Enter expenses for month 7: 45.75
Enter expenses for month 8: 195.00
Enter expenses for month 9: 123.45
Enter expenses for month 10: 111.11
Enter expenses for month 11: 222.20
Enter expenses for month 12: 120.00

Month 1 = $100.00
Month 2 = $200.12
Month 3 = $150.50
Month 4 = $300.00
Month 5 = $100.50
Month 6 = $34.25
Month 7 = $45.75
Month 8 = $195.00
Month 9 = $123.45
Month 10 = $111.11
Month 11 = $222.20
Month 12 = $120.00
```

Description

When you run EXPENSES.C, the program prompts you to enter expenses for months 1 – 12. The values you enter are stored in an array. You must enter some value for each month. After the twelfth value is entered, the array contents are displayed on the screen.

The flow of the program is similar to listings you have seen before. It starts with a comment that describes what the program is

	going to do. Notice that the name of the program is included, EXPENSES.C. By including the name of the program in a comment, you know which program you are viewing. This is helpful when you print the listings, and then want to make a change. Lines 6 – 7 contain an additional comment explaining the variables that are being declared. In line 9, an array of 13 elements is declared. In this program, only twelve elements are needed, one for each month, but 13 have been declared. The for loop in lines 15 – 18 ignores element 0. This allows the program to use elements 1 – 12, which relate directly to the twelve months. Going back to line 10, a variable, count, is declared and is used throughout the program as a counter and an array index. The program's main() function begins on line 12. As stated earlier, the program uses a for loop to print a message and accept a value for each of 12 months. Notice that in line 17, the scanf() function uses an array element. In line 9, the expenses array was declared as float, so %f is used. The *address of* operator (&) also is placed before the array element, just as if it were a regular type float variable and not an array element. Lines 22 – 25 contain a second for loop that prints the values just entered. An additional formatting command has been added to the printf() function so that the expenses values print in a more orderly fashion. For now, know that %.2f prints a floating number with 2 digits to the right of the decimal. Additional formatting commands are covered in more detail on Day 14, "Working with the Screen, Printer, and Keyboard."

Topic 1.1.2: Multidimensional Arrays

Multidimensional Array

A multidimensional array has more than one subscript. A *two-dimensional array* has two subscripts, a *three-dimensional array* has three subscripts, and so on. There is no limit to the number of dimensions a C array can have. (There is a limit on total array size, discussed later in the unit.)

Two-Dimensional Array Example

The structure of this two-dimensional array is illustrated in Figure 8.3.

Three- or Four-Dimensional Arrays
A three-dimensional array could be thought of as a cube. Four-dimensional arrays (and higher) are probably best left to your imagination. All arrays, no matter how many dimensions they have, are stored sequentially in memory. More detail on array storage is presented on Day 15, "More on Pointers."

You might write a program that plays checkers. The checkerboard contains 64 squares arranged in eight rows and eight columns. Your program could represent the board as a two-dimensional array, as follows:

```
int checker[8][8];
```

The resulting array has 64 elements: checker[0][0], checker[0][1], checker[0][2] ... checker[7][6], checker[7][7].

Topic 1.2: Naming and Declaring Arrays

Naming Arrays
The rules for assigning names to arrays are the same as for variable names, covered on Day 3, "Numeric Variables and Constants." An array name must be unique. It can't be used for another array or for any other identifier (variable, constant, and so on). As you have probably realized, array declarations follow the same form as declarations of nonarray variables, except that the number of elements in the array must be enclosed in square brackets immediately after the array name.

Declaring Arrays
When you declare an array, you can specify the number of elements with a literal constant (as was done in the earlier examples) or with a symbolic constant created with the #define directive.

Example

With most compilers, however, you cannot declare an array's elements with a symbolic constant created with the const keyword. Click the Example link and then the Tip button.

Example

DO use #define statements to create constants that can be used when declaring arrays. Then you can change easily the number of elements in the array. In GRADES.C, you could change the number of students in the #define and you wouldn't have to make any other changes in the program.

DO avoid multidimensional arrays with more than three dimensions. Remember multidimensional arrays can get very big, very quickly.

Thus,

```
#define MONTHS 12
int array[MONTHS];
```

is equivalent to

```
 int array[12];
```

```
const int MONTHS = 12;
int array[MONTHS];          /* Wrong! */
```

Example Program
Listing 8.2, GRADES.C, is another program demonstrating the use of a single-dimensional array. GRADES.C uses an array to store 10 grades.

Listing 8.2: GRADES.C

```
Code  1: /* LIST0802.c: Day 8 Listing 8.2 */
      2: /* GRADES.C -- Example program with array */
      3: /* Get 10 grades and then average them. */
      4:
      5: #include <stdio.h>
      6:
      7: #define MAX_GRADE 100
      8: #define STUDENTS 10
      9:
     10: int grades[STUDENTS];
```

```
11:
12: int idx;
13: int total = 0;    /* Used for average. */
14:
15: int main(void) {
16:   for (idx = 0; idx < STUDENTS; idx++) {
17:     printf("Enter Person %d's grade: ", idx +1);
18:     scanf("%d", &grades[idx]);
19:
20:     while (grades[idx] > MAX_GRADE) {
21:       printf("\nThe highest grade possible is %d",
22:               MAX_GRADE);
23:       printf("\nEnter correct grade: ");
24:       scanf("%d", &grades[idx]);
25:     }
26:
27:     total += grades[idx];
28:   }
29:
30:   printf("\n\nThe average score is %d",
31:           (total / STUDENTS));
32:
33:   return 0;
34: }
```

Output

```
Enter Person 1's grade: 95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade: 95
Enter Person 10's grade: 85
The average score is 73
```

Description

Like EXPENSES.C, this listing prompts the user for input. It prompts for 10 people's grades. Instead of printing each grade, it prints the average score.

As you have learned earlier, arrays are named like regular variables. On line 10, the array for this program is named grades. It should be safe to assume that this array holds grades. On lines 7 and 8, two constants are defined, MAX_GRADE and STUDENTS. These constants can be changed easily. Knowing that STUDENTS is defined as 10, you then know that the grades array has 10 elements. In the listing, there are two other variables

	declared, idx and total. An abbreviation for index, idx is used as a counter and array subscript. A running total of all grades is kept in total. The heart of this program is the for loop on lines 16 – 28. The for statement initializes idx to 0, the first subscript for an array. It then loops as long as idx is less than the number of students. Each time it loops, it increments idx by 1. For each loop, the program prompts for a person's grade (lines 17 and 18). Notice that in line 17, 1 is added to idx in order to count the people from 1 to 10 instead of from 0 to 9. Because arrays start with subscript 0, the first grade is put in grade[0]. Instead of confusing users by asking for Person 0's grade, they are asked for Person 1's grade. Lines 20 – 25 contain a while loop nested within the for loop. This is an edit check that ensures the grade is not higher than the maximum grade, MAX_GRADE. Users are prompted to enter a correct grade if they enter a grade that is too high. You should check program data whenever you can. Line 27 adds the entered grade to a total counter. On lines 31 – 32, this total is used to print the average score (total/STUDENTS).

Topic 1.2.1: Initializing Arrays

Initializing Single-Dimensional Arrays
You can initialize all or part of an array when you first declare it. Follow the array declaration with an equal sign and a list of values enclosed in braces and separated by commas. The listed values are assigned in order to array elements starting at number 0.

Example of Array Size Specified

If you omit the array size, the compiler creates an array just large enough to hold the initialization values.

Example of Array Size Omitted

You can, however, include too few initialization values.

Example of Too Few Initialization Values Specified

```
int array[4] = { 100, 200, 300, 400 };
```

assigns the value 100 to array[0], 200 to array[1], 300 to array[2], and 400 to array[3].

Thus, the statement

```
int array[] = { 100, 200, 300, 400 };
```

would have exactly the same effect as the previous array declaration statement, i.e. it assigns the value 100 to array[0], 200 to array[1], 300 to array[2], and 400 to array[3].

```
int array[10] = { 1, 2, 3 };
```

If you do not explicitly initialize an array element, you cannot be sure what value it holds when the program runs. If you include too many initializers (more initializers than array elements), the compiler detects an error.

Initializing Multidimensional Arrays
Multidimensional arrays also can be initialized. The list of initialization values is assigned to array elements in order, with the last array subscript changing first. Click the Example link.

Example

One Format
For example,

int array[4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

results in the following assignments:

```
array[0][0] is equal to 1
array[0][1] is equal to 2
array[0][2] is equal to 3
array[1][0] is equal to 4
array[1][1] is equal to 5
array[1][2] is equal to 6
...
array[3][1] is equal to 11
array[3][2] is equal to 12
```

A Clearer Format

When you initialize multidimensional arrays, you can make your source code clearer by using extra braces to group the initialization values and also by spreading them over several lines. The following initialization is equivalent to the one given previously:

```
int array[4][3] = { { 1, 2, 3 } , { 4, 5, 6 } ,
                    { 7, 8, 9 } , { 10, 11, 12 } };
```

Remember, initialization values must be separated by a comma—even when there is a brace between them. Also, be sure to use braces in pairs—a closing brace for every opening brace—or the compiler becomes confused.

Example Program

Now look at an example that demonstrates the advantages of arrays. The program in Listing 8.3, RANDOM.C, creates a 1000-element, three-dimensional array and fills it with random numbers. The program then displays the array elements on the screen. Imagine how many lines of source code you would need to perform the same task with nonarray variables.

You see a new library function, getchar(), in this program. The getchar() function reads a single character from the keyboard. In Listing 8.3, getchar() pauses the program until the user presses ENTER. getchar() is covered in detail on Day 14, "Working with the Screen, Printer, and Keyboard."

Listing 8.3: RANDOM.C

```
Code  1: /* LIST0803.c: Day 8 Listing 8.3 */
      2: /* RANDOM.C -- Demonstrates using a */
      3: /* multidimensional array */
      4:
      5: #include <stdio.h>
```

```
#include <stdlib.h>

/* Declare a three-dimensional array with 1000 */
/* elements. */

int random[10][10][10];
int a, b, c;
long total = 0;

int main(void) {
  /* Fill the array with random numbers. The C */
  /* library function rand() returns a random */
  /* number. Use one for loop for each array */
  /* subscript. */

  for (a=0; a < 10; a++) {
    for (b=0; b < 10; b++) {
        for (c=0; c < 10; c++) {
            random[a][b][c] = rand();
        }
    }
  }

  /* Display the array elements 10 at a time. */

  for (a=0; a < 10; a++) {
    for (b=0; b < 10; b++) {
        for (c=0; c < 10; c++) {
            printf("\nrandom[%d][%d][%d] =",
                    a, b, c);
            printf("%d", random[a][b][c]);
        }
        printf("\nPress ENTER to continue,"
               " CTRL-C to quit.");
        getchar();
    }
  }
  return 0;
}
```

Output

```
random[0][0][0] = 346
random[0][0][1] = 130
random[0][0][2] = 10982
random[0][0][3] = 1090
random[0][0][4] = 11656
random[0][0][5] = 7117
random[0][0][6] = 17595
random[0][0][7] = 6415
random[0][0][8] = 22948
random[0][0][9] = 31126
Press ENTER to continue, CTRL-C to quit.
random[0][1][0] = 9004
```

```
random[0][1][1] = 14558
random[0][1][2] = 3571
random[0][1][3] = 22879
random[0][1][4] = 18492
random[0][1][5] = 1360
random[0][1][6] = 5412
random[0][1][7] = 26721
random[0][1][8] = 22463
random[0][1][9] = 25047
Press ENTER to continue, CTRL-C to quit
      ...         ...
random[9][8][0] = 6287
random[9][8][1] = 26957
random[9][8][2] = 1530
random[9][8][3] = 14171
random[9][8][4] = 6951
random[9][8][5] = 213
random[9][8][6] = 14003
random[9][8][7] = 29736
random[9][8][8] = 15028
random[9][8][9] = 18968
Press ENTER to continue, CTRL-C to quit.
random[9][9][0] = 28559
random[9][9][1] = 5268
random[9][9][2] = 20182
random[9][9][3] = 3633
random[9][9][4] = 24779
random[9][9][5] = 3024
random[9][9][6] = 10853
random[9][9][7] = 28205
random[9][9][8] = 8930
random[9][9][9] = 2873
Press ENTER to continue, CTRL-C to quit.
```

Description

On Day 6, "Basic Program Control," you saw a program that used a nested for statement; this program has two for loops nested. Before you look at the for statements in detail, note that lines 11 and 12 declare 4 variables. The first is an array named random, used to hold random numbers. random is a three-dimensional type int array that is 10 by 10 by 10 giving a total of 1,000 type int elements (10x10x10). Imagine coming up with 1,000 unique variable names if you couldn't use arrays. Line 12 then declares three variables, a, b, and c, used to control the for loops.

This program also includes a new header file, STDLIB.H (for standard library), in line 6. This is included to provide the prototype for the rand() function used on line 23.

	The bulk of the program is contained in two nests of for statements. The first is on lines 20 – 26, the second on lines 30 – 41. Both for nests have the same structure. They work just like the loops in Listing 6.2, but they go one level deeper. In the first set of for statements, line 23 is executed repeatedly. Line 23 assigns the return value of a function, rand(), to an element of the random array. rand() is a library function that returns a random number. Going backwards through the listing, you can see that line 22 changes variable c from 0 to 9. This loops through the farthest right subscript of the random array. Line 21 loops through b, the middle subscript of the random array. Each time b changes, it loops through all the c elements. Line 20 increments variable a, which loops through the farthest left subscript. Each time this subscript changes, it loops through all 10 values of subscript b, which in turn loop through all 10 values of c. This loop initializes every value in the random array to a random number. Lines 30 – 41 contain the second nest of for statements. These work like the previous for statements, but this loop prints each of the values assigned previously. After 10 are displayed, lines 37 – 38 print a message and waits for a key to be pressed. Line 39 takes care of the keypress. getchar() returns the value of a key that has been pressed. If ENTER has not been pressed, getchar() waits. Run this program and watch the displayed values.

Topic 1.2.2: Maximum Array Size

64K of Data Variables

Because of the way memory models work, you should not try to create more than 64K of data variables for now. An explanation of this limitation is beyond the scope of this course, but there's no need to worry; none of the programs in this course exceed this limitation. To understand more, or to get around this limitation, consult your compiler manuals. Generally, 64K is enough data space for programs, particularly the relatively simple programs you write as you work though this course. A single array can take up the entire 64K of data storage if your program uses no other variables. Otherwise, you need to apportion the available data space as needed.

To Calculate the Storage Space Required

The size of an array in bytes depends on the number of elements it has, as well as each element's size. Element size depends on the data type of the array. The sizes for each numeric data type on a typical 32-bit machine are shown here for your convenience. To calculate the storage space required for an array, you multiply the number of elements in the array by the element size. For example, a 500-element array of type float requires `(500) * (4) = 2000` bytes of storage space.

Table 8.1: Storage Space Requirements for Numeric Data Types for Most PCs

Data Type	Size (Bytes)
int	4
short	2
long	4
float	4
double	8

sizeof() Operator

Storage space can be determined within a program or on a specific computer by using C's sizeof() operator; sizeof() is a unary operator and not a function. It takes as its argument a variable name or the name of a data type and returns the size, in bytes, of its argument.

The use of sizeof() is illustrated in Listing 8.4.

Listing 8.4: Using the sizeof() Operator

```
Code  1: /* LIST0804.c: Day 8 Listing 8.4 */
      2: /* Demonstrates the sizeof() operator.*/
      3:
      4: #include <stdio.h>
      5:
      6: /* Declare several 100 element arrays. */
      7:
      8: int intarray[100];
      9: float floatarray[100];
     10: double doublearray[100];
     11:
     12: int main(void) {
     13:   /* Display the sizes of numeric data types. */
     14:
```

```
15:   printf("\n\nSize of int = %d bytes",
16:          sizeof(int));
17:   printf("\nSize of short = %d bytes",
18:          sizeof(short));
19:   printf("\nSize of long = %d bytes",
20:          sizeof(long));
21:   printf("\nSize of float = %d bytes",
22:          sizeof(float));
23:   printf("\nSize of double = %d bytes",
24:          sizeof(double));
25:
26:   /* Display the sizes of the three arrays. */
27:
28:   printf("\nSize of intarray = %d bytes",
29:          sizeof(intarray));
30:   printf("\nSize of floatarray = %d bytes",
31:          sizeof(floatarray));
32:   printf("\nSize of doublearray = %d bytes",
33:          sizeof(doublearray));
34:
35:   return 0;
36: }
```

Output

```
Size of int = 4 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 400 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

Description

Enter and compile the program in this listing by using the procedures you learned on Day 1, "Getting Started." When the program runs, it displays the sizes—in bytes—of the three arrays and five numeric data types.

On Day 3, "Numeric Variables and Constants," you ran a similar program; however, this listing uses sizeof() to determine the storage size of arrays. Lines 8, 9, and 10 declare three arrays, each of different types. Lines 28 – 33 print the size of each array. The size should equal the size of the array's variable type times the number of elements. For example, if an int is 4 bytes, intarray should be 4 times 100 or 400 bytes. Run the program and check the values.

Topic 1.3: Day 8 Q&A

Questions & Answers
Take a look at some questions that are frequently asked by people new to C programming.

Question 1
What happens if I use a subscript on an array that is larger than number of elements in the array?

Answer
If you use a subscript that is out of bounds from the array declaration, the program will probably compile and even run. However, the results from such a mistake can be unpredictable. This can be a difficult error to find once it starts causing problems, so make sure you're careful when initializing and accessing array elements.

Question 2
What happens if I use an array without initializing it?

Answer
This mistake doesn't produce a compiler error. If you don't initialize an array, there can be any value in the array elements. You may get unpredictable results. You should always initialize variables and arrays so that you know exactly what is in them. On Day 12, "Variable Scope," you are introduced to one exception to the need to initialize. For now, play it safe.

Question 3
How many dimensions can an array have?

Answer
As stated in the unit, you can have as many dimensions as you want. As you add more dimensions, you use more data storage space. You should declare an array only as large as you need to avoid wasting storage space.

Question 4
Is there an easy way I can initialize an entire array at once?

Answer
Each element of an array must be initialized. The safest way for a beginning C programmer to initialize an array is either with a declaration as shown in this unit or with a for statement. There are other ways to initialize an array, but they are beyond the scope of this unit and the course at this point.

Question 5
Can I add two arrays together (or multiply, divide, or subtract them)?

Answer
If you declare two arrays, you cannot add the two together. Each element must be added individually.

Question 6
Why is it better to use an array instead of individual variables?

Answer
With arrays, you can group like values with a single name. In Listing 8.3, 1,000 values were stored. Creating 1,000 variable names and initializing each to a random number would have taken a tremendous amount of typing. By using an array, you made the task easy.

Topic 1.4: Day 8 Think About Its

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

```
int array[2][3][5][8];
```

Sample Array
```
int eightyeight[88];
```

Method 1
```
int eightyeight[88] = {88,88,88,88,88,88,88,...,88};
```

with 88 `88`s placed between the braces instead of using "`...`".

Method 2
```
eightyeight[88];
int x;

for (x = 0; x < 88; x++)
  eightyeight[x] = 88;
```

```
int main()
{
   int szBuffer[12][10];
   int iSub1, iSub2;

       for( iSub1 = 0; iSub1 < 12; iSub1++ )
             for ( iSub2 = 0; iSub2 < 10; iSub2++ )
                   szBuffer[ iSub1 ][ iSub2 ] = 0;
       return 0;
}
```

```
int x, y;
int array[10][3];
int main(void) {
    for ( x = 0; x < 3; x++ )
        for ( y = 0; y < 10; y++ )
            array[x][y] = 0;
}
```

```
int array[10];
int x = 1;

int main(void) {
    ??????????????????????????
        array[x] = 99;
}
```

Topic 1.5: Day 8 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 C program line that would declare three one-dimensional integer arrays,

named one, two, and three, with 1,000 elements each.

Answer

```
int one[1000], two[1000], three[1000];
```

*** Exercise 2**
Write the statement that would declare a 10-element integer array and initialize all its elements to 1.

Answer

```
int array[10] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };
```

*** Exercise 3**
Given the array

```
int eightyeight[88];
```

write the code to initialize all the array elements to 88.

Answer

This exercise can be solved in numerous ways. The first way is to initialize the array when it is declared:

```
int eightyeight[88] = {88,88,88,88,88,88,88,...,88};
```

This would require that 88 `88`s be placed between the braces instead of using "..." as shown. This method is not recommended for initializing such a big array. The following is a better method.

```
eightyeight[88];
int x;

for (x = 0; x < 88; x++)
  eightyeight[x] = 88;
```

*** Exercise 4**
Given the array

```
int stuff[12][10];
```

write the code to initialize all the array elements to 0.

Answer

```
stuff[12][10];
int sub1, sub2;

for (sub1 = 0; sub1 < 12; sub1++)
  for (sub2 = 0; sub2 < 10; sub2++)
    stuff[sub1][sub2] = 0;
```

*** Exercise 5**
BUG BUSTERS: What is wrong with the following code fragment?

```
int x, y;
int array[10][3];
int main(void) {
    for ( x = 0; x < 3; x++ )
        for ( y = 0; y < 10; y++ )
            array[x][y] = 0;
}
```

Answer

Be careful with this fragment. The bug presented here is easy to create. Notice that the array is 10-by-3, but is initialized as a 3-by-10 array.

To describe this differently, the left subscript is declared as 10, but the outer for loop uses x as the left subscript and increments it with only 3 values (not enough). Similarly, the right subscript is declared as 3m, but the second for loop uses y as the right subscript and increments it with 10 values (too many). This can cause unpredictable results.

You can fix this program in one of two ways. The first is to switch x and y in the line that does the initialization:

```
int x, y;
int array[10][3];
int main(void) {
    for ( x = 0; x < 3; x++ )
        for ( y = 0; y < 10; y++ )
            array[y][x] = 0; /* switched variables */
}
```

The second way (which is recommended) is to switch the values in the for loops.

```
int x, y;
int array[10][3];
int main(void) {
    for ( x = 0; x < 10; x++ ) /* switched values */
        for ( y = 0; y < 3; y++ )
            array[x][y] = 0;
}
```

*** Exercise 6**

BUG BUSTERS: What is wrong with the following?

```
int array[10];
int x = 1;

int main(void) {
    for (x = 1; x <= 10; x++)
        array[x] = 99;
}
```

Answer

This, we hope, was an easy bug to bust. This program initializes an element in the array that is out of bounds. If you have an array with 10 elements, their subscripts are 0 to 9. This program initializes elements with subscripts 1 through 10. You cannot initialize array[10] because it does not exist. The for statement should be changed to one of the following:

```
for (x = 1; x <= 9; x++) /* initializes 9 of the 10 elements */
```

or

```
for (x = 0; x <= 9; x++)
```

Notice that `x <=9` is the same as `x < 10`. Either is appropriate; `x < 10` is more common.

*** Exercise 7**

Write a program that puts random numbers into a two-dimensional array that is 5 by 4. Print the values in columns on the screen. (Hint: use the rand() function from Listing 8.3.)

Answer

Listing 8.3: RANDOM.C

```
Code  1: /* LIST0803.c: Day 8 Listing 8.3 */
      2: /* RANDOM.C -- Demonstrates using a */
      3: /* multidimensional array */
      4:
      5: #include <stdio.h>
      6: #include <stdlib.h>
      7:
      8: /* Declare a three-dimensional array with 1000 8?
      9: /* elements. */
     10:
     11: int random[10][10][10];
     12: int a, b, c;
     13:
     14: int main(void) {
     15:   /* Fill the array with random numbers. The C */
     16:   /* library function rand() returns a random */
     17:   /* number. Use one for loop for each array */
     18:   /* subscript. */
     19:
     20:   for (a=0; a < 10; a++) {
     21:     for (b=0; b < 10; b++) {
     22:         for (c=0; c < 10; c++) {
     23:             random[a][b][c] = rand();
     24:         }
     25:     }
     26:   }
     27:
     28:   /* Display the array elements 10 at a time. */
     29:
     30:   for (a=0; a < 10; a++) {
     31:     for (b=0; b < 10; b++) {
     32:         for (c=0; c < 10; c++) {
```

```
     33:              printf("\nrandom[%d][%d][%d] =",
     34:                      a, b, c);
     35:              printf("%d", random[a][b][c]);
     36:          }
     37:          printf("\nPress ENTER to continue,"
     38:                 " CTRL-C to quit.");
     39:          getchar();
     40:      }
     41:    }
     42:    return 0;
     43: }
```

The following is one of many possible answers:

```
/* EXER0807.c: Day 8 Exercise 8.7 */
/* Using 2 dimensional arrays and rand() */

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

/* Declare the array. */

int array[5][4];
int a, b;

int main(void) {
  for (a = 0; a < 5; a++) {
    for (b = 0; b < 4; b++) {
      array[a][b] = rand();
    }
  }

  /* Now print the array elements. */

  for (a = 0; a < 5; a++) {
    for (b = 0; b < 4; b++) {
      printf("%d\t", array[a][b]);
    }
    printf("\n");    /* Go to a new line. */
  }

  return 0;
}
```

* Exercise 8

Rewrite Listing 8.3 to use a single-dimensional array. Print the average of the 1,000 variables before printing the individual values. Note: Don't forget to pause after every 10 values are printed.

Answer

Listing 8.3: RANDOM.C

```
Code  1: /* LIST0803.c: Day 8 Listing 8.3 */
      2: /* RANDOM.C -- Demonstrates using a */
      3: /* multidimensional array */
      4:
      5: #include <stdio.h>
      6: #include <stdlib.h>
      7:
      8: /* Declare a three-dimensional array with 1000 8?
      9: /* elements. */
     10:
     11: int random[10][10][10];
     12: int a, b, c;
     13:
     14: int main(void) {
     15:   /* Fill the array with random numbers. The C */
     16:   /* library function rand() returns a random */
     17:   /* number. Use one for loop for each array */
     18:   /* subscript. */
     19:
     20:   for (a=0; a < 10; a++) {
     21:     for (b=0; b < 10; b++) {
     22:         for (c=0; c < 10; c++) {
     23:             random[a][b][c] = rand();
     24:         }
     25:     }
     26:   }
     27:
     28:   /* Display the array elements 10 at a time. */
     29:
     30:   for (a=0; a < 10; a++) {
     31:     for (b=0; b < 10; b++) {
     32:         for (c=0; c < 10; c++) {
     33:             printf("\nrandom[%d][%d][%d] =",
     34:                     a, b, c);
     35:             printf("%d", random[a][b][c]);
     36:         }
     37:         printf("\nPress ENTER to continue,"
     38:                " CTRL-C to quit.");
     39:         getchar();
     40:     }
     41:   }
```

```
    42:   return 0;
    43: }
```

The following is one of many possible answers:

```
/* EXER0808.c: Day 8 Exercise 8.8 */
/* RANDOM.C using a single dimensional array */

#include <stdio.h>
#include <stdlib.h>
/* Declare a single-dimensional 1000-element array. */

int random[1000];
int a, b, c;
long total = 0;

int main(void) {
  /* Fill the array with random numbers.
   * The C library function rand() returns
   * a random number. Use one for loop for
   * each array subscript. */

  for (a=0; a < 1000; a++) {
    random[a] = rand();
    total += random[a];
  }

  /* Now display the average */

  printf("\n\nAverage is: %ld",total/1000);

  /* Now display the array elements 10 at a time. */

  for (a=0; a < 1000; a++) {
        printf("\nrandom[%d] =", a);
    printf("%d", random[a]);

    if (a % 10 == 0 && a > 0) {
        printf("\nPress ENTER to continue, CTRL-C to quit.");
        getchar();
    }
  }
  return 0;
}
```

*** Exercise**

Write a program that initializes an array of 10 elements. Each element should be equal to its subscript. The program should then print each of the 10 elements.

Answer

Following are two solutions. The first initializes the array at the time it is declared, the second initializes it in a for loop:

Answer 1:

```
/* EXER0809.c: Day 8 Exercise 8.9a */

#include <stdio.h>

/* Declare a single-dimensional array. */

int elements[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int idx;

int main(void) {
  for (idx=0; idx < 10; idx++) {
    printf("\nelements[%d] = %d", idx, elements[idx]);
  }
  return 0;
}
```

Answer 2:

```
/* EXER0809.c: Day 8 Exercise 8.9b */
/* Answer 2 */

#include <stdio.h>

/* Declare a single-dimensional array */

int elements[10];
int idx;

int main(void)
{
  for (idx=0; idx < 10; idx++)
    elements[idx] = idx;
```

```
  for (idx=0; idx < 10; idx++)
    printf("\nelements[%d] = %d", idx, elements[idx]);
  return 0;
}
```

*** Exercise 1**
Modify the program from exercise nine. After printing the initialized values, the program should copy the values to a new array and add 10 to each value. Then the new array values should be printed.

Answer

The following is one of many possibilities:

```
/* EXER0810.c: Day 8 Exercise 8.10 */

#include <stdio.h>

/* Declare a single-dimensional array. */

int elements[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int new_array[10];
int idx;

int main(void) {
  for (idx=0; idx < 10; idx++) {
    new_array[idx] = elements[idx] + 10;
  }

  for (idx=0; idx < 10; idx++) {
    printf("\nelements[%d] = %d \tnew_array[%d] = %d",
        idx, elements[idx], idx, new_array[idx]);
  }
  return 0;
}
```

Topic 1.6: Day 8 Summary

What Is an Array?
This unit introduced numeric arrays, a powerful data storage method that enables you

to group a number of same-type data items under the same group name. Individual items, or elements, in an array are identified by using a subscript after the array name. Computer programming tasks that involve repetitive data processing lend themselves to array storage.

Declaring and Initializing Arrays
Like nonarray variables, arrays must be declared before they can be used. Optionally, array elements can be initialized when the array is declared.

Unit 2. Day 9 Pointers

This unit introduces you to pointers, an important part of the C language. Pointers provide a powerful and flexible method for manipulating data in your programs.

Today, you learn

- The definition of a pointer.

- The uses of pointers.
- How to declare and initialize pointers.
- How to use pointers with simple variables and arrays.
- How to use pointers to pass arrays to functions.

Topic 2.1: What Is a Pointer?

What Is a Pointer?
To understand pointers, you need a basic knowledge of how your computer stores information in memory. The following is a somewhat simplified account of PC memory storage.

Topic 2.1.1: Your Computer's Memory

Addresses in Memory
A PC's RAM consists of many thousands of sequential storage locations, and each location is identified by a unique address. The memory addresses in a given computer range from 0 to a maximum value that depends on the amount of memory installed.

Use of System's Memory
When you are using your computer, the operating system uses some of the system's memory. When you're running a program, the program's code and data (the machine language instructions for the program's various tasks and the information the program is using, respectively) also use some of the system's memory. This section examines the memory storage for program data.

When Declaring Variables
When you declare a variable in a C program, the compiler sets aside a memory location with a unique address to store that variable. The compiler associates that address with the variable's name. When your program uses the variable name, it automatically accesses the proper memory location. The location's address is used, but it is hidden from you, and you need not be concerned with it.

Schematic Representation
Figure 9.1 shows this schematically. A variable named rate has been declared and initialized to 100. The compiler has set aside storage at address 1004 for the variable and has associated the name rate with the address 1004.

Topic 2.1.2: Creating a Pointer

A Variable Containing the Address of Another Variable

You should note that in Figure 9.1 the address of the variable rate (or any other variable) is a number and can be treated like any other number in C. If you know a variable's address, you can create a second variable in which to store the address of the first. The steps are as follows:

Steps in Creating Pointers		
Step	**Action**	**Example**
1	The first step is to declare a variable to hold the address of rate. Give it the name p_rate, for example. At first, p_rate is uninitialized. Storage has been allocated for p_rate, but its value is undetermined.	Figure 9.2.
2	The next step is to store the address of the variable rate in the variable p_rate. Because p_rate now contains the address of rate, it indicates its storage location in memory. In C parlance, p_rate points to rate or is a pointer to rate.	Figure 9.3.

To summarize, a pointer is a variable that contains the address of another variable. Now you can get down to the details of using pointers in your C programs.

Topic 2.2: Pointers and Simple Variables

Pointers to Simple Variables

In the example just given, a pointer variable pointed to a simple (that is, nonarray) variable. This section shows you how to create and use pointers to simple variables.

Topic 2.2.1: Declaring Pointers

Declaring Pointers

A pointer is a numeric variable and, like all variables, must be declared before it can be used. Pointer variable names follow the same rules as other variables and must be unique. Some programmers use the convention that a pointer to the variable name is called p_*name*. In this course we also sometimes identify pointer names by using

"ptr."

Syntax
A pointer declaration takes the following form:

```
typename *ptrname;
```

typename is any one of C's variable types and indicates the type of the variable that the pointer points to. The asterisk (*) is the *indirection operator* (also known as the dereferencing operator), and it indicates that ptrname is a pointer to type typename and not a variable of type typename. Pointers can be declared along with nonpointer variables. Click the Examples link.

Examples

Here are some more examples:

```
/* ch1 and ch2 both are pointers to
 * variables of type char.*/
char *ch1, *ch2;

/* value is a pointer to type float */
/* percent is an ordinary float variable. */
float *value, percent;
```

Indirection or Multiplication?
The * symbol is used as both the indirection operator (also known as the dereferencing operator) and the multiplication operator. Don't worry about the compiler becoming confused. The context in which * is used always provides enough information so that the compiler can figure out whether you mean indirection or multiplication.

Topic 2.2.2: Initializing Pointers

The address of Operator
Now that you've declared a pointer, what can you do with it? Nothing until you make it point to something. Like regular variables, uninitialized pointers can be used, but the results are unpredictable and potentially disastrous. Until a pointer holds the address of a variable, it isn't useful. Your program must store the address in the

pointer by using the address of operator, the ampersand (&). When placed before the name of a variable, the address of operator returns the address of the variable.

Syntax
Therefore, you initialize a pointer with a statement of the form

```
ptrname = &variable;
```

Example
Look back at the example in Figure 9.3.The program statement to initialize the variable p_rate to point at the variable rate would be

```
p_rate = &rate;
/* assign the address of rate to p_rate */
```

Before the initialization, p_rate didn't point to anything in particular. After the initialization, p_rate is a pointer to rate.

Topic 2.2.3: Using Pointers

Indirection Operator
Now that you know how to declare and initialize pointers, you are probably wondering how to use them. The indirection operator (*) (also known as the dereferencing operator) comes into play again. When the * precedes the name of a pointer, it refers to the variable pointed to. Click the Example link and then the Tip button.

Example

DO understand what pointers are and how they work. The mastering of C requires mastering pointers.

DON'T use an uninitialized pointer. Results can be disastrous if you do.

Continue now with the previous example, where the pointer p_rate has been initialized to point to the variable rate. If you write `*p_rate`, it refers to the variable rate. If you want to print the value of rate (which is 100 in the example), you could write

```
printf("%d", rate);
```

or you could write

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

In C, the two statements are equivalent.

Direct Versus Indirect Access
Accessing the contents of a variable by using the variable name is called *direct access*. Accessing the contents of a variable by using a pointer to the variable is called *indirect access* or *indirection*.

Schematic Representation
Figure 9.4 illustrates that a pointer name preceded by the indirection operator refers to the value of the pointed-to variable.

Review
Pause a minute and think about this material. Pointers are an integral part of the C language, and it is essential that you understand them. Pointers have confused many people, so don't worry if you're feeling a bit puzzled. If you need to review, that's fine. Maybe the following summary can help.

Summary

The program in Listing 9.1 demonstrates basic pointer use. You should enter, compile, and run this program. Note that in the output of this program, the address reported for var may not be 96 on your system.

Listing 9.1: Illustration of Basic Pointer Use

```
Code  1: /* LIST0901.c: Day 9 Listing 9.1 */
      2: /* Demonstrates basic pointer use. */
      3:
      4: #include <stdio.h>
      5:
      6: /* Declare and initialize an int variable. */
      7:
      8: int var = 1;
      9:
     10: /* Declare a pointer to int. */
     11:
     12: int *ptr;
```

```
13:
14: int main(void) {
15:   /* Initialize ptr to point to var. */
16:
17:   ptr = &var;
18:
19:   /* Access var directly and indirectly. */
20:
21:   printf("\nDirect access, var = %d", var);
22:   printf("\nIndirect access, var = %d", *ptr);
23:
24:   /* Display the address of var two ways. */
25:
26:   printf("\n\nThe address of var = %d", &var);
27:   printf("\nThe address of var = %d", ptr);
28:   return 0;
29: }
```

Output

```
Direct access, var = 1
Indirect access, var = 1

The address of var = 96
The address of var = 96
```

Description

In this listing, two variables are declared. In line 8, var is declared as an int and initialized to 1. In line 12, a pointer to a variable of type int is declared and named ptr.

In line 17, the pointer ptr is assigned the address of var using the address of operator (&).

The rest of the program prints the values from these two variables to the screen. Line 21 prints the value of var while line 22 prints the value stored in the location pointed to by ptr. In this program, this value is 1.

Line 26 prints the address of var using the address of operator. This is the same value printed by line 27 using the pointer variable, ptr.

This listing is good to study. It shows the relationship between a variable, its address, a pointer, and the dereferencing of a pointer.

If you have a pointer named ptr that has been initialized to point to the variable var,

then

*ptr and var both refer to the contents of var (that is, whatever value the program has stored there).

ptr and &var refer to the address of var.

As you can see, a pointer name without the indirection operator accesses the pointer value itself, which is, of course, the address of the variable pointed to.

Topic 2.3: Pointers and Variable Types

Multibyte Variables
The previous discussion ignores the fact that different variable types occupy different amounts of memory. On most PCs, a short takes 2 bytes, a float takes 4 bytes, and so on. Each individual byte of memory has its own address, so a multibyte variable actually occupies several addresses.

Addresses of Multibyte Variables
How, then, do pointers handle the addresses of multibyte variables? This is how it works: The address of a variable is actually the address of the lowest byte it occupies. This can be illustrated with an example.

Declare and Initialize the Variables

Pointers to the Variables
The short variable occupies 2 bytes, the char variable occupies 1 byte, and the float variable occupies 4 bytes.

Declare and Initialize the Pointers to the Variables

This example declares and initializes three variables.

```
short vshort = 12252;
char vchar = 90;
float vfloat = 1200.156004;
```

These variables are stored in memory as shown in Figure 9.5.

Now declare and initialize pointers to these three variables.

```
short *p_vshort;
char *p_vchar;
float *p_vfloat;
/* additional code goes here */
p_vshort = &vshort;
p_vchar = &vchar;
p_vfloat = &vfloat;
```

Each pointer is equal to the address of the first byte of the pointed-to variable. Thus, p_vshort equals 1000, p_vchar equals 1003, and p_vfloat equals 1006. Remember, however, that each pointer was declared to point to a certain type of variable. The compiler "knows" that a pointer to type short points to the first of two bytes, a pointer to type float points to the first of four bytes, and so on.

This is diagrammed in Figure 9.6.

Figures 9.5 and 9.6 show some empty memory storage locations among the three variables. This is for the sake of visual clarity. In actual practice, the C compiler stores the three variables in adjacent memory locations with no unused bytes between them.

Topic 2.4: Pointers and Arrays

Pointers and Arrays
Pointers can be useful when you are working with simple variables, but they are more helpful with arrays. There is a special relationship between pointers and arrays in C. In fact, when you use the array subscript notation that you learned on Day 8, "Numeric Arrays," you really are using pointers without knowing it. The following paragraphs explain how this works.

Topic 2.4.1: The Array Name as a Pointer

A Pointer Constant
An array name without brackets is a pointer to the array's first element. Thus, if you have declared an array data[ ], data is the address of the first array element.

"Wait a minute," you might be thinking, "don't you need the address of operator to get an address?" Yes, you also can use the expression `&data[0]` to obtain the address of the array's first element. In C, the relationship `(data == &data[0])` is true.

You've seen that the name of an array is a pointer to the array. Remember that this is a *pointer constant;* it can't be changed and remains fixed for the duration of program execution. This makes sense; if you changed its value, it would point elsewhere and not to the array (which remains at a fixed location in memory).

A Pointer Variable
You can, however, declare a pointer variable and initialize it to point at the array. Click the Example link.

Example

The code

```
int array[100], *p_array;
/* additional code goes here */
p_array = array;
```

initializes the pointer variable p_array with the address of the first element of array[ ]. Because p_array is a pointer variable, it can be modified to point elsewhere. Unlike array, p_array is not locked to pointing at the first element of array[ ]. It could, for example, be pointed at other elements of array[ ]. How would you do this? First, you need to look at how array elements are stored in memory.

Topic 2.4.2: Array Element Storage

Storing Array Elements
As you might remember from Day 8, "Numeric Arrays," the elements of an array are stored in sequential memory locations with the first element in the lowest address. Subsequent array elements (those with an index greater than 0) are stored in higher addresses. How much higher depends on the array's data type (char, int, float, and so forth). Click the Example link.

Example

Take an array of type float. As you learned on Day 3, "Numeric Variables and Constants," a single float variable can occupy four bytes of memory. Each array element is therefore located four bytes above the preceding element, and the address of each array element is four higher than the address of the preceding element. A type double, on the other hand, can occupy eight bytes. In an array of type double, each array element is located eight bytes above the preceding element, and the address of each array element is eight higher than the address of the preceding element.

Relationship Between Array Storage and Addresses

Figure 9.7 illustrates the relationship between array storage and addresses for a six-element short array and a three-element int array.

By looking at Figure 9.7, you should be able to see why the following relationships are true:

Lines	Description
`1: x == 1000`	x without the array brackets is the address of the first element. Looking at Figure 9.7, you can see that x[0] is the address of 1000.
`2: &x[0] == 1000`	Line 2 shows this also. It can be read as the address of the first element of the array x is equal to 1000.
`3: &x[1] = 1002`	Line 3 shows that the address of the second element (subscripted as 1 in an array) is 1002. Again the figure can confirm this.

`4: expenses == 1250`
`5: &expenses[0] == 1250`
`6: &expenses[1] == 1254`
	Lines 4, 5, and 6 are virtually identical to 1, 2, and 3, respectively. They vary in the difference between the addresses of the two array elements. In the type short array x, the difference is 2 bytes, and in the type int array, expenses, the difference is 4 bytes.

Accessing Successive Arrays Elements Using a Pointer

How do you access these successive array elements using a pointer? You can see from these examples that a pointer must be increased by 2 to access successive elements of a type short array, and by 4 to access successive elements of a type int array. You can generalize and say that to access successive elements of an array of a particular data type, a pointer must be increased by sizeof(datatype). Remember from

Day 3, "Numeric Variables and Constants," the sizeof() operator returns the size in bytes of a C data type.

Example Program

The program in Listing 9.2 illustrates the relationship between addresses and the elements of different type arrays by declaring arrays of type short, float, and double, and by displaying the addresses of successive elements. In the program output, the exact addresses that your system displays may be different, but the relationships are the same: 2 bytes between short elements, 4 bytes between float elements, and 8 bytes between double elements. (Note: Some machines use different sizes for variable types. If your machine differs, the following output might have different size gaps; however, they will be consistent gaps.)

Listing 9.2: Display of the Addresses of Successive Array Elements

Code

```
 1: /* LIST0902.c: Day 9 Listing 9.2 */
 2: /* Demonstrates the relationship between */
 3: /* addresses and elements of arrays of */
 4: /* different data types. */
 5:
 6: #include <stdio.h>
 7:
 8: /* Declare three arrays and a counter variable. */
 9:
10: short s[10], x;
11: float f[10];
12: double d[10];
13:
14: int main(void) {
15:   /* Print the table heading. */
16:
17:   printf("\t\tShort\t\tFloat\t\tDouble");
18:   printf("\n================================");
19:   printf("====================");
20:
21:   /* Print the addresses of each array element. */
22:
23:   for (x = 0; x < 10; x++)
24:     printf("\nElement %d:\t%d\t\t%d\t\t%d", x,
25:            &s[x], &f[x], &d[x]);
26:
27:   printf("\n================================");
28:   printf("====================");
29:   return 0;
30: }
```

Output

```
                Short     Float     Double
============================================
Element 0:      1392        1414      1454
```

```
Element 1:      1394          1418       1462
Element 2:      1396          1422       1470
Element 3:      1398          1426       1478
Element 4:      1400          1430       1486
Element 5:      1402          1434       1494
Element 6:      1404          1438       1502
Element 7:      1406          1442       1510
Element 8:      1408          1446       1518
Element 9:      1410          1450       1526
================================================
```

Description

This listing takes advantage of the escape characters learned on Day 7, "Basic Input/Output." The printf() calls in lines 17 and 24 use the tab escape character (\t) to help format the table by aligning the columns.

Looking more closely at the listing, you can see that three arrays are created in lines 10, 11, and 12. Line 10 declares array s of type short, line 11 declares array f of type float, and line 12 declares array d of type double.

Line 17 prints the column headers for the table that will be displayed. Lines 18 and 19 along with lines 27 and 28 print dashed lines across the top and bottom of the table data. This is a nice touch to a report.

Lines 23, 24, and 25 are a for loop that prints each of the table's rows. The number of the element, x, is printed first. This is followed by the address of the element in each of the three arrays.

Topic 2.4.3: Pointer Arithmetic

Pointer Arithmetic

You have a pointer to the first array element; the pointer must increment by an amount equal to the size of the data type stored in the array. How do you access array elements using pointer notation? You use *pointer arithmetic*.

Incrementing and Decrementing

"Just what I don't need," you might think, "another kind of arithmetic to learn!" Don't worry. Pointer arithmetic is simple, and it makes using pointers in your programs much easier. You only have to be concerned with two pointer operations: incrementing and decrementing.

Incrementing Pointers
Increments of 1

When you increment a pointer, you are increasing its value. For example, when you increment a pointer by 1, pointer arithmetic automatically increases the pointer's value so that it points to the next array element. In other words, C "knows" the data type that the pointer points to (from the pointer declaration), and increases the address stored in the pointer by the size of the data type.

Example

Suppose that ptr_to_short is a pointer variable to some element of an short array. If you execute the statement

```
ptr_to_short++;
```

the value of ptr_to_short is increased by the size of type short (usually 2 bytes), and ptr_to_short now points to the next array element. Likewise, if ptr_to_float points to an element of a type float array, then

```
ptr_to_float++;
```

increases the value of ptr_to_float by the size of type float (usually 4 bytes).

Incrementing Pointers
Increments Greater Than 1

The same holds true for increments greater than 1. If you add the value n to a pointer, C increments the pointer by n array elements of the associated data type.

Therefore,

```
ptr_to_short += 4;
```

increases the value stored in ptr_to_short by 8, because it points 4 array elements ahead. Likewise,

```
ptr_to_float += 10;
```

increases the value stored in ptr_to_float by 40, because it points 10 array elements ahead.

Decrementing Pointers
Incrementing by a Negative Value

The same concepts hold true for decrementing a pointer that apply to incrementing

pointers. Decrementing a pointer is actually a special case of incrementing by adding a *negative* value. If you decrement a pointer with the -- or -= operators, pointer arithmetic automatically adjusts for the size of the array elements.

Listing 9.3 presents an example of how pointer arithmetic can be used to access array elements. By incrementing pointers, the program can step through all elements of the arrays efficiently.

Listing 9.3: Using Pointer Arithmetic and Pointer Notation to Access Array Elements

Code

```
 1: /* LIST0903.c: Day 9 Listing 9.3 */
 2: /* Demonstrates using pointer arithmetic to */
 3: /* access array elements with pointer notation. */
 4:
 5: #include <stdio.h>
 6: #define MAX 10
 7:
 8: /* Declare and initialize an integer array. */
 9:
10: int i_array[MAX] = { 0,1,2,3,4,5,6,7,8,9};
11:
12: /* Declare a pointer to int and then declare
13:  * an int variable. */
14:
15: int *i_ptr, count;
16:
17: /* Declare and initialize a float array. */
18:
19: float f_array[MAX] =
20:   {.0,.1,.2,.3,.4,.5,.6,.7,.8,.9};
21:
22: /* Declare a pointer to float. */
23:
24: float *f_ptr;
25:
26: int main(void) {
27:   /* Initialize the pointers. */
28:
29:   i_ptr = i_array;
30:   f_ptr = f_array;
31:
32:   /* Print the array values by using the pointers. */
33:
34:   for (count = 0; count < MAX; count++)
35:     printf("\n%d\t%f", *i_ptr++, *f_ptr++);
36:   return 0;
}
```

Output

```
0       0.000000
```

```
1       0.100000
2       0.200000
3       0.300000
4       0.400000
5       0.500000
6       0.600000
7       0.700000
8       0.800000
9       0.900000
```

Description

In this program, a defined constant named MAX is set to 10 on line 6; it is used throughout the listing. On line 10, MAX is used to set the number of elements in an array of ints named i_array. The elements in this array are initialized at the same time that the array is declared.

Line 14 declares two additional int variables. The first is a pointer named i_ptr. You know this is a pointer because an indirection operator (*) is used. The other variable is a simple type int variable named count.

In lines 18 – 19, a second array is defined and initialized. This array is of type float, contains MAX values, and is initialized with float values. Line 23 declares a pointer to a float named f_ptr.

The main() function is on lines 25 – 36. On lines 28 and 29, the program assigns the beginning address of the two arrays to the two pointers. Remember that an array name without the subscript is the same as the address of the array's first element.

In lines 33-34, a for statement uses the int variable count to count from 0 to the value of MAX. For each count, line 35 *dereferences* (accesses the values of) the two pointers and prints their values in a printf() function call. The increment operator then increments each of the pointers so that each points to the next element in the array before continuing with the next iteration of the for loop.

You might be thinking that the program in Listing 9.3 could just as well have used array subscript notation and dispensed with pointers altogether. This is true, and in simple programming tasks like this, the use of pointer notation doesn't offer any major advantages. As you start to write more complex programs, however, you should find the use of pointers advantageous.

Please remember that you cannot perform incrementing and

	decrementing operations on pointer constants. (An array name without brackets is a pointer constant.) However, you can perform arithmetic operations on a pointer constant. Also remember that when you're manipulating pointers to array elements, the C compiler does not keep track of the start and finish of the array. If you're not careful, you can increment or decrement the pointer so it points somewhere in memory before or after the array. There is something stored there, but it isn't an array element. You should keep track of pointers and where they're pointing.

Other Pointer Manipulations
Differencing
The only other pointer arithmetic operation is called *differencing*, which refers to subtracting two pointers. If you have two pointers to different elements of the same array, you can subtract them and find out how far apart they are. Again, pointer arithmetic automatically scales the answer so that it refers to array elements.

Thus, if ptr1 and ptr2 point to elements of an array (of any type), the expression

```
ptr1 - ptr2
```

tells you how far apart the elements are.

Other Pointer Manipulations
Comparisons
Pointer comparisons are valid only between pointers that point to the same array. Under these circumstances, the relational operators ==, !=, >, <, >=, and <= work properly. Lower array elements (that is, those having a lower subscript) always have a lower address than higher array elements.

Thus, if ptr1 and ptr2 point to elements of the same array, the comparison

```
ptr1 < ptr2
```

is true if ptr1 points to an earlier member of the array than ptr2 does. Click the Tip button on the toolbar.

DON'T try to do mathematical operations such as division, multiplication, and modulus on pointers. Adding (incrementing) and subtracting (differencing) pointers are acceptable.

DON'T forget that subtracting or adding to a pointer increases the pointer based on the size of the data type it points to, not by 1 or the number being added (unless it is a pointer to a 1-byte character).

DO understand the size of variable types on your PC. As you can begin to see, you need to know variable sizes when working with pointers and memory.

DON'T try to increment or decrement an array variable. Assign a pointer to the beginning address of the array and increment it (see Listing 9.3).

Topic 2.4.4: Summary: Pointer Operations

Six Allowable Operations

You can do a total of six operations with a pointer, all of which have been covered in this unit:

- *Assignment*. You can assign a value to a pointer. The value should be an address, obtained with the address of operator (&) or from a pointer constant (array name).
- *Indirection*. The indirection operator (*) (also known as the dereferencing operator) gives the value stored in the pointed-to location. This operator is also used when declaring the pointer.
- *address of*. You can use the address of operator to find the address of a pointer, so you can have pointers to pointers. This is an advanced topic and is covered on Day 15, "More on Pointers."
- *Incrementing*. C "knows" the data type that the pointer points to (from the pointer declaration), and increases the address stored in the pointer by the size of the data type.
- *Differencing*. If you have two pointers to different elements of the same array, you can subtract them and find out how far apart they are.
- *Comparisons*. Comparisons use the relational operators ==, !=, >, <, >=, and <=. They are valid only with two pointers that point to the same array.

Operations Not Allowed

Many arithmetic operations that can be performed with regular variables, such as multiplication and division, do not make sense with pointers. The C compiler does

not allow them. For example, if ptr is a pointer, the statement

```
ptr *= 2;
```

generates an error message.

Topic 2.5: Pointer Cautions

One Serious Error
When you are writing a program that uses pointers, you must avoid one serious error: using an uninitialized pointer on the left side of an assignment statement. Remember that initializing a pointer involves assigning an address of a variable to it, for example `p_myVar=&myVar.`

Example

Properly Initialize Pointers
The left side of an assignment statement is the most dangerous place to use an uninitialized pointer. Other errors, although less serious, can also result from using an uninitialized pointer anywhere in your program, so be sure your program's pointers are properly initialized before you use them. You must do this yourself. The compiler doesn't watch out for you!

The statement

```
int *ptr;
```

declares a pointer to type int. This pointer is not yet initialized, so it doesn't point to anything. To be more exact, it doesn't point to anything *known*. An uninitialized pointer has some value; you just don't know what it is. In many cases it is zero. If you use an uninitialized pointer in an assignment statement, therefore, this is what happens:

```
*ptr = 12;
```

The value 12 is assigned to whatever address ptr points to. That address can be almost anywhere in memory—where the operating system is stored or somewhere in the program's code. The 12 that is stored there may overwrite some important information, and the result can be anything from strange program errors to a full system crash.

Topic 2.6: Array Subscript Notation and Pointers

Equivalence of Array Subscript Notation and Array Pointer Notation
An array name without brackets is a pointer to the array's first element. You can, therefore, access the first array element using the indirection operator (also known as the dereferencing operator). If `array[]` is a declared array, the expression `*array` is the array's first element, `*(array + 1)` is the array's second element, and so on. If you generalize for the entire array, the following relationships hold true:

```
*(array) == array[0]
*(array + 1) == array[1]
*(array + 2) == array[2]
...
*(array + n) == array[n]
```

This illustrates the equivalence of array subscript notation and array pointer notation. You can use either in your program; the C compiler sees them as two different ways of accessing array data using pointers.

Topic 2.7: Passing Arrays to Functions

Passing Arrays to Functions
This unit has already discussed the special relationship that exists in C between pointers and arrays. This relationship comes into play when you need to pass an array as an argument to a function. The only way you can pass an array to a function is by means of a pointer.

An Argument Cannot Be an Entire Array
As you learned on Day 5, "Functions: The Basics," an argument is a value that the calling program passes to a function. It can be an int, a float, or any other simple data type, but it has to be a single numerical value. It can be a single array element, but it cannot be an entire array.

Pass the Array's Address
What if you need to pass an entire array to a function? Well, you can have a pointer to an array, and that pointer is a single numeric value (the address of the array's first element). If you pass that value to a function, the function "knows" the address of the array and can access the array elements using pointer notation.

Pass the Array's Size

Consider another problem, however. If you write a function that takes an array as an argument, you want a function able to handle arrays of different sizes. For example, you could write a function that finds the largest element in an integer array. The function wouldn't be much use if it were limited to dealing with arrays of one fixed size (number of elements).

How does the function know the size of the array whose address it was passed? Remember, the value passed to a function is a pointer to the first array element. It could be the first of 10 elements or the first of 10,000. There are two methods for letting a function "know" an array's size.

Method 1

You can identify the last array element by storing some special value there. As the function processes the array, it looks for that value in each element. When the value is found, the end of the array has been reached. The disadvantage of this method is that it forces you to reserve some value as the end-of-array indicator, reducing the flexibility you have for storing real data in the array.

Method 2

The other method is more flexible and straightforward: pass the function the array size as an argument. This can be a simple type int argument. Thus, the function is passed two arguments: a pointer to the first array element and an integer specifying the number of elements in the array. This second method is used in this course.

Listing 9.4 accepts a list of values from the user and stores them in an array. It then calls a function named largest(), passing the array (both pointer and size). The function finds the largest value in the array and returns it to the calling program.

Listing 9.5 shows the other way of passing arrays to functions.

Listing 9.4: Demonstration of Passing an Array to a Function

```
Code 1:  /* LIST0904.c: Day 9 Listing 9.4 */
      2:  /* Passing an array to a function. */
      3:
      4:  #include <stdio.h>
      5:  #define MAX 10
      6:
      7:  int array[MAX], count;
      8:
      9:  int largest(int x[], int y);
     10:
     11:  int main(void) {
```

```
12:   /* Input MAX values from the keyboard. */
13:
14:   for (count = 0; count < MAX; count++) {
15:     printf("Enter an integer value: ");
16:     scanf("%d", &array[count]);
17:   }
18:
19:   /* Call the function and display the return */
20:   /* value. */
21:
22:   printf("\n\nLargest value = %d",
23:           largest(array, MAX));
24:   return 0;
25: }
26:
27: /* Function largest() returns the largest value */
28: /* in an integer array. */
29:
30: int largest(int x[], int y) {
31:   int count, biggest = -12000;
32:
33:   for (count = 0; count < y; count++) {
34:     if (x[count] > biggest)
35:       biggest = x[count];
36:   }
37:   return biggest;
38: }
```

Description

There is a function prototype in line 9 and a function header in line 30 that are identical. They read

```
int largest(int x[], int y)
```

Most of this line should make sense to you: largest() is a function that returns an int to the calling program; its second argument is an int represented by the parameter y. The only thing new is the first parameter int x[ ], which indicates that the first argument is a pointer to type int, represented by the parameter x. You could also write the function declaration and header as follows:

```
int largest(int *x, int y);
```

This is equivalent to the first form; both int x[ ] and int *x mean "pointer to int." The first form may be preferable because it reminds you that the parameter represents a pointer to an array. Of course, the pointer doesn't know that it points to an array, but the function uses it that way.

Now look at the function largest(). When it is called, the parameter x holds the value of the first argument and is therefore a pointer to the first element of the array. You can use x anywhere an array pointer could be used. In largest(), the array elements are accessed using subscript notation on lines 34 and 35. You also could have used pointer notation, rewriting the if loop to read

```
for (count = 0; count < y; count++){
    if (*(x+count) > biggest)
        biggest = *(x+count);

}
```

In either case, the for loop checks each value in the array to see if it is greater than -1200. Remember that x is a pointer to the first element in the array, and x[count] is another way of saying *(x + count).

Listing 9.5: An Alternative Way for Passing an Array to a Function

```
Code 1:  /* LIST0905.c: Day 9 Listing 9.5 */
     2:  /* Passing an array to a function. */
     3:  /* Alternative way. */
     4:
     5:  #include <stdio.h>
     6:
     7:  #include <limits.h>
     8:
     9:  #define MAX 10
    10:
    11:  int array[MAX + 1], count;
    12:
    13:  int largest(int x[]);
    14:
    15:  int main(void) {
    16:    /* Input MAX values from the keyboard. */
    17:
    18:    for (count = 0; count < MAX; count++) {
    19:      printf("Enter an integer value: ");
    20:      scanf("%d", &array[count]);
    21:
    22:      if (array[count] == 0)
    23:        count = MAX;      /* Will exit for loop. */
    24:    }
    25:    array[MAX] = 0;
    26:
```

```
27:   /* Call the function and display the return */
28:   /* value. */
29:
30:   printf("\n\nLargest value = %d", largest(array));
31:   return 0;
32: }
33:
34: /* Function largest() returns the largest value */
35: /* in an integer array. */
36:
37: int largest(int x[]) {
38:   int count, biggest = INT_MIN;
39:
40:   for (count = 0; x[count] !=0; count++) {
41:     if (x[count] > biggest)
42:       biggest = x[count];
43:   }
44:   return biggest;
45: }
```

Description

This program uses a largest() function that has the same functionality as the previous listing. The difference is that only the array tag is needed. The for loop in line 40 continues looking for the largest value until it encounters a 0, at which point it knows it is done.

Looking at the early parts of the listing, you can see the differences between Listing 9.4 and Listing 9.5. First in line 11, you need to add an extra element to the array to store the value that flags the end. In lines 22 and 23 an if statement is added to see whether the users entered a zero, thus signaling that they are done entering values. If zero is entered, count is set to its maximum value so that the for loop can be cleanly exited. Line 25 ensures that the last element is a zero in case users entered the maximum number of values (MAX).

By adding the extra commands when entering the data, you can make the largest() function work with any size of array; however, there is one catch. What happens if you forget to put a zero at the end of the array? Then largest() continues past the end of the array, comparing values in memory until it finds a zero.

As you can see, passing an array to a function is not particularly difficult. You simply pass a pointer to the array's first element. In most situations, you also need to pass the number of elements in the array. In the function, the pointer value can be used to access

	the array elements with either subscript or pointer notation. Recall from Day 5, "Functions: The Basics," that when a simple variable is passed to a function, only a copy of the variable's value is passed. The function can use the value, but cannot change the original variable because it doesn't have access to the variable itself. When you pass an array to a function, things are different. A function is passed the array's address, not just a copy of the values in the array. The code in the function is working with the actual array elements and *can* modify the values stored in the array.

Topic 2.8: Day 9 Q&A

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

Question 1
Why are pointers so important in C?

Answer
Pointers give you more control of the computer and your data. When used with functions, pointers enable you to change the values of variables that were passed, regardless of where they have originated. On Day 15, "More on Pointers," you will learn additional uses for pointers.

Question 2
How does the compiler know the difference among * for multiplication, * dereferencing, and * for declaring a pointer?

Answer
The compiler interprets the different uses of the asterisk based on the context in which it is used. If the statement being evaluated starts with a variable type, it can be assumed that the asterisk is for declaring a pointer. If the asterisk is used with a variable that has been declared as a pointer, but not in a variable declaration, the asterisk is assumed to dereference. If it is used in a mathematical expression, but not with a pointer variable, the asterisk can be assumed to be the multiplication operator.

Question 3
What happens if I use the address of operator (&) on a pointer?

Answer
You get the address of the pointer variable. Remember, a pointer is just another variable that holds the address of the variable it points to.

Question 4
Are variables always stored in the same location?

Answer
No. Each time a program runs, its variables can be stored at different addresses. You should never assign a constant address value to a pointer.

Topic 2.9: Day 9 Think About Its

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

Method 1
Pass the length of the array as a parameter to the function.

Method 2
Have a special value in the array, such as NULL, to signify the array's end.

Topic 2.10: Day 9 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**
Show a declaration for a pointer to a type char variable. Name the pointer char_ptr.

Answer

To declare a pointer to a character, do the following:

```
char *char_ptr;
```

*** Exercise 2**
If you have a type int variable named cost, how would you declare and initialize a pointer named p_cost that points to that variable?

Answer

The following declares a pointer to cost, and then assigns the address of cost, (&cost), to it.

```
int *p_cost;
p_cost = &cost;
```

*** Exercise 3**
Continuing with exercise two, how would you assign the value 100 to the variable cost using both direct access and indirect access?

Answer

Direct access: `cost = 100;` Indirect access: `*p_cost = 100;`

*** Exercise 4**
Continuing with exercise three, how would you print the value of the pointer, plus the value being pointed to?

Answer

```
printf( "Pointer value: %d, points at value: %d", p_cost,
*p_cost);
```

Since p_cost points to the variable cost, this prints out the address of the variable cost, and then the value of the variable cost.

*** Exercise 5**
Show how to assign the address of a float value called radius to a pointer.

Answer

```
float *variable = &radius;
```

*** Exercise 6**
Show two ways to assign the value 100 to the third element of data[ ].

Answer

```
data[2] = 100;
*(data + 2) = 100;
```

*** Exercise 7**
Write a function named sumarrays() that accepts two arrays as arguments, totals all values in both arrays, and returns the total to the calling program.

Answer

```
int sumarrays(int x1[], int len_x1, int x2[], int len_x2) {
  int total = 0, count = 0;

  for (count = 0; count < len_x1; count++)
    total += x1[count];

  for (count = 0; count < len_x2; count++)
    total += x2[count];

  return total;
}
```

*** Exercise 8**

Use the function created in exercise seven in a simple program.

Answer

```
/* EXER0908.c: Day 9 Exercise 9.8 */

#include <stdio.h>

#define MAX1 5
#define MAX2 8

int array1[MAX1] = { 1, 2, 3, 4, 5 };
int array2[MAX2] = { 1, 2, 3, 4, 5, 6, 7, 8,};
int total;

int sumarrays(int x1[], int len_x1, int x2[], int len_x2);

int main(void) {
  total = sumarrays(array1, MAX1, array2, MAX2);
  printf("The total is %d", total);
  return 0;
}

int sumarrays(int x1[], int len_x1, int x2[], int len_x2) {
  int total = 0, count = 0;

  for (count = 0; count < len_x1; count++)
    total += x1[count];

  for (count = 0; count < len_x2; count++)
    total += x2[count];

  return total;
}
```

*** Exercise**

Write a function named addarrays() that accepts two arrays which are the same size. The function should add each element in the arrays together and place the values in a third array.

Answer

Here is one possible answer.

```
/* EXER0909: Day 9 Exercise 9 */
/* Function addarrays() totals the elements in the */
/* first 2 arrays and places the sum in the */
/* elements of the third array.*/

void addarrays(int x[], int y[], int totarray[], int arrsize) {

  int count;

  for (count = 0; count < arrsize; count++)
    totarray[count] = x[count] + y[count];

  return;
}
```

*** Exercise 1**

Modify the function in exercise nine to return a pointer to the array containing the totals. Place this function in a program that also prints the values in all three arrays.

Answer

Here is one possible answer.

```
/* EXER0910: Day 9 Exercise 10 */
/* Combine the attendance of boys and girls in */
/* several classes to determine the total */
/* attendance of each class. */

#include <stdio.h>

#define MAX 10

int boys[MAX] = {7, 10, 5, 8, 13, 7, 13, 9, 11, 9};
int girls[MAX] = { 8, 9, 12, 16, 7, 10, 11, 13, 10, 9};
int attend[MAX];
int count;
int * addarrays(int x[], int y[], int z[], int arrsize);
int *arrptr;

int main(void) {
```

```
  arrptr = addarrays(boys, girls, attend, MAX);

  /* Print totals for each class. */

  printf("Totals per Class\n");
  printf("Class #\tBoys\tGirls\tTotal\n");
  for (count = 0; count < MAX; count++)  {
    printf("  %d\t  %d\t  %d\t  %d\n",
          (count + 1), boys[count], girls[count],
          *(arrptr + count));
  }

  return 0;
}

/* Function addarrays() totals the elements */
/* in the first 2 arrays and places the sum */
/* in the elements of the third array. */
/* The function returns a pointer to the array */
/* containing the totals. */

int * addarrays(int x[], int y[], int totarray[], int arrsize) {
  int count;

  for (count = 0; count < arrsize; count++)
    totarray[count] = x[count] + y[count];

  return (totarray);
}
```

Topic 2.11: Day 9 Summary

Definition of a Pointer This unit introduced you to pointers, a central part of C programming. A pointer is a variable that holds the address of another variable; a pointer is said to "point to" the variable whose address it holds.

Address of and Indirection Operators
The two operators needed with pointers are the address of operator (&) and the indirection operator (*) (also known as the dereferencing operator). When placed before a variable name, the address of operator returns the variable's address. When placed before a pointer name, the indirection operator returns the value of the pointed-to variable.

Pointers and Arrays

Pointers and arrays have a special relationship. An array name without brackets is a pointer to the array's first element. The special features of pointer arithmetic make it easy to access array elements using pointers. Array subscript notation is in fact a special form of pointer notation.

Passing Arrays to Functions
You also learned to pass arrays as arguments to functions by passing a pointer to the array. Once the function "knows" the array's address and length, it can access the array elements using either pointer notation or subscript notation.

Unit 3. Day 10 Characters and Strings

A *character* is a single letter, numeral, punctuation mark, or other such symbol. A *string* is any sequence of characters. Strings are used to hold text data—comprised of letters, numerals, punctuation marks, and other symbols. Clearly, characters and strings are extremely useful in many programming applications.

Today, you learn

- How to use C's char data type to hold single characters.
- How to create arrays of type char to hold multiple-character strings.
- How to initialize characters and strings.
- How to use pointers with strings.
- How to print and input characters and strings.

Topic 3.1: The char Data Type

A Numeric Type

C uses the char data type to hold characters. You saw on Day 3, "Numeric Variables and Constants," that char is one of C's numeric integer data types. If char is a numeric type, how can it be used to hold characters?

How C Stores Characters

The answer lies in how C stores characters. Your computer's memory stores all data in numeric form. There is no direct way to store characters. A numeric code exists for each character, however. This is called the *ASCII code* or the ASCII character set. (ASCII stands for American Standard Code for Information Interchange.) The code assigns values between 0 and 255 for upper- and lowercase letters, numeric digits, punctuation marks, and other symbols.

For example, 97 is the ASCII code for the letter a. When you store the character a in a type char variable, you're really storing the value 97. Because the allowable numeric range for type char matches the standard ASCII character set, char is ideally suited for storing characters.

char Variables: Character or Number?

At this point, you might be a bit puzzled. If C stores characters as numbers, how does your program know whether a given type char variable is a character or a number? As you learn later, declaring a variable as type char is not enough; you must do something else with the variable.

- If a char variable is used somewhere in a C program where a character is expected, it is interpreted as a character.
- If a char variable is used somewhere in a C program where a number is expected, it is interpreted as a number.

This gives you some understanding of how C uses a numeric data type to store character data. Now you can go on to the details.

Topic 3.2: Using Character Variables

Declaring and Initializing
Like other variables, you must declare chars before using them, and you can initialize them at the time of declaration. Click the Examples link.

Examples

Here are some examples:

```
char a, b, c;    /* Declare 3 uninitialized char */
                  * variables */

char code = 'x'; /* Declare the char variable */
                  * named code and store                  * the
character x there */

code = '!';   /* Store ! in the variable named code */
```

Literal Character Constants
To create literal character constants, enclose a single character in single quotation marks. The compiler automatically translates literal character constants into the corresponding ASCII codes, and the numeric code value is assigned to the variable.

Symbolic Character Constants
You can create symbolic character constants by using either the #define directive or the const keyword. Click the Example link. Then click the Tip button on the toolbar.

Example

DO use %c to print the character value of a number.

DON'T use double quotations when initializing a character variable.

DO use single quotations when initializing a variable.

DON'T try to put extended ASCII character values into a type signed char variable.

DO look at the ASCII chart to see the interesting characters that can be printed.

```
#define EX 'x'
char code = EX;      /* Sets code equal to 'x' */
const char A = 'Z';
```

Example Programs
Now that you know how to declare and initialize character variables, it's time for a demonstration. The program in Listing 10.1 illustrates the numeric nature of character storage using the printf() function you learned on Day 7, "Basic Input/Output." The function printf() can be used to print both characters and numbers. The format string %c instructs printf() to print a character, whereas %d instructs it to print a decimal integer. Listing 10.1 initializes two type char variables and prints each one, first as a character and then as a number.

The program in Listing 10.2 demonstrates printing some of the extended ASCII characters.

Listing 10.1: Demonstration of the Numeric Nature of Type char Variables

```
Code  1: /* LIST1001.c: Day 10 Listing 10.1 */
      2: /* Demonstrates the numeric nature of char */
      3: /* variables */
      4:
      5: #include <stdio.h>
      6:
      7: /* Declare and initialize two char variables. */
      8:
      9: char c1 = 'a';
     10: char c2 = 90;
     11:
     12: int main(void) {
     13:   /* Print variable c1 as a character */
     14:   /* then as a number. */
     15:
     16:   printf("\nAs a character, variable c1 is %c",
     17:          c1);
```

```
18:   printf("\nAs a number, variable c1 is %d", c1);
19:
20:   /* Do the same for variable c2. */
21:
22:   printf("\nAs a character, variable c2 is %c",
23:           c2);
24:   printf("\nAs a number, variable c2 is %d", c2);
25:
26:   return 0;
27: }
```

Output

```
As a character, variable c1 is a
As a number, variable c1 is 97
As a character, variable c2 is Z
As a number, variable c2 is 90
```

Description

You learned on Day 3, "Numeric Variables and Constants," that the allowable range for a variable of type char goes only to 127, whereas the ASCII codes go to 255. The ASCII codes are actually divided into two parts. The standard ASCII codes go only to 127; this range includes all letters, numbers, punctuation marks, and other keyboard symbols. The codes 128 – 255 are the extended ASCII codes and represent special characters such as foreign letters and graphics symbols. Thus, for standard text data, you can use type char variables; if you want to print the extended ASCII characters, you must use unsigned char.

Listing 10.2: Printing Extended ASCII Characters

Code

```
1:  /* LIST1002.c: Day 10 Listing 10.2 */
2:  /* Demonstrates printing extended ASCII */
3:  /* characters. MS-DOS Application Only */
4:
5:  #include <stdio.h>
6:
7:  unsigned char x;
8:  /* Must be unsigned for extended ASCII. */
9:
10: int main(void) {
11: /* Print extended ASCII characters 180 - 203 */
12:
13:   for (x = 180; x < 204; x++) {
14:   printf("\nASCII code %d is character %c", x, x);
15:   }
16:   return 0;
17: }
```

Description

Looking at this program, you see that line 7 declares an unsigned

	character variable, x. This gives a range of 0 to 255. As with other numeric data types, you must not initialize a char variable to a value outside of the allowed range or you get unpredictable results. In line 13, x is not initialized outside the range; instead, it is initialized to 180. In the for statement, x is incremented by 1 until it reaches 204. Each time x is incremented, line 14 prints the value of x and the character value of x. Remember that %c prints the character, or ASCII, value of x.

Topic 3.3: Using Strings

Using Strings
Variables of type char can hold only a single character, so they have limited usefulness. You also need a way to store *strings,* which are sequences of characters. An individual's name or address are examples of strings. Although there is no special data type for strings, C handles this type of information with arrays of characters.

Topic 3.3.1: Arrays of Characters

Storing Strings
To hold a string of six characters, for example, you need to declare an array of type char with seven elements. Arrays of type char are declared like arrays of other data types. Click the Example link.

Example

Terminating Null Character
"But wait," you may be thinking, "it's a 10-element array, so why can it hold only nine characters? " In C, a string is defined as a sequence of characters ending with the null character, a special character represented by \0. Although it's represented by two characters (backslash-zero), the null character is interpreted as a single character and has the ASCII value of 0. It's one of C's escape sequences, covered on Day 7, "Basic Input/Output." Click the Example link.

Example

The statement

```
char string[10];
```

declares a ten-element array of type char. This array could be used to hold a string of nine or fewer characters.

When a C program stores the string Alabama, for example, it actually stores the seven characters A, l, a, b, a, m, and a, followed by the null character \0, for a total of eight characters. Thus, a character array can hold a string of characters numbering one less than the total number of elements in the array.

Size of Character Array
A type char variable is one byte in size, so the number of bytes in an array of type char variables is the same as the number of elements in the array.

Topic 3.3.2: Initializing Character Arrays

Initializing an Array Within Its Declaration
Like other C data types, character arrays can be initialized when they are declared.

Assigning Values Element by Element
Character arrays can be assigned values element by element, as shown here:

```
char string[10] =
  { 'A', 'l', 'a', 'b', 'a', 'm', 'a', '\0' };
```

Assigning Values Using a Literal String
It's more convenient, however, to use a *literal string,* which is a sequence of characters enclosed in double quotes:

```
char string[10] = "Alabama";
```

When you use a literal string in your program, the compiler automatically adds the terminating null character at the end of the string. If you do not specify the number of subscripts when you declare an array, the compiler calculates the size of the array for you. Thus,

```
char string[] = "Alabama";
```

creates and initializes an eight-element array.

Terminating Null Character
Remember that strings require a terminating null character. The C functions that manipulate strings (covered on Day 17, "Manipulating Strings") determine string length by looking for the null character. The functions have no other way of recognizing the end of the string. If the null character is missing, your program thinks that the string extends until the next null character in memory. Pesky program bugs can result from this sort of error.

Topic 3.4: Strings and Pointers

Accessing Strings by Array Names
You've seen that strings are stored in arrays of type char, with the end of the string (which may not occupy the entire array) marked by the null character. Because the end of the string is marked, all you need to define a given string is something that points to its beginning. (Is *points* the right word? Indeed it is!)

With that hint, you might be leaping ahead of the game. From Day 9, "Pointers," you know that the name of an array is a pointer to the first element of the array. Therefore, for a string that's stored in an array, you need only the array name in order to access it. In fact, using the array's name is C's standard method for accessing strings.

Using the String Manipulation Functions
To be more precise, using the array's name for accessing strings is the method the C library functions expect. The C standard library includes a number of functions that manipulate strings. (These functions are covered on Day 17, "Manipulating Strings.") To pass a string to one of these functions, you pass the array name. The same is true for the string display functions printf() and puts(), discussed later in this unit.

Strings Without Arrays
You may have noticed that the phrase, "strings stored in an array," is used. Does this imply that some strings are not stored in arrays? Indeed it does, and the next section explains how.

Topic 3.5: Strings Without Arrays

Storing Strings

From the last section, you know that a string is defined by the character array's name — which is a type char pointer to the beginning of the string—and by a null character — which marks its end—that the space occupied by the string in an array is incidental. In fact, the only purpose the array serves is to provide allocated space for the string.

Finding Memory Storage Space

What if you could find some memory storage space without allocating an array? You could then store a string with its terminating null character there instead. A pointer to the first character could serve to specify the string's beginning just as if the string were in an allocated array. How do you go about finding memory storage space? There are two methods: one allocates space for a literal string when the program is compiled and the other uses the malloc() function to allocate space while the program is executing, a process known as dynamic allocation.

Topic 3.5.1: Allocating String Space at Compilation

Storing Strings

The start of a string, as mentioned earlier, is indicated by a pointer to a variable of type char. You might recall how to declare such a pointer:

```
char *message;
```

This statement declares a pointer to a variable of type char named message. At present, it doesn't point to anything, but what if you change the pointer declaration to read:

```
char *message = "Great Caesar's Ghost!";
```

When this statement executes, the string Great Caesar's Ghost! (with terminating null character) is stored somewhere in memory, and the pointer message is initialized to point at the first character of the string. Don't worry where in memory the string is stored; it's handled automatically by the compiler. Once defined, message is a pointer to the string and can be used as such.

Equivalent Notation

The preceding declaration/initialization is equivalent to

```
char message[] = "Great Caesar's Ghost!";
```

and the two notations `*message` and `message[]` are equivalent. They both mean "a pointer to."

Suitability of Allocating at Compilation
This method of allocating space for string storage is fine when you know what you need while writing the program. What if the program has varying string storage needs, depending on user input or other factors that are unknown when you are writing the program? You use the malloc() function, which enables you to allocate storage space "on the fly."

Topic 3.5.2: The malloc() Function

Description
malloc() is one of C's *m*emory *alloc*ation functions. When you call malloc(), you pass it the number of bytes of memory needed. malloc() finds and reserves a block of memory of the required size and returns the address of the first byte in the block. You don't need to worry about where the memory is found; it's handled automatically.

void Return Type
The malloc() function returns an address, and its return type is a pointer to type `void`. Why `void`? A pointer to type `void` is compatible with all data types. Because the memory allocated by malloc() can be used to store any of C's data types, the `void` return type is appropriate. Click the Tip button on the toolbar.

DON'T allocate more memory than you need. Not everyone has a lot of memory, so you should try to use it sparingly.
DON'T try to assign a new string to a character array that was previously allocated only enough memory to hold a smaller string. For example,

```
char a_string[] = "NO";
```

In this declaration, a_string points to "NO". If you try to assign "YES" to this array, you could have serious problems. The array initially could hold only three characters, 'N', 'O', and a null. "YES" is four characters, 'Y', 'E', 'S' and a null. You have no idea what the fourth character, null, overwrites.

Allocating Storage for a Single Type char
You can use malloc() to allocate memory to store a single type char. Click the Example link.

Example

Allocating Storage for a String

Allocating storage for a string with malloc() is almost identical to using malloc() to allocate space for a single variable of type char. The main difference is that you need to know the amount of space to allocate—the maximum number of characters in the string. This maximum depends on the needs of your program. Click the Example link.

Example

First, declare a pointer to type char:

```
char *ptr;
```

Next, call malloc() and pass the size of the desired memory block. Because a type char occupies one byte, you need a block of one byte. The value returned by malloc() is assigned to the pointer.

```
ptr = malloc(1);
```

This statement allocates a memory block of one byte and assigns its address to ptr. Unlike variables that are declared in the program, this byte of memory has no name. Only the pointer can reference the variable. For example, to store the character 'x' there, you would write

```
*ptr = 'x';
```

For this example, say you want to allocate space for a string of 99 characters, plus one for the terminating null character, for a total of 100. First, you declare a pointer to type char, and then call malloc().

```
char *ptr;
ptr = malloc(100);
```

Now, ptr points to a reserved block of 100 bytes that can be used for string storage and manipulation. You can use ptr, just as if your program had explicitly allocated that space with the following array declaration.

```
char ptr[100];
```

Testing the Return Value

Using malloc() enables your program to allocate storage space as needed in response to demand. Of course, available space is not unlimited; it depends on the amount of memory installed in your computer and on the program's other storage requirements. If not enough memory is available, malloc() returns 0 (null). Your program should test the return value of malloc(), so you are sure the memory requested was allocated successfully. You always should test malloc()'s return value against the symbolic constant NULL, which is defined in STDLIB.H.

Example Program

Listing 10.3 illustrates the use of malloc(). Any program using malloc() must include the header file STDLIB.H.

Listing 10.3: Using the malloc() Function to Allocate Storage Space for String Data

```
Code  1: /* LIST1003.c: Day 10 Listing 10.3 */
      2: /* Demonstrates the use of malloc() to allocate */
      3: /* storage space for string data. */
      4:
      5: #include <stdio.h>
      6: #include <stdlib.h>
      7:
      8: char count, *ptr, *p;
      9:
     10: int main(void) {
     11:   /* Allocate a block of 35 bytes. Test for */
     12:   /* success. The exit() library function */
     13:   /* terminates the program. */
     14:
     15:   ptr = malloc(35 * sizeof(char));
     16:
     17:   if (ptr == NULL) {
     18:     puts("Memory allocation error.");
     19:     exit(1);
     20:   }
     21:
     22:   /* Fill the string with values 65 through 90, */
     23:   /* which are the ASCII codes for A - Z. */
     24:
     25:   /* p is a pointer used to step through the */
     26:   /* string. You want ptr to remain pointed */
     27:   /* at the start of the string. */
     28:
     29:   p = ptr;
```

```
30:
31:   for (count = 65; count < 91; count++)
32:     *p++ = count;
33:
34:   /* Add the terminating null character. */
35:
36:   *p = '\0';
37:
38:   /* Display the string on the screen. */
39:
40:   puts(ptr);
41:   return 0;
42: }
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Description

This program uses malloc() in a simple way. Line 6 includes the STDLIB.H header file needed for malloc(), and line 5 includes the STDIO.H header file for the puts() functions. Line 8 declares two pointers and a character variable used later in the listing. None of these variables are initialized, so they should not be used —yet!

The malloc() function is called in line 15 with a parameter of 35 multiplied by *the size of* a char. Could you have just put 35? Yes, but you are making an assumption that everyone running this program will be using a computer that stores char type variables as one byte in size. Remember from Day 3, "Numeric Variables and Constants," that different compilers can use different size variables. Using the sizeof operator is an easy way to create portable code.

Never assume that malloc() gets the memory you tell it to get. In fact, you are not telling it to get memory, you are asking it. Line 17 shows the easiest way to check to see whether malloc() provided the memory. If the memory was allocated, ptr points to it; otherwise, ptr is null. If the program failed to get the memory, lines 18 and 19 display an error message and gracefully exit the program.

Line 29 initializes the other pointer declared in line 7, p. It is assigned the same address value as ptr. A for loop uses this new pointer to place values into the allocated memory. Looking at line 31, you see that count is initialized to 65 and incremented by 1 until it reaches 91. For each loop of the for statement, the value of count is assigned to the address pointed to by p. Notice that

	each time count is incremented, the address pointed to by p is also incremented. This means that each value is placed one after the other in memory. You should have noticed that numbers are being assigned to count, which is a type char variable. Don't forget the discussion about the ASCII characters and their numeric equivalents. The number 65 is equivalent to A, 66 = B, 67 = C, etc. The for loop ends after the alphabet is assigned to the memory locations pointed to. Line 36 caps off the character values pointed to by putting a null at the final address pointed to by p. By appending the null, you now can use these values as a string. Remember that ptr still points to the first value, A, so if you use it as a string, it prints every character until it reaches the null. Line 40 uses puts() to prove this point and to show the results of what has been done.

Topic 3.6: Displaying Strings and Characters

Displaying Strings
If your program uses string data, it probably needs to display the data on the screen at some time. String display is usually done with either the puts() function or the printf() function.

Topic 3.6.1: puts() Function

Description
You've seen the puts() library function used in some of the programs given in this course. The puts() function puts a string on the screen—hence its name. A pointer to the string to be displayed is the only argument puts() takes. Because a literal string evaluates as a pointer to a string, puts() can be used to display literal strings as well as string variables. The puts() function automatically inserts a newline character at the end of each string that it displays, so each subsequent string displayed with puts() is on its own line.

Example Program
The program in Listing 10.4 illustrates the use of puts().

Listing 10.4: Using the puts() Function to Display Text on the Screen

Code

```
 1: /* LIST1004.c: Day 10 Listing 10.4 */
 2: /* Demonstrates displaying strings with puts(). */
 3:
 4: #include <stdio.h>
 5:
 6: char *message1 = "C";
 7: char *message2 = "is the";
 8: char *message3 = "best";
 9: char *message4 = "programming";
10: char *message5 = "language!!";
11:
12: int main(void) {
13:   puts(message1);
14:   puts(message2);
15:   puts(message3);
16:   puts(message4);
17:   puts(message5);
18:   return 0;
19: }
```

Output

```
C
is the
best
programming
language!!
```

Description

This is a fairly simple listing to follow. Because puts() is a standard input/output function, the STDIO.H header file "needs" to be included as done on line 4. Lines 6 – 10 declare and initialize 5 different message variables. Each of these variables is a character pointer, or string variable. Lines 13–17 use the puts() function to print each string.

Topic 3.6.2: printf() Function

Description

You also can display strings with the printf() library function. Recall from Day 7, "Basic Input/Output," that printf() uses a format string and conversion specifiers (also known as format specifiers) to shape its output.

The Conversion Specifier %s

To display a string, use the conversion specifier %s. When printf() encounters a %s in

its format string, the function matches the %s with the corresponding argument in its argument list. For a string, this argument must be a pointer to the string that you want displayed. The printf() function displays the string on-screen, stopping when it reaches the string's terminating null character. Click the Example link.

Example

```
char *str = "A message to display";
printf("%s", str);
```

To Display Multiple Strings
You also can display multiple strings and mix them with literal text and/or numeric variables. Click the Example link. Then click the Note button.

Example

For now, this information should be sufficient for you to be able to display string data in your programs. Complete details on using printf() are given on Day 14, "Working with the Screen, Printer, and Keyboard."

```
char *bank = "First Federal";
char *name = "John Doe";
int balance = 1000;
printf("The balance at %s for %s is %d.",
        bank, name, balance);
```

The resulting output is

```
The balance at First Federal for John Doe is 1000.
```

Topic 3.7: Reading Strings from the Keyboard

Reading Strings
In addition to displaying strings, programs often need to accept inputted string data from the user, via the keyboard. The C library has two functions that can be used for this purpose, gets() and scanf(). Before you can read in a string from the keyboard,

however, you must have somewhere to put it. Space for string storage can be created with either of the methods discussed earlier today, an array declaration or the malloc() function.

Topic 3.7.1: Inputting Strings with the gets() Function

Description

The gets() function gets a string from the keyboard. When gets() is called, it reads all characters typed at the keyboard up to the first newline character (which you generate by pressing Enter). The function discards the newline, adds a null character, and gives the string to the calling program. The string is stored at the location indicated by a pointer to type char passed to gets(). A program that uses gets() must #include the file STDIO.H. Listing 10.5 presents an example. Listing 10.6 presents a modified example that accounts for the user entering a blank line.

Listing 10.5. Using gets() to Input String Data from the Keyboard

Code

```
 1: /* LIST1005.c: Day 10 Listing 10.5 */
 2: /* Demonstrates using gets() library function. */
 3:
 4: #include <stdio.h>
 5:
 6: /* Allocate a character array to hold input. */
 7:
 8: char input[81];
 9:
10: int main(void) {
11:   puts("Enter some text, then press Enter");
12:   gets(input);
13:   printf("You entered %s", input);
14:   return 0;
15: }
```

Description

In this example, the argument to gets() is the expression input, which is the name of a type char array and therefore a pointer to the first array element. The array was declared with 81 elements in line 8. Because the maximum line length possible on most computer screens is 80 characters, this array size provides space for the longest possible input line (plus the null character that gets() adds at the end).

The gets() function also has a return value, which is not used in this example. gets() returns a pointer to type char with the

address where the input string is stored. Yes, this is the same value that is passed to gets(), but having the value returned to the program in this way enables your program to test for a blank line. Listing 10.6 shows how to use the return value.

Listing 10.6: Using the gets() Return Value to Test for Input of a Blank Line

Code

```
 1: /* LIST1006.c: Day 10 Listing 10.6 */
 2: /* Demonstrates using the gets() return value. */
 3:
 4: #include <stdio.h>
 5:
 6: /* Declare a character array to hold input, and */
 7: /* a pointer. */
 8:
 9: char input[81], *ptr;
10:
11: int main(void) {
12:   /* Display instructions. */
13:
14:   puts("Enter text a line at a time,"
15:        " then press Enter");
16:   puts("Enter a blank line when done.");
17:
18:   /* Loop as long as input is not a blank line. */
19:
20:   while (*(ptr = gets(input)) != '\0')
21:     printf("You entered %s\n", input);
22:
23:   puts("Thank you and good-bye");
24:   return 0;
25: }
```

Description

Now you can see how the gets() return value works. If you enter a blank line (that is, if you simply press Enter) in response to lines 14 – 16, the string (which contains 0 characters) is still stored with a null character at the end. Because the string has a length of 0, the null character is stored in the first position. This is the position pointed to by the return value of gets(), so if you test that position and find a null character, you know that a blank line was entered.

Listing 10.6 performs this test in the while statement in line 20. This statement is a bit complicated, so look carefully at the details in order.

	1. The gets() function accepts input from the keyboard until it reaches a newline character. 2. The input string, minus the newline and with a trailing null character, is stored in the memory location pointed to by input. 3. gets() also returns the address of the string (the same value as input), which is assigned to the pointer ptr. 4. An assignment statement is an expression that evaluates to the value of the variable on the left side of the assignment operator. Therefore, the entire expression ptr = gets(input) evaluates to the value of ptr. By enclosing this expression in parentheses and preceding it with the indirection operator (*), the value stored at the pointed-to address is obtained. This is, of course, the first character of the input string. 5. The value of ptr is compared to the null character ('\0'). 6. If the first character of the input string is not the null character (if a blank line has not been entered), the comparison operator returns true and the while loop executes. If the first character is the null character (if a blank line has been entered), the comparison operator returns false and the while loop terminates.

Pointer Errors

When you use gets() or any other function that stores data using a pointer, be sure that the pointer points to allocated space. It's easy to make a mistake such as the following:

```
char *ptr;
gets(ptr);
```

The pointer ptr has been declared but not initialized. It points somewhere, but you don't know where. The gets() function can't know this, so it simply goes ahead and stores the input string at the address contained in ptr. The string may overwrite something important, such as program code or the operating system. The compiler can't catch these kinds of mistakes, so you, the programmer, must be vigilant.

Topic 3.7.2: Inputting Strings with the scanf() Function

Description

You saw on Day 7, "Basic Input/Output," that the scanf() library function accepts numeric data input from the keyboard. This function also can input strings. Remember that scanf() uses a *format string* that tells it how to read the input. To read a string, include the specifier %s in scanf()'s format string. Like gets(), scanf() is passed a pointer to the string's storage location.

Function of Whitespace

How does scanf() decide where the string begins and ends? The beginning is the first non-whitespace character encountered. The end can be specified in one of two ways. If you use %s in the format string, the string runs up to (but not including) the next whitespace character (space, tab, or newline). If you use %ns (where n is an integer constant that specifies field width), scanf() inputs the next n characters or up to the next whitespace character, whichever comes first.

Reading Multiple Strings

You can read in multiple strings with scanf() by including more than one %s in the format string. For each %s in the format string, scanf() uses the preceding rules to find the requested number of strings in the input. Click the Example link.

Example

Using the Field Width Specifier

What about using the field width specifier? Click the Example link.

Example

Entering Fewer Strings Than Expected

If you enter fewer strings, scanf() continues to "look" for the missing strings, and the program does not continue until they are entered. Click the Example link.

Example

```
scanf("%s%s%s", s1, s2, s3);
```

If in response to this statement, you enter January February March, January is assigned to the pointer location s1, February is assigned to s2, and March to s3.

If you execute the statement

```
scanf("%3s%3s%3s", s1, s2, s3);
```

and in response, enter

```
September
```

`Sep` is assigned to s1, `tem` is assigned to s2, and `ber` is assigned to s3.

If in response to the statement

```
scanf("%s%s%s", s1, s2, s3);
```

you enter

```
January February
```

the program sits and waits for the third string specified in the scanf() format string.

Entering More Strings Than Expected

If you enter more strings than requested, the unmatched strings remain (waiting in the keyboard buffer) pending and are read by any subsequent scanf() or other input statements. Click the Example link.

Example

```
scanf("%s%s", s1, s2);
scanf("%s", s3);
```

If you enter

```
January February March
```

the result is that `January` is assigned to the string s1, `February` is assigned to s2, and `March` to s3.

gets() Versus scanf()

The scanf() function has a return value, an integer value equaling the number of items successfully inputted. The return value is often ignored. When you are reading text only, the gets() function is usually preferable to scanf(). The scanf() function is best used when you are reading in a combination of text and numeric data. This is illustrated by the program in Listing 10.7. Remember from Day 7, "Basic Input/Output," that you must use the address of operator (&) when inputting numeric variables with scanf().

Listing 10.7: Inputting Numeric and Text Data with scanf()

Code

```
 1: /* LIST1007.c: Day 10 Listing 10.7 */
 2: /* Demonstrates using scanf() to input numeric */
 3: /* and text data. */
 4:
 5: #include <stdio.h>
 6:
 7: char lname[81], fname[81];
 8: int count, id_num;
 9:
10: int main(void) {
11:   /* Prompt the user. */
12:
13:   puts("Enter last name, first name, ID number ");
14:   puts("separated by spaces, then press Enter.");
15:
16:   /* Input the three data items. */
17:
18:   count = scanf("%s%s%d", lname, fname, &id_num);
19:
20:   /* Display the data. */
21:
22:   printf("%d items entered: %s %s %d",
23:          count, fname, lname, id_num);
24:   return 0;
25: }
```

Description

Remember that scanf() requires the addresses of variables to be used as arguments. In Listing 10.7, lname and fname are pointers (that is, addresses), so they do not need the preceding address of operator (&). In contrast, id_num is a regular variable name, so it requires the & when passed to scanf() on line 18.

Some programmers feel that data entry with scanf() is prone to errors. They prefer to input all data, numeric and string, using gets(), and then have the program separate the numbers and convert them to numeric variables. Such techniques are beyond

	the scope of this course, but they would make a good programming exercise. For that task, you need the string manipulation functions covered on Day 17, "Manipulating Strings."

Topic 3.8: Day 10 Q&A

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

Question 1
What is the difference between a string and an array of characters?

Answer
A string is defined as a sequence of characters ending with the null character. An array is a sequence of characters. A string, therefore, is a null-terminated array of characters.

If you define an array of type char, the actual storage space allocated for the array is the specified size, not the size –1. You are limited to that size; you cannot store a larger string. Here's an example:

```
char state[10]="Minneapolis";
/* Wrong! String longer than array.*/
char state2[10]="MN";
/* OK, but wastes space because string shorter than array. */
```

If, on the other hand, you define a pointer to type char, these restrictions do not apply. The variable is a storage space only for the pointer. The actual strings are stored elsewhere in memory (but you need not worry about where in memory). There's no length restriction or wasted space. A pointer can point to a string of any length.

Question 2
Why shouldn't I just declare big arrays to hold values instead of using a memory allocation function such as malloc()?

Answer
Although it may seem easier to declare large arrays, it is not an effective use of memory. When writing small programs, such as those in this unit, it may seem trivial

to use a function such as malloc() instead of arrays, but as your programs get bigger, you will want to be able to allocate memory only as needed. When you are done with memory, you can put it back by *freeing* it. When you free memory, some other variable or array in a different part of the program can use the memory. (Day 20, "Odds and Ends," covers freeing allocated memory.)

Question 3
Do all computers support the extended ASCII character set?

Answer
No. Most PCs support the extended ASCII set. Some older PCs don't, but the number of older PCs lacking this support is diminishing. Most programmers use the line and block characters of the extended set.

Question 4
What happens if I put a string into a character array that is bigger than the array?

Answer
This can cause a hard-to-find error. You can do this in C, but anything stored in the memory directly after the character array is overwritten. This could be an area of memory not used, some other data, or some vital system information. Your results are going to depend on what you overwrite. Many times nothing happens...for a while. You don't want to do this.

Topic 3.9: Day 10 Think About Its

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

```
char *str1 = { "String 1" };
char str2[] = { "String 2" };
```

Topic 3.10: Day 10 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 line of code that declares a type char variable named letter and initialize it to the character $.

Answer

```
char letter = '$';
```

*** Exercise 2**
Write a line of code that declares an array of type char and initialize it to the string "Pointers are fun!" Make the array just large enough to hold the string.

Answer

```
char array[18] = "Pointers are fun!";
```

*** Exercise 3**
Write a line of code that allocates storage for the string "Pointers are fun!" as in exercise two, but without using an array.

Answer

```
char *array = "Pointers are fun!";
```

*** Exercise 4**
Write code that allocates space for an 80-character string, and then inputs a string from the keyboard and stores it in the allocated space.

Answer

```
char *ptr;
```

```
ptr = malloc(81);
gets(ptr);
```

*** Exercise 5**

Write a function that copies one array of characters into another. (Hint: Do this just like the programs you did in Day 9, "Pointers.")

Answer

Here is one possible answer.

```
/* EXER1005: Day 10 Exercise 5 */
/* Function COPYCHAR copies one array of characters */
/* into another. */

#include <stdio.h>

char greeting[] = "Hello, world!";
char newarray[15];
void copychar(char x[], char y[]);

int main(void) {

  copychar(greeting, newarray);

  printf("Contents of newarray: %s", newarray);

  return 0;
}

void copychar(char inarray[], char outarray[]) {

  int count;

  for (count = 0; inarray[count] != '\0'; count++)
    outarray[count] = inarray[count];

  return;
}
```

*** Exercise 6**

Write a function that accepts two strings. Count the number of characters in each and

return a pointer to the longer string.

Answer

Here is one possible answer.

```
/* EXER1006.C Day 10 Exercise 6 */
/* Function LONGER accepts 2 strings and returns a */
/* pointer to the longer string */

#include <stdio.h>

char string1[] = "This is the first";
char string2[] = "This is the second";
char * longer(char a[], char b[]);
char * sptr;

int main(void) {

  sptr = longer(string1, string2);

  printf("Longer string is: %s", sptr);

  return 0;
}

/* This function returns a pointer to the longer
 * string. The second string is returned
 * if both strings are equal. */

char * longer(char a[], char b[]) {

  int counta, countb;

  for (counta = 0; a[counta] != '\0'; counta++)
    ;

  for (countb = 0; b[countb] != '\0'; countb++)
    ;

  return(counta > countb ? a : b);
}
```

* Exercise 7

Write a function that accepts two strings. Use the malloc() function to allocate enough memory to hold the two strings after they have been concatenated (linked together). Return a pointer to this new string.

For example, if I pass "Hello " and "World!", the function returns a pointer to "Hello World!". Having the concatenated value be the third string is easiest. (You may be able to use your answers from exercises five and six.)

Answer

Here is one possible answer.

```
/* EXER1007.C Day 10 Exercise 7 */
/* Function LONGER accepts 2 strings, allocates */
/* enough memory to concatenate the strings, and */
/* returns a pointer to the concatenated string. */

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

char string1[] = "This is the first";
char string2[] = "This is the second";
char * longer(char a[], char b[]);
void copychar(char a[], char b[], char c[]);

int main(void)  {

  printf("Concatenated string: %s", longer(string1, string2));
  return 0;
}

char * longer(char a[], char b[])  {

  char *newstr;
  int counta, countb;

  for (counta = 0; a[counta] != '\0'; counta++)
    ;

  for (countb = 0; b[countb] != '\0'; countb++)
    ;

  if ((newstr = (char *) malloc(sizeof(char)*
     (counta + countb + 1)))== NULL) {
    printf("Not enough memory to allocate buffer\n");
    exit(1);
```

```
  }

  copychar(a, b, newstr);

  return(newstr);

}

void copychar(char array1[], char array2[], char outarray[])  {

  int count, count2;

  for (count = 0; array1[count] != '\0'; count++)
       outarray[count] = array1[count];
  for (count2 = 0; array2[count2] != '\0';
       count++, count2++)
    outarray[count] = array2[count2];

  return;
}
```

*** Exercise 8**
BUG BUSTERS: Is anything wrong with the following?

```
char a_string[10] = "This is a string";
```

Answer

a_string is declared as an array of 10 characters; however, it is initialized with a string larger than 10 characters. a_string needs to be bigger.

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

```
char *quote[100]={"Smile, Friday is almost here!"};
```

Answer

If the intent of this line of code is to initialize a string, this is wrong. You should use either char *quote, or char quote[100].

*** Exercise 1**
BUG BUSTERS: Is anything wrong with the following?

```
char *string1;
char *string2 = "Second";

string1 = string2;
```

Answer

No.

*** Exercise 1**
BUG BUSTERS: Is anything wrong with the following:

```
char string1[] = "First";
char string2[] = "Second";
string1 = string2;
```

Answer

Yes. Although you can assign one pointer to another, you cannot assign one array to another. You should change the assignment to a string-copying command such as strcpy().

*** Exercise 1**
Using the ASCII chart, write a program that prints a box on the screen using the double-line characters.

Answer

Here is one possible answer.

```
/* EXER1012.C Day 10 Exercise 12 */
```

```
/* MS-DOS Application */
/* Print an ASCII box on the screen */

#include <stdio.h>

int ulcorner = 201;
int urcorner = 187;
int llcorner = 200;
int lrcorner = 188;
int horizont = 205;
int vertical = 186;
int count;

int main(void)  {

/* Print top line of box. */
  printf("%c", ulcorner);
  for (count = 0; count < 15; count++)
    printf("%c", horizont);
  printf("%c\n", urcorner);

/* Print sides of box. */
  for (count = 0; count < 10; count++)  {
    printf("%c", vertical);
    printf("               ");
    printf("%c\n", vertical);
  }

/* Print bottom line of box. */
  printf("%c", llcorner);
  for (count = 0; count < 15; count++)
    printf("%c", horizont);
  printf("%c\n", lrcorner);

  return 0;
}
```

Topic 3.11: Day 10 Summary

Using Character Variables

This unit covered C's char data type. One use for type char variables is storing individual characters. You saw that characters are actually stored as numbers: the ASCII code has assigned a numerical code to each character. Therefore, you can use type char to store small integer values as well. Both signed and unsigned char types are available.

Using Strings

A string is a sequence of characters terminated by the null character. Strings can be used for text data. C stores strings in arrays of type char. To store a string of length n, you need an array of type char with n+1 elements.

Using the malloc() Function

You can use memory allocation functions such as malloc() to make your programs more dynamic. By using malloc(), you can allocate the right amount of memory for your program. Without such functions, you would have to guess at the amount of memory storage the program needs. Your estimate is usually high, so you allocate more memory than needed.

Unit 4. Day 11 Structures

Many programming tasks are simplified by the C data constructs called *structures*. A structure is a data storage method designed by you, the programmer, to suit your program needs exactly.

Today, you learn

- What simple and complex structures are.
- How to define and declare structures.
- How to access data in structures.
- How to create structures that contain arrays and arrays of structures.
- How to declare pointers in structures and pointers to structures.
- How to pass structures as arguments to functions.
- How to define, declare, and use unions.
- How to use structures to create linked lists.

Topic 4.1: Simple Structures

Structures
A *structure* is a collection of one or more variables grouped under a single name for easy manipulation. The variables in a structure, unlike those in an array, can be of different variable types. A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a *member* of the structure. The next section shows a simple example.

Simple Versus Complex Structures
You should start with simple structures. Note that the C language makes no distinction between simple and complex structures, but it is easier to explain structures in this way.

Topic 4.1.1: Defining and Declaring Structures

Defining Structures
If you are writing a graphics program, your code needs to deal with the coordinates

of points on the screen. Screen coordinates are written as an x value, giving horizontal position, and a y value, giving vertical position. You can define a structure named coord that contains both the x and y values of a screen location as follows:

```
struct coord {
    int x;
    int y;
};
```

Syntax
The struct keyword, which identifies the beginning of a structure definition, must be followed immediately by the structure name, or *tag* (which follows the same rules as other C variable names). Within the braces following the structure name is a list of the structure's member variables. You must give a variable type and name for each member.

Declaring Structures: Two Methods
The previous statements define a structure type named coord that contains two integer variables, x and y. They do not, however, actually create any instances of the structure coord. That is, they do not declare (set aside storage for) any structures. There are two ways to declare structures.

One Method
One is to follow the structure definition with a list of one or more variable names. Click the Example link.

Example

Another Method
The previous method of declaring structures combines the declaration with the definition. The second method is to declare structure variables at a different location in your source code from the definition. Click the Example link.

Example

```
struct coord {
    int x;
    int y;
} first, second;
```

These statements define the structure type coord and declare two structures, named first and second, of type coord. first and second are each *instances* of type coord; first contains two integer members named x and y, as does second.

The following statements also declare two instances of type coord:

```
struct coord {
    int x;
    int y;
};
/* Additional code may go here */
struct coord first, second;
```

Topic 4.1.2: Accessing Structure Members

The Structure Member Operator
Individual structure members can be used like other variables of the same type. Structure members are accessed using the *structure member operator* (.), also called the *dot operator*, between the structure name and the member name. Click the Example link and then the Tip button.

Example

DON'T forget the structure instance name and member operator (.) when using a structure's members.

DON'T confuse a structure's tag with its instances! The tag is to declare the structure's template, or format. The instance is a variable declared using the tag.

DON'T forget the struct keyword when declaring an instance from a previously defined structure.

DO declare structure instances with the same scope rules as other variables. (Day 12, "Variable Scope," will cover this topic fully.)

Thus, to have the structure named first refer to a screen location with coordinates `x=50, y=100`, you could write

```
first.x = 50;
```

```
first.y = 100;
```

To display the screen locations stored in the structure second, you could write

```
printf("%d,%d", second.x, second.y);
```

One Major Advantage of Using Structures
At this point you may be wondering what the advantage is of using structures rather than individual variables. One major advantage is the ability to copy information between structures of the same type with a simple equation statement. Click the Example link.

Example

Other Advantages
Other advantages of structures will become apparent as you learn some advanced capabilities. In general, you will find structures to be useful any time information of different variable types needs to be treated as a group. For example, in a mailing list database, each entry could be a structure, and each piece of information (name, address, city, and so on) could be a structure member.

Continuing with the previous example, the statement

```
first = second;
```

is the equivalent of

```
first.x = second.x;
first.y = second.y;
```

When your program uses complex structures with many members, this notation can be a great time-saver.

Topic 4.2: More Complex Structures

Complex Structures
Now that you have been introduced to simple structures, you can get to the more interesting and complex types of structures. These are structures that contain other structures as members and structures that contain arrays as members.

Topic 4.2.1: Structures that Contain Structures

Defining the Structure

As mentioned earlier, a C structure can contain any of C's data types. For example, a structure can contain other structures. The previous example can be extended to illustrate this. Click the Example link.

Example

Declaring the Structure

The previous statement defines only the type rectangle structure. To declare a structure, you must then include a statement such as

```
struct rectangle mybox;
```

You could have combined the definition and declaration, as you did before for the type coord:

```
struct rectangle {
    struct coord topleft;
    struct coord bottomrt;
} mybox;
```

Assume that your graphics program now needs to deal with rectangles. A *rectangle* can be defined by the coordinates of two diagonally opposite corners. You have already seen how to define a structure that can hold the two coordinates required for a single point. You would need two such structures to define a rectangle. You can define a structure as follows (assuming, of course, that you have already defined the type coord structure):

```
struct rectangle {
    struct coord topleft;
    struct coord bottomrt;
};
```

Accessing the Members

To access the actual data locations (the type int members), you must apply the member operator (.) twice. Click the Example link.

Example

Schematic Representation
This may be getting a bit confusing. You might understand better if you look at Figure 11.1, which shows the relationship between the type rectangle structure, the two type coord structures it contains, and the two type int variables each type coord structure contains. The structures are named as in the previous example.

Thus, the expression

```
mybox.topleft.x
```

refers to the x member of the topleft member of the type rectangle structure named mybox. To define a rectangle with coordinates (0,10),(100,200), you would write

```
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 200;
```

Example Program
It's time to look at an example of using structures that contain other structures. The program in Listing 11.1 takes input from the user for the coordinates of a rectangle, and then calculates and displays the rectangle's area. Note the program's assumptions, given in comments near the start of the code (lines 3–8).

Nesting of Structures
C places no limits on the nesting of structures. While memory allows, you can define structures that contain structures that contain structures that contain structures—well, you get the idea! Of course, there's a limit beyond which nesting becomes unproductive. Rarely are more than three levels of nesting used in any C program.

Listing 11.1: A Demonstration of Structures that Contain Other Structures

```
Code  1: /* LIST1101.c: Day 11 Listing 11.1 */
      2: /* Demonstrates structures that contain other */
      3: /* structures. */
      4:
      5: /* Receives input for corner coordinates of a */
      6: /* rectangle and calculates the area. Assumes */
      7: /* that the y coordinate of the upper-left */
      8: /* corner is greater than the y coordinate of */
```

```
 9: /* the lower-right corner, that the x coordinate */
10: /* of the lower-right corner is greater than the */
11: /* x coordinate of the upper-left corner, and */
12: /* that all coordinates are positive. */
13:
14: #include <stdio.h>
15:
16: int length, width;
17: long area;
18:
19: struct coord{
20:   int x;
21:   int y;
22: };
23:
24: struct rectangle{
25:   struct coord topleft;
26:   struct coord bottomrt;
27: } mybox;
28:
29: int main(void) {
30:     /* Input the coordinates. */
31:
32:     printf("\nEnter top left x coordinate: ");
33:     scanf("%d", &mybox.topleft.x);
34:
35:     printf("\nEnter top left y coordinate: ");
36:     scanf("%d", &mybox.topleft.y);
37:
38:     printf("\nEnter bottom right x coordinate: ");
39:     scanf("%d", &mybox.bottomrt.x);
40:
41:     printf("\nEnter bottom right y coordinate: ");
42:     scanf("%d", &mybox.bottomrt.y);
43:
44:     /* Calculate the length and width. */
45:
46:     width = mybox.bottomrt.x - mybox.topleft.x;
47:     length = mybox.bottomrt.y - mybox.topleft.y;
48:
49:     /* Calculate and display the area. */
50:
51:     area = width * length;
52:     printf("The area is %ld units.", area);
53:     return 0;
54: }
```

Output

```
Enter the top left x coordinate: 1
Enter the top left y coordinate: 1
Enter the bottom right x coordinate: 10
Enter the bottom right y coordinate: 10
The area is 81 units.
```

Description	Lines 19 – 22 define the coord structure with its two members, x and y. Lines 24 – 27 declare and define an instance, called mybox, of the rectangle structure. The two members of the rectangle structure are topleft and bottomrt, which are both structures of type coord. Lines 32 – 42 fill in the values in the mybox structure. At first, it might seem that there are only two values to fill because mybox has only two members. Each of mybox's members, however, has its own members: topleft and bottomrt have two members each, x and y from the coord structure. This gives a total of four members to be filled. After the members are filled with values, the length, width, and area are calculated using the structure and member names (lines 44 – 51). When using the x and y values to perform these calculations, you must include the structure instance name. Because x and y are in a structure within a structure, you must use the instance names of both structures — mybox.bottomrt.x, mybox.bottomrt.y, mybox.topleft.x, and mybox.topleft.y — in the calculations.

Topic 4.2.2: Structures that Contain Arrays

Defining the Structure
You can define a structure that contains one array or more as members. The array can be of any C data type (integer, character, and so on). Click the Example link.

Example

Declaring the Structure
You can then declare a structure named record of type data as follows:

```
struct data record;
```

The statements

```
struct data{
    short int x[4];
    char y[10];
};
```

define a structure of type data that contains a 4-element short integer array member named x, and a 10-element character array member named y.

Schematic Representation

The organization of the data structure is shown in Figure 11.2. Note that in this figure the elements of array x take up twice as much space as the elements of array y. This is because (as you learned on Day 3, "Numeric Variables and Constants") a type short typically requires two bytes of storage, whereas a type char usually requires only one byte.

Accessing the Members

Accessing individual elements of arrays that are structure members is done with a combination of the member operator and array subscripts:

```
record.x[2] = 100;
record.y[1] = 'x';
```

You probably remember that character arrays are most frequently used to store strings. You also should remember (from Day 9, "Pointers") that the name of an array, without brackets, is a pointer to the array. Because this holds true for arrays that are structure members, the expression

```
record.y
```

is a pointer to the first element of array y[ ] in the structure record. You could, therefore, print the contents of y[ ] on the screen with the statement

```
puts(record.y);
```

Example Program

Now look at another example. The program in Listing 11.2 uses a structure that contains a type float variable and two type char arrays.

Listing 11.2: A Demonstration of A Structure that Contains Array Members

```
Code  1: /* LIST1102.c: Day 11 Listing 11.2 */
      2: /* Demonstrates a structure with array members. */
      3:
```

```
 4: #include <stdio.h>
 5:
 6: /* Define and declare a structure to hold the */
 7: /* data. It contains one float variable and two */
 8: /* char arrays. */
 9:
10: struct data{
11:   float amount;
12:   char fname[30];
13:   char lname[30];
14: } rec;
15:
16: int main(void) {
17:   /* Input the data from the keyboard. */
18:
19:   printf("\nEnter the donor's first and last "
20:          "names, separated by a space: ");
21:   scanf("%s %s", rec.fname, rec.lname);
22:
23:   printf("\nEnter the donation amount: ");
24:   scanf("%f", &rec.amount);
25:
26:   /* Display the information. Note: %.2f */
27:   /* specifies a floating point value to be */
28:   /* displayed with two digits to the right */
29:   /* of the decimal point. */
30:
31:   /* Display the data on the screen. */
32:
33:   printf("\nDonor %s %s gave $%.2f.",
34:           rec.fname, rec.lname, rec.amount);
35:   return 0;
36: }
```

Output

```
Enter the donor's first and last names, separated
by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

Description

This program includes a structure that contains array members named fname[30] and lname[30]. Both are arrays of characters that hold a person's first name and last name, respectively. The structure declared in lines 10 – 14 is called data. It contains the fname and lname character arrays with a type float variable called amount. This structure is ideal for holding a person's name (in two parts, first name and last name) and a value, such as the amount the person donated to a charitable organization.

An instance of the array, called rec, has also been declared in line 14. The rest of the program uses rec to get values from the user

	(lines 19 – 24), and then print them (lines 33 – 34).

Topic 4.3: Arrays of Structures

Defining the Structure
If you can have structures that contain arrays, can you also have arrays of structures? You bet you can! In fact, arrays of structures are very powerful programming tools. Here's how it's done.

You've seen how a structure definition can be tailored to fit the data your program needs to work with. Usually a program needs to work with more than one instance of the data. For example, in a program to maintain a list of phone numbers, you can define a structure to hold each person's name and number.

```
struct entry{
    char fname[10];
    char lname[12];
    char phone[8];
};
```

Declaring the Array of Structures
A phone list has to hold many entries, however, so a single instance of the entry structure isn't of much use. What you need is an array of structures of type entry. After the structure has been defined, you can declare an array as follows:

```
struct entry list[1000];
```

This statement declares an array named list that contains 1,000 elements. Every element is a structure of type entry and is identified by subscript like other array element types. Each of these structures has three elements, each of which is an array of type char. This entire complex creation is diagrammed in Figure 11.3.

Assigning the Data in One Array Element to Another
When you have declared the array of structures, you can manipulate the data in many ways. For example, you can assign the data in one array element to another array element. Click the Example link.

Example

The following statement assigns to each member of the structure list[1] the values contained in the corresponding members of list[5].

```
list[1] = list[5];
```

Moving Data Between Individual Structure Members
You also can move data between individual structure members. Click the Example link.

Example

Moving Data Between Individual Elements of the Structure Member Arrays
You also can, if you desire, move data between individual elements of the structure member arrays. Click the Example link.

Example

The statement

```
strcpy(list[1].phone, list[5].phone);
```

copies the string in list[5].phone to list[1].phone. (The strcpy() library function copies one string to another string. You learn the details of this on Day 17, "Manipulating Strings.")

The following statement moves the second character of list[5]'s phone number to the fourth position in list[2]'s phone number. (Don't forget that subscripts start at offset 0.)

```
list[2].phone[3] = list[5].phone[1];
```

Example Program
The program in Listing 11.3 demonstrates the use of arrays of structures. Moreover, it demonstrates arrays of structures that contain arrays as members. You should become familiar with the techniques used in Listing 11.3. Many real-world programming tasks are best accomplished by using arrays of structures that contain arrays as members.

Listing 11.3: Demonstration of Arrays of Structures

Code

```
/* LIST1103.c: Day 11 Listing 11.3 */
/* Demonstrates using arrays of structures. */

#include <stdio.h>

/* Define a structure to hold entries. */

struct entry{
  char fname[20];
  char lname[20];
  char phone[10];
};

/* Declare an array of structures. */

struct entry list[4];

int i;

int main(void) {
    /* Loop to input data for four people. */

  for (i = 0; i < 4; i++) {
      printf("\nEnter first name: ");
      scanf("%s", list[i].fname);
      printf("Enter last name: ");
      scanf("%s", list[i].lname);
      printf("Enter phone in 123-4567 format: ");
      scanf("%s", list[i].phone);
  }

  /* Print two blank lines. */

  printf("\n\n");

  /* Loop to display data. */

  for (i = 0; i < 4; i++) {
  printf("Name: %s %s", list[i].fname,
          list[i].lname);
  printf("\t\tPhone: %s\n", list[i].phone);
  }
  return 0;
}
```

Output

```
Enter first name: Bradley
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212

Enter first name: Peter
```

```
Enter last name: Aitken
Enter phone in 123-4567 format: 555-3434

Enter first name: Melissa
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212

Enter first name: John
Enter last name: Smith
Enter phone in 123-4567 format: 555-1234



Name: Bradley Jones              Phone: 555-1212
Name: Peter Aitken               Phone: 555-3434
Name: Melissa Jones              Phone: 555-1212
Name: John Smith                 Phone: 555-1234
```

Description

Lines 8 – 12 define a template structure called entry that contains three character arrays, fname, lname, and phone.

Line 16 uses the template to define an array of four entry structure variables called list.

Line 18 defines a variable of type int to be used as a counter throughout the program.

main() starts in line 20. The first function of main() is to perform a loop four times with a for statement. This loop is used to get information for the array of structures. This can be seen in lines 23 – 30. Notice that list is being used with the subscript as the array variables on Day 8, "Numeric Arrays," were subscripted.

Line 34 provides a break from the input before starting with the output. It prints two blank lines.

Lines 38 – 42 display the data that the user entered in the preceding step. The values in the array of structures are printed with the subscripted array name followed by the member operator (.) and the structure member name.

Topic 4.4: Initializing Structures

Initializing Structures When Declared

Like other C variable types, structures can be initialized when they are declared. The procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values separated by commas and enclosed in braces.

Structures that Contain Arrays

The following statements show how you can initialize arrays of structures. Click the Example link.

Example

Structures that Contain Structures

For a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Click the Example link.

Example

```
1: struct sale {
2:     char customer[20];
3:     char item[20];
4:     float amount;
5: } mysale = { "Acme Industries",
6:              "Left-handed widget",
7:               1000.00
8:            };
```

Explanation

When these statements are executed, they perform the following actions:

1. Define a structure type named sale (lines 1–5).
2. Declare an instance of structure type sale named mysale (line 5).
3. Initialize the structure member mysale.customer to the string "Acme Industries" (line 5).
4. Initialize the structure member mysale.item to the string "Left-handed widget" (line 6).
5. Initialize the structure member mysale.amount to the value 1000.00 (line 7).

Here's an example that expands slightly on the previous one:

```
1: struct customer {
2:     char firm[20];
3:     char contact[25];
4: }
5:
6: struct sale {
7:     struct customer buyer;
8:     char item[20];
9:     float amount;
10: } mysale = { { "Acme Industries", "George Adams"},
11:                "Left-handed widget",
12:                 1000.00
13:            };
```

Explanation

These statements perform the following initializations:

1. The structure member mysale.buyer.firm is initialized to the string "Acme Industries" (line 10).
2. The structure member mysale.buyer.contact is initialized to the string "George Adams" (line 10).
3. The structure member mysale.item is initialized to the string "Left-handed widget" (line 11).
4. The structure member mysale.amount is initialized to the amount 1000.00 (line 12).

Arrays of Structures

You also can initialize arrays of structures. The initialization data that you supply is applied, in order, to the structures in the array. Click the Example link.

Example

To declare an array of structures of type sale and initialize the first two array elements (that is, the first two structures), you could write

```
1: struct customer {
2:     char firm[20];
3:     char contact[25];
4: };
5:
6: struct sale {
7:     struct customer buyer;
8:     char item[20];
9:     float amount;
10: };
11:
12:
13: struct sale y1990[100] = {
14:      { { "Acme Industries", "George Adams"},
15:          "Left-handed widget",
16:           1000.00
17:      }
18:      { { "Wilson & Co.", "Ed Wilson"},
19:          "Type 12 gizmo",
20:           290.00
21:      }
22: };
```

Explanation

These statements perform the following initializations:

1. The structure member y1990[0].buyer.firm is initialized to the string "Acme Industries" (line 14).
2. The structure member y1990[0].buyer.contact is initialized to the string "George Adams" (line 14).
3. The structure member y1990[0].item is initialized to the string "Left-handed widget" (line 15).
4. The structure member y1990[0].amount is initialized to the amount 1000.00 (line 16).
5. The structure member y1990[1].buyer.firm is initialized to the string "Wilson & Co." (line 18).
6. The structure member y1990[1].buyer.contact is initialized to the string "Ed Wilson" (line 18).
7. The structure member y1990[1].item is initialized to the string "Type 12 gizmo" (line 19).
8. The structure member y1990[1].amount is initialized to the amount

290.00 (line 20).

Topic 4.5: Structures and Pointers

Structures and Pointers
Given that pointers are such an important part of C, you shouldn't be surprised to find that they can be used with structures. You can use pointers as structure members, and you also can declare pointers to structures. These are covered, in turn, in the following paragraphs.

Topic 4.5.1: Pointers as Structure Members

Defining and Declaring the Structure
You have complete flexibility in using pointers as structure members. Pointer members are declared in the same manner as pointers that are not members of structures; that is, by using the indirection operator * (also known as the dereferencing operator). Click the Example link.

Example

Initializing the Pointers
As with all pointers, declaring them is not enough; you must, by assigning them the address of a variable, initialize them to point to something. Click the Example link.

Example

These statements define and declare a structure whose two members are both pointers to type int.

```
struct data {
    int *value;
    int *rate;
} first;
```

If cost and interest have been declared to be type int variables, you could write

```
first.value = &cost;
first.rate = &interest;
```

Now that the pointers have been initialized, you can use the indirection operator (*), as explained on Day 9, "Pointers." The expression `*first.value` evaluates to the value of cost, and the expression `*first.rate` evaluates to the value of interest.

Pointer to Type char

Perhaps the most frequent type of pointer used as a structure member is a pointer to type char. Recall from Day 10, "Characters and Strings," that a string is a sequence of characters delineated by a pointer that points to the string's first character and a null character that indicates the end of the string. To refresh your memory, you can declare a pointer to type char and initialize it to point at a string as shown in the Example. Click the Example link.

Example

You can do the same thing with pointers to type char that are structure members. Click the Example link.

Example

```
char *p_message;
p_message = "C User Guide";
```

```
struct msg {
    char *p1;
    char *p2;
} myptrs;

myptrs.p1 = "C User Guide";
myptrs.p2 = "By DPEC";
```

Schematic Representation

Figure 11.4 illustrates the result of executing the previous statements. Each pointer member of the structure points to the first byte of a string, stored elsewhere in memory. Contrast this with Figure 11.3, which shows how data is stored in a structure that contains arrays of type char.

Using Pointer Structure Members

You can use pointer structure members anywhere a pointer can be used. Click the Example link.

Example

To print the pointed-to strings, you would write

```
printf("%s %s", myptrs.p1, myptrs.p2);
```

Arrays of Type char Versus Pointers to Type char

What is the difference between using an array of type char as a structure member and using a pointer to type char? These are both methods for "storing" a string in a structure, as shown here in the structure msg that uses both methods:

```
struct msg {
    char p1[10];
    char *p2;
} myptrs;
```

Recall that an array name without brackets is a pointer to the first array element. You can, therefore, use these two structure members in similar fashion. Click the Example link.

Example

```
strcpy(myptrs.p1, "C User Guide");
strcpy(myptrs.p2, "By DPEC");
/* additional code goes here */
puts(myptrs.p1);
puts(myptrs.p2);
```

Arrays of Type char

What is the difference between these methods? It is this: If you define a structure that contains an array of type char, every instance of that structure type contains storage space for an array of the specified size. Furthermore, you are limited to the specified size and cannot store a larger string in the structure. Click the Example link.

Example

```
struct msg {
    char p1[10];
    char p2[10];
} myptrs;
...
strcpy(p1, "Minneapolis");
/* Wrong! String longer than array.*/
strcpy(p2, "MN");
/* OK, but wastes space because string
 * shorter than array. */
```

Pointers to Type char

If, on the other hand, you define a structure that contains pointers to type char, these restrictions do not apply. Each instance of the structure contains storage space for only the pointer. The actual strings are stored elsewhere in memory (but you need not worry about *where* in memory). There's no length restriction or wasted space. The actual strings are stored elsewhere, not as part of the structure. Each pointer in the structure can be pointed to a string of any length. That string becomes part of the structure even though it is not stored in the structure.

Topic 4.5.2: Pointers to Structures

Using Pointers to Structures

A C program can declare and use pointers to structures, just as pointers to any other data storage type. Pointers to structures are often used when passing a structure as an argument to a function. Pointers to structures are also used in a very powerful data storage method known as *linked lists*. Both of these topics are covered later in this unit.

Defining the Structure

For now, take a look at how your program can create and use pointers to structures. First, define a structure.

```
struct part {
    int number;
    char name[10];
};
```

Declaring a Pointer to Type part

Now declare a pointer to type part.

```
struct part *p_part;
```

Remember, the indirection operator (*) in the declaration says that p_part is a pointer to type part and not an instance of type part.

Declaring an Instance of Type part
Can the pointer be initialized now? No, because the structure part has been defined but no instances of it have been declared. Remember that it's a declaration, not a definition, that sets aside storage space in memory for a data object. Because a pointer needs a memory address to point to, you must declare an instance of type part before anything can point at it. So here's the declaration:

```
struct part gizmo;
```

Initializing the Pointer
Now you can perform the pointer initialization.

```
p_part = &gizmo;
```

The preceding statement assigns the address of gizmo to p_part. (Recall the address-of operator [&] from Day 9, "Pointers.") The relationship between a structure and a pointer to the structure is shown in Figure 11.5.

Using the Indirection Operator
Now that you have a pointer to the structure gizmo, how do you make use of it? One method uses the indirection operator (*). Recall from Day 9, "Pointers," that if `ptr` is a pointer to a data object, the expression `*ptr` refers to the object pointed to. Applying this to the current example, you know that p_part is a pointer to the structure gizmo and therefore *p_part refers to gizmo. You then apply the structure member operator (.) to access individual members of gizmo. To assign the value 100 to gizmo.number you could write

```
(*p_part).number = 100;
```

The `*p_part` must be enclosed in parentheses because the (.) operator has a higher precedence than the (*) operator.

Using the Indirect Membership Operator
A second method to access structure members using a pointer to the structure is to use the *indirect membership operator,* which consists of the symbols ->, (a hyphen followed by the greater than symbol). (Note that used together in this way, C treats them as a single operator, not two.) The symbol is placed between the pointer name

and the member name. Click the Examples link.

Examples

To access the number member of gizmo with the p_part pointer, you would write

```
p_part->number
```

Looking at another example, if str is a structure, p_str is a pointer to str, and memb is a member of str, you can access str.memb by writing

```
p_str->memb
```

Summary
There are three ways, therefore, to access a structure member: one, using the structure name; two, using a pointer to the structure with the indirection operator (*); and three, using a pointer to the structure with the indirect membership operator (->). If p_str is a pointer to the structure str, the following expressions are all equivalent:

```
str.memb
(*p_str).memb
p_str->memb
```

Topic 4.5.3: Pointers and Arrays of Structures

Defining the Structure
You've seen that arrays of structures can be a very powerful programming tool, as can pointers to structures. You can combine the two, using pointers to access structures that are array elements.

To illustrate, here is a structure definition from an earlier example:

```
struct part {
    int number;
    char name[10];
};
```

Declaring an Array of Type part
After the structure part is defined, you can declare an array of type part.

```
struct part data[100];
```

DON'T confuse arrays with structures!

DO take advantage of declaring a pointer to a structure—especially when using arrays of structures.

DON'T forget that when you increment a pointer, it moves a distance equivalent to the size of the data to which it points. In the case of a pointer to a structure, this is the size of the structure.

DO use the indirect membership operator (->) when working with a pointer to a structure.

Declaring a Pointer to Type part and Initializing It

Next, you can declare a pointer to type part and initialize it to point at the first structure in the array data:

```
struct part *p_part;
p_part = &data[0];
```

Recall that the name of an array without brackets is a pointer to the first array element, so the second line could also have been written

```
p_part = data;
```

Accessing the First Array Element

You now have an array of structures of type part, and a pointer to the first array element (that is, the first structure in the array). You could, for example, print the contents of the first element with the statement

```
printf("%d %s", p_part->number, p_part->name);
```

Accessing Successive Array Elements

What if you want to print all the array elements? You would probably use a for loop, printing one array element with each iteration of the loop. To access the members using pointer notation, you must change the pointer p_part so that with each iteration of the loop it points at the next array element (that is, the next structure in the array). How do you do this?

Using Pointer Arithmetic

C's pointer arithmetic comes to your aid. The unary increment operator (++) has a

special meaning when applied to a pointer: it means "increment the pointer by the size of the object it points to." Put another way, if you have a pointer ptr that points to a data object of type obj, the statement `ptr++;` has the same effect as `ptr += sizeof(obj);`.

This aspect of pointer arithmetic is particularly relevant to arrays as follows: array elements are stored sequentially in memory. If a pointer points to array element n, incrementing the pointer with the (++) operator causes it to point to element n + 1.

Schematic Representation

This is illustrated in Figure 11.6, which shows an array named x[ ] that consists of four-byte elements (for example, a structure containing two type short members, each two bytes long). The pointer ptr was initialized to point at x[0]; each time ptr is incremented, it points at the next array element.

What this means is that your program can step though an array of structures (or an array of any other data type, for that matter) by incrementing a pointer. This sort of notation is usually easier to use and more concise than using array subscripts to perform the same task.

The program in Listing 11.4 demonstrates how you do this.

Listing 11.4: Accessing Successive Array Elements by Incrementing a Pointer

```
Code  1: /* LIST1104.c: Day 11 Listing 11.4 */
      2: /* Demonstrates stepping through an array of  */
      3: /* structures using pointer notation. */
      4:
      5: #include <stdio.h>
      6:
      7: #define MAX 4
      8:
      9: /* Define a structure, then declare and */
     10: /* initialize an array of 4 structures. */
     11:
     12: struct part{
     13:     int number;
     14:     char name[10];
     15: } data[MAX] = {1, "Smith",
     16:                2, "Jones",
     17:                3, "Adams",
     18:                4, "Wilson"
     19:                };
     20:
     21: /* Declare a pointer to type part, and a */
     22: /* counter variable. */
```

```
23:
24: struct part *p_part;
25: int count;
26:
27: int main(void) {
28:   /* Initialize the pointer to the first array */
29:   /* element. */
30:
31:   p_part = data;
32:
33:   /* Loop through the array, incrementing the */
34:   /* pointer with each iteration. */
35:
36:   for (count = 0; count < MAX; count++) {
37:       printf("\nAt address %d: %d %s", p_part,
38:               p_part->number, p_part->name);
39:       p_part++;
40:   }
41:   return 0;
42: }
```

Output

```
At address 96: 1 Smith
At address 108: 2 Jones
At address 120: 3 Adams
At address 132: 4 Wilson
```

Description

In lines 12 – 19, the program declares and initializes an array of structures called data.

In line 24, a pointer called p_part is defined that will be used to point to the data structure.

The main() function's first task is to set the pointer, p_part, to point to the part structure that was declared (line 31). Next, all the elements are then printed using a for loop that increments the pointer to the array with each iteration. The program displays the address of each element along with the member values.

Look closely at the addresses displayed. The precise values may differ on your system, but they are in equal-sized increments—just the size of the structure part (other systems may have an increment of 14). This illustrates clearly that incrementing a pointer increases it by an amount equal to the size of the data object it points to.

Topic 4.5.4: Passing Structures as Arguments to Functions

Passing a Structure

Like other data types, a structure can be passed as an argument to a function. The program in Listing 11.5 shows how to do this. This program is a modification of the program in Listing 11.2, using a function to display data on the screen (whereas Listing 11.2 uses statements that are part of main()).

Listing 11.5: Passing a Structure as a Function Argument

```
Code  1: /* LIST1105.c: Day 11 Listing 11.5 */
      2: /* Demonstrates passing a structure to a */
      3: /* function. */
      4:
      5: #include <stdio.h>
      6:
      7: /* Declare and define a structure to hold the */
      8: /* data. */
      9:
     10: struct data{
     11:     float amount;
     12:     char fname[30];
     13:     char lname[30];
     14: }  rec;
     15:
     16: /* The function prototype. The function has no */
     17: /* return value, and it takes a structure of */
     18: /* type data as its one argument. */
     19:
     20: void print_rec(struct data x);
     21:
     22: int main(void) {
     23:     /* Input the data from the keyboard. */
     24:
     25:     printf("\nEnter the donor's first and last "
     26:            "names, separated by a space: ");
     27:     scanf("%s %s", rec.fname, rec.lname);
     28:
     29:     printf("\nEnter the donation amount: ");
     30:     scanf("%f", &rec.amount);
     31:
     32:     /* Call the display function. */
     33:
     34:     print_rec(rec);
     35:     return 0;
     36: }
     37:
```

```
38: void print_rec(struct data x) {
39:   printf("\nDonor %s %s gave $%.2f.",
40:           x.fname, x.lname, x.amount);
41: }
```

Output

```
Enter the donor's first and last names,
separated by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

Description

Looking at line 20, you see the function prototype for the function that is to receive the structure. As you would with any other data type that was going to be passed, you need to include the proper arguments. In this case, it is a structure of type data. This is repeated in the header for the function in line 38. When calling the function, you need only pass the structure instance name, in this case rec (line 34). That's all there is to it. Passing a structure to a function is not much different from passing a simple variable.

Listing 11.2: A Demonstration of A Structure that Contains Array Members

Code

```
 1: /* LIST1102.c: Day 11 Listing 11.2 */
 2: /* Demonstrates a structure with array members. */
 3:
 4: #include <stdio.h>
 5:
 6: /* Define and declare a structure to hold the */
 7: /* data. It contains one float variable and two */
 8: /* char arrays. */
 9:
10: struct data{
11:   float amount;
12:   char fname[30];
13:   char lname[30];
14: } rec;
15:
16: int main(void) {
17:   /* Input the data from the keyboard. */
18:
19:   printf("\nEnter the donor's first and last "
20:          "names, separated by a space: ");
21:   scanf("%s %s", rec.fname, rec.lname);
22:
23:   printf("\nEnter the donation amount: ");
24:   scanf("%f", &rec.amount);
25:
26:   /* Display the information. Note: %.2f */
```

```
27:   /* specifies a floating point value to be */
28:   /* displayed with two digits to the right */
29:   /* of the decimal point. */
30:
31:   /* Display the data on the screen. */
32:
33:   printf("\nDonor %s %s gave $%.2f.",
34:           rec.fname, rec.lname, rec.amount);
35:   return 0;
36: }
```

Output

```
Enter the donor's first and last names, separated
by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

Description

This program includes a structure that contains array members named fname[30] and lname[30]. Both are arrays of characters that hold a person's first name and last name, respectively. The structure declared in lines 10 – 14 is called data. It contains the fname and lname character arrays with a type float variable called amount. This structure is ideal for holding a person's name (in two parts, first name and last name) and a value, such as the amount the person donated to a charitable organization.

An instance of the array, called rec, has also been declared in line 14. The rest of the program uses rec to get values from the user (lines 19 – 24), and then print them (lines 33 – 34).

Passing the Structure's Address

You also can pass a structure to a function by passing the structure's address (that is, a pointer to the structure). In older versions of C, this was, in fact, the only way to pass a structure as an argument. It's not necessary now, but you may see older programs that still use this method. If you pass a pointer to a structure as an argument, remember that you must use the indirect membership operator (->) to access structure members in the function.

Topic 4.6: Unions

Unions

Unions are similar to structures. A union is declared and used in the same ways that a structure is. A union differs from a structure in that only one of its members can be used at a time. The reason for this is simple. All the members of a union occupy the

same area of memory. They are laid on top of each other.

Topic 4.6.1: Defining, Declaring, and Initializing Unions

Defining and Declaring Unions
Unions are defined and declared in the same fashion as structures. The only difference in the declarations is that the keyword union is used instead of struct. Click the Example link.

Example

Initializing Unions
A union can be initialized on its declaration. Because only one member can be used at a time, only one can be initialized. To avoid confusion, only the first member of the union can be initialized. Click the Example link.

Example

To define a simple union of a char variable and an integer variable, you would do the following:

```
union shared {
char c;
int i;
};
```

This union, shared, can be used to create instances of a union that can hold either a character value c or an integer value i. This is an OR condition. Unlike a structure that would hold both values, the union can only hold one value at a time.

The following shows an instance of the shared union being declared and initialized:

```
union shared generic_variable = {'@'};
```

Notice that the generic_variable union was initialized just as the first member of a structure would be initialized.

Topic 4.6.2: Accessing Union Members

Using the Member Operator

Individual union members can be used in the same way that structure members can be used—by using the member operator (.). There is an important difference in accessing union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it is important to access only one member at a time. This is best shown in an example.

Example Programs

Listing 11.6 is an example of the wrong use of unions.

Listing 11.7 shows a more practical use of a union. This listing shows a simplistic use of a union. Although this use is simplistic, it is one of the more common uses of a union. Click both Listing buttons and then click the Tip button on the toolbar.

DON'T try to initialize more than the first union member.

DO remember which union member is being used. If you fill in a member of one type and try to use a different type, you can get unpredictable results.

DON'T forget that the size of a union is equal to its largest member.

DO note that unions are an advanced C topic.

Listing 11.6: An Example of the Wrong Use of Unions

```
Code  1: /* LIST1106.c: Day 11 Listing 11.6 */
      2: /* Example of using more than one union member */
      3: /* at a time */
      4:
      5: #include <stdio.h>
      6:
      7: int main(void) {
      8:     union shared_tag{
      9:     char  c;
     10:     int    i;
     11:     long  l;
     12:     float  f;
     13:     double  d;
     14:   } shared;
```

```
15:
16:   shared.c = '$';
17:
18:   printf("\nchar c  = %c",  shared.c);
19:   printf("\nint i    = %d",  shared.i);
20:   printf("\nlong l  = %ld",  shared.l);
21:   printf("\nfloat f  = %f",  shared.f);
22:   printf("\ndouble d  = %f",  shared.d);
23:
24:   shared.d = 123456789.8765;
25:
26:   printf("\n\nchar c  = %c",  shared.c);
27:   printf("\nint i    = %d",  shared.i);
28:   printf("\nlong l  = %ld",  shared.l);
29:   printf("\nfloat f  = %f",  shared.f);
30:   printf("\ndouble d  = %f",  shared.d);
31:
32:   return 0;
33: }
```

Output

```
char c   = $
int i    = 4900
long l   = 437785380
float f  = 0.000000
double d = 0.000000

char c   = 7
int i    = -30409
long l   = 1468107063
float f  = 284852666499072.000000
double d = 123456789.876500
```

Description

In this listing, you can see that a union named shared was defined and declared in lines 8 – 14. shared contains 5 members, each of a different type. Lines 16 and 24 initialize individual members of shared. Lines 18 through 22 and 26 through 30 then present the values of each member using printf() statements.

Note that with the exceptions of `char c = $` and `double d = 123456789.876500`, the output may not be the same on your computer. Because the character variable, c, was initialized in line 16, it is the only value that should be used until a different member is initialized. The results from printing the other union member variables, (i, l, f, and d) can be unpredictable (lines 18 – 22).

Similarly, line 24 puts a value into the double variable, d. Again, notice that the printing of the variables again is

	unpredictable for all but d. The value entered into c in line 16 has been lost because it was overwritten when the value of d in line 24 was entered. This is evidence that the members all occupy the same space.

Listing 11.7: A Practical Use of a Union

```
Code  1: /* LIST1107.c: Day 11 Listing 11.7 */
      2: /* Example of a typical use of a union */
      3:
      4: #include <stdio.h>
      5:
      6: #define CHARACTER  'C'
      7: #define INTEGER    'I'
      8: #define FLOAT    'F'
      9:
     10: struct generic_tag{
     11:     char type;
     12:     union shared_tag {
     13:       char  c;
     14:       int    i;
     15:       float  f;
     16:   } shared;
     17: };
     18:
     19: void print_function(struct generic_tag generic);
     20:
     21: int main(void) {
     22:   struct generic_tag var;
     23:
     24:   var.type = CHARACTER;
     25:   var.shared.c = '$';
     26:   print_function(var);
     27:
     28:   var.type = FLOAT;
     29:   var.shared.f = 12345.67890;
     30:   print_function(var);
     31:
     32:   var.type = 'x';
     33:   var.shared.i = 111;
     34:   print_function(var);
     35:   return 0;
     36: }
     37:
     38: void print_function(struct generic_tag generic) {
     39:   printf("\n\nThe generic value is...");
     40:   switch(generic.type) {
     41:     case CHARACTER: printf("%c", generic.shared.c);
     42:                     break;
```

```
43:     case INTEGER:   printf("%d", generic.shared.i);
44:                     break;
45:     case FLOAT:     printf("%f", generic.shared.f);
46:                     break;
47:     default:        printf("an unknown type: %c",
48:                             generic.type);
49:                     break;
50:   }
51: }
```

Output

```
The generic value is...$

The generic value is...12345.678711

The generic value is...an unknown type: x
```

Description

This program is a very simplistic version of what could be done with a union. This program provides a way of storing multiple data types in a single storage space. The generic_tag structure allows you to store either a character or an integer or a floating point number within the same area. This area is a union called shared that operates just like the examples in Listing 11.6. Notice that the generic_tag structure also adds an additional field called type. This field is used to store information on the type of variable contained in shared. type helps prevent shared from being used in the wrong way—thus helping to avoid erroneous data such as that presented in Listing 11.6.

A formal look at the program shows that lines 6, 7, and 8 define constants CHARACTER, INTEGER, and FLOAT. These are used later in the program to make the listing more readable. Lines 10 – 17 define a generic_tag structure that will be used later. Line 19 presents a prototype for the print_function(). The structure var is declared in line 22 and is first initialized to hold a character value in lines 24 and 25. A call to print_function() in 26 lets the value be printed. Lines 28 to 30 and 32 to 40 repeat this process with other values.

The print_function() is the heart of this listing. Although this function is used to print the value from a generic_tag variable, a similar function could have been used to initialize it. print_function() will evaluate the type variable in order to print a statement with the appropriate variable type. This prevents getting erroneous data such as that in Listing 11.6.

Topic 4.7: Linked Lists

Linked Lists

The last topic in this unit is a brief introduction to linked lists. The term *linked list* refers to a general class of data storage methods in which each item of information is linked to one or more other items using pointers.

Kinds of Linked Lists

There are several kinds of linked lists, including singly linked lists, doubly linked lists, and binary trees. Each type is suited for certain data storage tasks. What they have in common is that the links between data items are defined by information in the data items themselves. This is distinct from arrays, in which the links between data items result from the structure of the array. This unit explains the simplest kind of linked list, the singly linked list (which is referred to simply as a linked list).

Topic 4.7.1: The Organization of a Linked List

Constructing a Linked List

To illustrate how linked lists are constructed, start with a simple structure definition:

```
struct data {
    char name[10];
    struct data *next;
};
```

Do you notice anything unusual about the definition? The second member of the type data is a pointer to type data. In other words, the structure includes a pointer to its own type. This means that one instance of type data can point to another instance of the same type. This is how you create the links in a linked list; each structure points to the next structure in the list. This is illustrated in Figure 11.7.

Marking the Beginning and End of the List

Every list must have a beginning and an end. How are these represented in C? The beginning of the list is marked by the *head pointer*, which points to the first structure in the list. The head pointer is not a structure, but simply a pointer to the data type that makes up the list. The end of the list is marked by a structure that has a pointer value of NULL. Because every other structure in the list contains a non-NULL pointer that points to the next list item, a pointer value of NULL is an unmistakable way to indicate the end of the list. The structure of a linked list with its head pointer

and last element is shown in Figure 11.8.

Advantage Over an Array of Structures
The linked list may seem to have no real advantage over an array of structures. There are, however, several significant advantages. One has to do with inserting and deleting elements in a sorted list (a common programming task). Click the Example link.

Example

Another Advantage
Another advantage of linked lists relates to storage space. To use an array, its size (number of elements) must be set when the program is compiled. If, during execution, your program runs out of memory space, there's nothing that can be done. With a linked list, however, the program can allocate storage space for structures as needed, up to the limit imposed by your hardware. This is done with C's malloc() function, discussed next.

Imagine that your program maintains a list of 1,000 data items sorted in alphabetical order. To maintain the sorted order, new items must be inserted in the proper location in the list: Baker between Adams and Clark, for example.

Using an Array
If you're using an array, you must make an empty slot in the proper location before you can insert the new data. Say, for example, that Adams is at position 5 and Clark is at position 6. To insert Baker in the proper location (location 6), Clark and all higher array elements must be moved up one slot. This requires a lot of computer processing! Likewise, to delete an item from the middle of the array, all higher items need to be moved down to fill the vacated slot.

Using a Linked List
Linked lists make inserting and deleting data items much easier. All that is required is some simple pointer manipulation. No data actually needs to be moved. Continuing with the preceding example, you need to insert Baker between Adams and Clark. In a linked list, initially the Adams entry points to the Clark entry. To insert the new item, Baker, all you need to do is make the Adams entry point at the Baker entry and the Baker entry point at the Clark entry.

Schematic Representation
This procedure is illustrated in Figure 11.9. (The procedure for deleting an item is just as simple, but figuring out how to do this is left as an exercise for you!)

Topic 4.7.2: Using the malloc() Function

Allocating Space for a Structure

You learned about malloc(), C's memory allocation function, on Day 10, "Characters and Strings." Using malloc() to allocate space for a structure type is essentially identical to using it to allocate space for type char. Click the Example link.

Example

If you have defined a structure type named data, you first declare a pointer to the type

```
struct data *ptr;
```

and then call malloc()

```
ptr = malloc(sizeof(struct data));
```

If malloc() is successful, it allocates a properly sized block of memory and returns a pointer to it. If malloc() fails—if not enough memory is available—it returns NULL (which a real-world program should test for).

Accessing the Allocated Memory

The value returned by malloc() has been assigned to ptr, a pointer to type data. By using this pointer and the indirect membership operator (->), you can access the allocated memory according to the members of data. Thus, if the type data has a type float member named value, you could write

```
ptr->value = 1.205;
```

Structures allocated with malloc() have no name *per se,* so you can never use the membership operator (.) to access their members. You must always use a pointer and the indirect membership operator (->).

Topic 4.7.3: Implementing a Linked List

Implementing a Linked List

The information given in this unit should be sufficient for you to implement a linked-list program. This would be an excellent programming exercise (see Day 16, exercise 10). If you succeed in writing a program that uses a linked list to store data, you can pat yourself on the back; you are well on the way to becoming a proficient C programmer!

Topic 4.8: typedef and Structures

Creating a Synonym for a Structure or Union Type

You can use the typedef keyword to create a synonym for a structure or union type. Click the Example link.

Example

The statements

```
typedef struct {
    int x;
    int y;
} coord;
```

define coord as a synonym for the indicated structure. You can then declare instances of this structure using the coord identifier.

```
coord topleft, bottomright;
```

typedef Versus a Structure Tag

Note that a typedef is different from a structure tag, as described earlier in this unit. Click the Example link.

Example

Whether you use typedef or a structure tag to declare structures makes little practical difference. Using typedef results in slightly more concise code because the struct keyword need not be used. On the other hand, using a tag and having the struct keyword explicit makes it clear that it is a structure being declared.

If you write

```
struct coord {
    int x;
    int y;
};
```

the identifier coord is a tag for the structure. You can use the tag to declare instances of the structure, but unlike with a typedef, you must include the struct keyword:

```
struct coord topleft, bottomright;
```

Topic 4.9: Day 11 Q&A

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

Question 1
Is there any purpose to declaring a structure without an instance?

Answer
Two ways were shown to declare a structure. The first was to declare a structure body, tag, and instance all at once. The second was to declare a structure body and tag without an instance. An instance can then be declared later by using the struct keyword, the tag, and a name for the instance. It is a common programming practice to use the second method. Many programmers will declare the structure body and tag without any instances. The instances will then be declared later in the program. In the next unit variable scope is described. Scope will apply to the instance, but not to the tag or structure body.

Question 2
Is it more common to use a typedef or a structure tag?
Answer
Many programmers use typedefs to make their code easier to read; it makes little practical difference, however. Many add-in libraries that contain functions can be purchased. These add-in products usually have a lot of typedefs to make the product unique. This is especially true of database add-in products.

Question 3
Can I simply assign one structure to another with the assignment operator?

Answer
Yes and No! Newer versions of C compilers will let you assign one structure to

another; however, older versions may not. In older versions of C, you may need to assign each member of the structures individually! This is true of unions also.

Question 4
How big is a union?

Answer
Because each of the members in a union are stored in the same memory location, the amount of room required to store the union is equal to that of its largest member.

Topic 4.10: Day 11 Think About Its

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

```
struct address {
    char name[31];
    char add1[31];
    char add2[31];
    char city[11];
    char state[3];
    char zip[11];
} myaddress =
    { "Bradley Jones",
      "RTSoftware",
      "P.O. Box 1213",
      "Carmel", "IN",
      "46032-1213"};
```

Topic 4.11: Day 11 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 that defines a structure named time that contains three int members.

Answer

```
struct time {
    int hours;
    int minutes;
    int seconds;
} ;
```

*** Exercise 2**

Write the code that: a) defines a structure named data that contains one type int member and two type float members and b) declares an instance of type data named info.

Answer

```
struct data {
    int value1;
    float value2;
    float value3;

} info ;
```

*** Exercise 3**

Continuing from question two, how would you assign the value 100 to the integer member of the info instance of the data structure?

Answer

```
info.value1 = 100;
```

*** Exercise 4**

Write the code that declares and initializes a pointer to info.

Answer

```
struct data *ptr;
ptr = &info;
```

*** Exercise 5**

Show two ways to use pointer notation to assign the value 5.5 to the first float member of info.

Answer

```
ptr->value2 = 5.5;
(*ptr).value2 = 5.5;
```

*** Exercise 6**

Write the definition for a structure type named data that can hold a single string of up to 20 characters and that can be used in a linked list.

Answer

```
struct data {
char name[21];
struct data *ptr;
};
```

*** Exercise 7**

Create a structure containing five strings, address1, address2, city, state, and zip. Create a typedef called RECORD that can be used to create instances of this structure.

Answer

```
typedef struct {
    char address1[31];
    char address2[31];
    char city[11];
    char state[3];
    char zip[11];
```

```
} RECORD;
```

*** Exercise 8**
Using the typedef from exercise seven, allocate and initialize an element called myaddress.

Answer

The following uses the values from quiz question five for the initialization:

```
RECORD myaddress = {"RTSoftware",
                    "P.O. Box 1213",
                    "Carmel", "IN", "46032-1213" };
```

*** Exercise**
BUG BUSTERS: What is wrong with the following code fragment?

```
struct {
    char zodiac_sign[21];
    int month;
} sign = "Leo", 8;
```

Answer

This code fragment has two problems. The first is that the structure should contain a tag. The second problem is the way that sign is initialized. The initialization values should be in braces. The corrected code is

```
struct zodiac {
    char zodiac_sign[21];
    int month;
} sign = {"Leo", 8};
```

*** Exercise 1**
BUG BUSTERS: What is wrong with the following code fragment?

```
/* setting up a union */
```

```
union data{
    char a_word[4];
    long a_number;
}generic_variable = { "WOW", 1000 };
```

Answer

The union declaration has only one problem. Only one variable in a union can be used at a time. This is true of initializing the union also. Only the first member of the union can be initialized. The correct initialization is

```
/* setting up a union */
union data{
    char a_word[4];
    long a_number;
}generic_variable = { "WOW" };
```

*** Exercise 1**

BUG BUSTERS: What is wrong with the following code fragment?

```
/* a structure to be used in a linked list */
struct data{
    char firstname[10];
    char lastname[10];
    char middlename[10];
    char *next_name;
}
```

Answer

If the structure is intended to be used in a linked list, it should contain a pointer to the next name of the structure type. The following is the corrected code.

```
/* a structure to be used in a linked list */
struct data{
    char firstname[10];
    char lastname[10];
    char middlename[10];
    struct data *next_name;
};
```

Topic 4.12: Day 11 Summary

Simple and Complex Structures
This unit showed you how to use structures, a data type that you design to meet the needs of your program. A structure can contain any of C's data types, including other structures, pointers, and arrays. Each data item within a structure, called a member, is accessed using the structure member operator (.) between the structure name and the member name. Structures can be used individually, and they also can be used in arrays.

Structures and Pointers
Pointers to structures open up further possibilities. You can use pointers to create linked lists of structures, in which each list element is linked to the next by means of a pointer. By combining a linked list with the malloc() functions, a C program can dynamically allocate storage as it is needed.

Unions
Unions were presented as being similar to structures. The main difference between a union and a structure is that the union stores all of its members in the same area. This means that only an individual member of a union can be used at a time.

Unit 5. Day 12 Variable Scope 2

On Day 5, "Functions: The Basics," you saw that a variable defined within a function is different from a variable defined outside a function. Without knowing it, you were being introduced to the concept of *variable scope,* an important aspect of C programming.

Today, you learn

- About scope and why it's important.
- What external variables are and why you should usually avoid them.
- About local variables.
- The difference between static and automatic variables.
- About local variables and blocks.
- How to select a storage class.

Topic 5.1: What Is Scope?

Scope
The *scope* of a variable refers to the extent to which different parts of a program have access to the variable, or where the variable is *visible*. When referring to C variables, the terms *accessibility* and *visibility* are used interchangeably. When speaking about scope, the term *variable* refers to all C data types: simple variables, arrays, structures, pointers, and so forth. It refers, too, to symbolic constants defined with the const keyword.

Lifetime
Scope also affects a variable's lifetime: how long the variable persists in memory, or when the variable's storage is allocated and deallocated. First, this unit examines visibility.

Topic 5.1.1: A Demonstration of Scope

Example Programs

Look at the program in Listing 12.1. The program in Listing 12.1 compiles and runs with no problems. Now make a minor modification in the program, moving the definition of the variable x to a location within the main() function. The new source code is shown in Listing 12.2.

Accessibility of Variables

The only difference between Listing 12.1 and Listing 12.2 is where variable x is defined. By moving the definition of x, you change its scope. In Listing 12.1, x is an *external* variable, and its scope is the entire program. It is accessible within both the main() function and the print_value() function. In Listing 12.2, x is a *local* variable, and its scope is limited to within the main() function. As far as print_value() is concerned, x doesn't exist. Later in this unit, you learn more about local and external variables, but first, you need some understanding of the importance of scope.

Listing 12.1: The Variable x Is Accessible Within the Function print_value

Code

```
 1: /* LIST1201.c: Day 12 Listing 12.1 */
 2: /* Illustrates variable scope. */
 3:
 4: #include <stdio.h>
 5:
 6: int x = 999;
 7:
 8: void print_value(void);
 9:
10: int main(void) {
11:   printf("%d\n", x);
12:   print_value();
13:   return 0;
14: }
15:
16: void print_value(void) {
17:   printf("%d\n", x);
18: }
```

Output

```
999
999
```

Description

Listing 12.1 defines the variable x in line 6, uses printf() to display the value of x in line 11, and then calls the function print_value() to display the value of x again. Note that the function print_value() is not passed the value of x as an argument; it simply uses x as an argument to printf() in line 17.

Listing 12.2: The Variable x is Not Accessible Within the Function print_value

Code

```
 1: /* LIST1202.c: Day 12 Listing 12.2 */
 2: /* Illustrates variable scope. */
 3:
 4: #include <stdio.h>
 5:
 6: void print_value(void);
 7:
 8: int main(void) {
 9:   int x = 999;
10:
11:   printf("%d\n", x);
12:   print_value();
13:   return 0;
14: }
15:
16: void print_value(void) {
17:   printf("%d\n", x);
18: }
```

Description

If you try to compile Listing 12.2, the compiler generates an error message similar to the following:

```
list1202.c(17) : Error: undefined identifier 'x'.
```

Remember that in an error message, the number in parentheses refers to the program line where the error was found. Line 17 is the call to printf() within the print_value() function.

This error message tells you that within the print_value() function, the variable x is "undefined" or, in other words, not visible. Note, however, that the call to printf() in line 11 does not generate an error message; in this part of the program, the variable x is visible.

Topic 5.1.2: Why Is Scope Important?

Controlling the Degree of Data Isolation

To understand the importance of variable scope, you need to recall the discussion of structured programming on Day 5, "Functions: The Basics." The structured approach, you may remember, divides the program into independent functions that perform a specific task. The key word here is *independent*. For true independence, it's

necessary that each function's variables be isolated from interference caused by other functions. Only by isolating each function's data can you make sure the function goes about its job without some other part of the program throwing a monkey wrench into the works.

Data Isolation Is Not Always Desirable

You may be thinking that complete data isolation between functions is not always desirable, and you are correct. You will soon realize that by specifying the scope of variables, a programmer has a great deal of control over the degree of data isolation.

Topic 5.2: External Variables

External Variables

An *external* variable is one defined outside of any function. This means outside of main() as well, because main() is a function, too. Until now, most of the variable definitions in this course have been external, placed in the source code before the start of main(). External variables are sometimes referred to as *global* variables. If you do not explicitly initialize an external variable when it is defined, the compiler initializes it to 0.

Topic 5.2.1: External Variable Scope

The Entire Program

The scope of an external variable is the entire program. This means that an external variable is visible throughout main() and every other function in the program. For example, the variable x in Listing 12.1 is an external variable. As you saw when you compiled and ran the program, x is visible within both functions, main() and print_value().

The Entire Source Code File

Strictly speaking, however, it's not accurate to say the scope of an external variable is the entire program. The scope is, rather, the entire source code file that contains the variable definition. If the entire program is contained in one source code file, the two scope definitions are equivalent. Most small-to-moderate size C programs are contained in one file, and that's certainly true of the programs you're writing now.

Source Code in Two or More Files

It's possible, however, for a program's source code to be contained in two or more separate files. You learn how and why this is done on Day 21, "Taking Advantage of

Preprocessor Directives and More," and what special handling is required for external variables in these situations.

Topic 5.2.2: When to Use External Variables

Principle of Modular Independence
Although the sample programs to this point have used external variables, in actual practice, you should use them rarely. Why? Because when you use external variables, you are violating the principle of *modular independence* that is central to structured programming. Modular independence is the idea that each function, or module, in a program contains all the code and data it needs to do its job. With the relatively small programs you are writing now, this may not seem important, but as you progress to larger and more complex programs, over reliance on external variables can start to cause problems.

Good Candidates for External Status
When should you use external variables? Make a variable external only when all or most of the program's functions need access to the variable. Symbolic constants defined with the const keyword are often good candidates for external status. If only some of your functions need access to a variable, pass it to the functions as an argument rather than making it external. Click the Tip button on the toolbar.

DO use local variables for items such as loop counters.

DON'T use external variables if they are not needed by a majority of the program's functions.

DO use local variables to isolate the values the variables contain from the rest of the program.

Topic 5.2.3: The extern Keyword

Syntax
When a function uses an external variable, it is good programming practice to re-declare the variable within the function using the extern keyword. The declaration takes the form

```
extern type name;
```

in which type is the variable type and name is the variable name.

For example, you would add the declaration of x to the functions main() and print_value() in Listing 12.1. The resulting program is shown in Listing 12.3.

Listing 12.1: The Variable x Is Accessible Within the Function print_value

Code

```
 1: /* LIST1201.c: Day 12 Listing 12.1 */
 2: /* Illustrates variable scope. */
 3:
 4: #include <stdio.h>
 5:
 6: int x = 999;
 7:
 8: void print_value(void);
 9:
10: int main(void) {
11:   printf("%d\n", x);
12:   print_value();
13:   return 0;
14: }
15:
16: void print_value(void) {
17:   printf("%d\n", x);
18: }
```

Output

```
999
999
```

Description

Listing 12.1 defines the variable x in line 6, uses printf() to display the value of x in line 11, and then calls the function print_value() to display the value of x again. Note that the function print_value() is not passed the value of x as an argument; it simply uses x as an argument to printf() in line 17.

Listing 12.3: The External Variable x Is Declared as Extern Within the Functions main() and print_value

Code

```
 1: /* LIST1203.c: Day 12 Listing 12.3 */
 2: /* Illustrates variable scope. */
 3:
 4: #include <stdio.h>
 5:
 6: int x = 999;
 7:
```

```
 8: void print_value(void);
 9:
10: int main(void) {
11:   extern int x;
12:
13:   printf("%d", x);
14:   print_value();
15:   return 0;
16: }
17:
18: void print_value(void) {
19:   extern int x;
20:   printf("%d", x);
21: }
```

Output	`999999`
Description	This program prints the value of x twice, first in line 13 as a part of main(), and then in line 21 as a part of print_value(). Line 6 defines x as a type int variable equal to 999. Lines 11 and 19 declare x as an extern int. Note the distinction between a variable definition, which sets aside storage for the variable, and an extern declaration. The extern keyword basically says: "This function uses an external variable with such and such a name and type that is defined elsewhere."

Topic 5.3: Using Local Variables

Local to the Function

A *local* variable is one that is defined within a function. The scope of a local variable is limited to the function in which it is defined. Day 5, "Functions: The Basics," describes local variables within functions, how to define them, and what their advantages are. Local variables are not automatically initialized to 0 by the compiler. If you do not initialize a local variable when it is defined, it has an undefined or *garbage* value. You must explicitly assign a value to local variables before they're used for the first time.

Local to main()

A variable can be local to the main() function as well. This is the case for x in Listing 12.2. It is defined within main(), and as compiling and executing that program illustrates, it's also visible only within main().

Topic 5.3.1: Static Versus Automatic Variables

Automatic
Local variables are *automatic* by default. This means that local variables are created anew each time the function is called, and they are destroyed when execution leaves the function. What this means, in practical terms, is that an automatic variable does not retain its value between calls to the function in which it is defined. Click the Example link.

Example

Static
What if the function needs to retain the value of a local variable between calls? For example, a printing function may need to remember the number of lines already sent to the printer to determine when a new page is needed. For a local variable to retain its value between calls, it must be defined as *static* with the static keyword. Click the Example link.

Example

Suppose your program has a function that uses a local variable x. Also suppose that the first time it is called, the function assigns the value 100 to x. Execution returns to the calling program, and the function is called again later. Does the variable x still hold the value 100? No, it does not. The first instance of variable x was destroyed when execution left the function after the first call. When the function was called again, a new instance of x was created. The old x is gone forever.

```
void func1(int x) {
    static int a;
...
}
```

Example Program
The program in Listing 12.4 illustrates the difference between automatic and static local variables.

If you do some experimenting with automatic variables, you may get results that disagree with what you've examined here. Click the Example link for experiments.

Example

Listing 12.4. The Difference Between Automatic and Static Local Variables

Code

```
/* LIST1204.c: Day 12 Listing 12.4 */
/* Demonstrates automatic and static local */
/* variables. */

#include <stdio.h>

void func1(void);

int main(void) {
  int count;

  for (count = 0; count < 20; count++) {
     printf("At iteration %d: ", count);
     func1();
  }
  return 0;
}

void func1(void) {
  static int x = 0;
  int y = 0;

  printf("x = %d, y = %d\n", x++, y++);
}
```

Output

```
At iteration 0: x = 0, y = 0
At iteration 1: x = 1, y = 0
At iteration 2: x = 2, y = 0
At iteration 3: x = 3, y = 0
At iteration 4: x = 4, y = 0
At iteration 5: x = 5, y = 0
At iteration 6: x = 6, y = 0
At iteration 7: x = 7, y = 0
At iteration 8: x = 8, y = 0
At iteration 9: x = 9, y = 0
At iteration 10: x = 10, y = 0
At iteration 11: x = 11, y = 0
At iteration 12: x = 12, y = 0
At iteration 13: x = 13, y = 0
At iteration 14: x = 14, y = 0
At iteration 15: x = 15, y = 0
At iteration 16: x = 16, y = 0
At iteration 17: x = 17, y = 0
```

	`At iteration 18: x = 18, y = 0` `At iteration 19: x = 19, y = 0`
Description	This program has a function that defines and initializes one variable of each type: automatic and static. This function is func1() in lines 19 – 24. Each time the function is called, both variables x and y are displayed on the screen and incremented (line 23). The main() function in lines 9 – 17 contains a for loop (lines 12 – 15) that prints a message (line 13) and then calls func1() (line 14). The for loop iterates 20 times. In the output listing, note that x, the static variable, increases with each iteration because it retains its value between calls. The automatic variable y, on the other hand, is reinitialized to 0 with each call to func1(). This program also illustrates a difference in the way explicit variable initialization is handled (that is, when a variable is initialized at the time of definition). A static variable is initialized only the first time the function is called. At later calls, the program "remembers" that the variable has already been initialized and does not reinitialize. Instead, the variable retains the value it had when execution last exited the function. In contrast, an automatic variable is initialized to the specified value every time the function is called.

Sample Modification

If you modify the program in Listing 12.4 so that the two local variables aren't initialized when they're defined, the function func1() in lines 19 to 24 reads

```
19: void func1(void) {
20:     static int x;
21:     int y;
22:
23:     printf("x = %d, y = %d\n", x++, y++);
24: }
```

The Automatic Variable Keeps Its Value By Chance

When you run the modified program, you may find that the value of y increases by one with each iteration. This means that y is keeping its value between calls to the

function. Is what you've seen here about automatic variables losing their value a bunch of malarkey?

No, what you see is true. (Have faith!) If you get the results described previously, in which an automatic variable keeps its value on repeated calls to the function, it's only by chance.

Explanation
Here's what happens. Each time the function is called, a new y is created. The compiler may use the same memory location for the new y that was used for y the preceding time the function was called. If y is not explicitly initialized by the function, the storage location may contain the value that y had during the preceding call. The variable seems to have kept its old value, but it's just a chance occurrence; you definitely cannot count on it happening every time!

The auto Keyword
Because automatic is the default for local variables, it need not be specified in the variable definition. However, if you want, you can include the auto keyword in the definition before the type keyword, as shown here:

```
void func1(int y) {
    auto int count;
    /* Additional code goes here */
}
```

Topic 5.3.2: The Scope of Function Parameters

Local Scope
A variable that is contained in a function heading's parameter list has *local* scope. Click the Example link.

Example

Because parameter variables always start with the value passed as the corresponding argument, it is meaningless to think of them as being either static or automatic.

Look at the following function:

```
void func1(int x) {
    int y;
```

```
    /* Additional code goes here */
}
```

Both x and y are local variables with a scope that is the entire function func1(). Of course, x initially contains whatever value was passed to the function by the calling program. When you've made use of that value, you can use x like any other local variable.

Topic 5.3.3: External Static Variables

Making an External Variable Static
An external variable can be made static by including the static keyword in its definition. Click the Example link.

Example

Ordinary External Variables Versus Static External Variables
The difference between an ordinary external variable and a static external variable is one of scope. An ordinary external variable is visible to all functions in the file and can be used by functions in other files. A static external variable is visible only to functions in its own file and below the point of definition.

The previous distinctions obviously apply mostly to programs with source code that is contained in two or more files. This topic is covered on Day 21, "Taking Advantage of Preprocessor Directives and More."

```
static float rate;

main() {
/* Additional code goes here */
}.
```

Topic 5.3.4: Register Variables

The register Keyword
The register keyword is used to suggest to the compiler that an automatic local variable be stored in a processor register rather than in regular memory. What is a

processor register and the advantage of using it?

Processor Registers
The central processing unit, or CPU, of your computer contains a few data storage locations called *registers*. It is in the CPU registers that actual data operations, such as addition and division, take place. To manipulate data, the CPU must move the data from memory to its registers, perform the manipulations, and then move the data back to memory. Moving data to and from memory takes a finite amount of time. If a particular variable could be kept in a register to begin with, manipulations of the variable would proceed much faster. Click the Tip button on the toolbar.

DO initialize local variables or you won't know what value they will contain.

DO initialize global variables even though they're initialized to 0 by default. If you always initialize your variables, you avoid problems such as forgetting to initialize local variables.

DO pass local variables as function parameters instead of declaring them as global if they are needed in only a few functions.

DON'T use register variables for nonnumeric values, structures, or arrays.

Benefits of the register Storage Class
By using the register keyword in the definition of an automatic variable, you ask the compiler to store that variable in a register. Take a look at the following example:

```
void func1(void) {
    register int x;
    /* Additional code goes here */
}
```

Note that the term is "ask" and not "tell." Depending on the needs of the program, a register may not be available for the variable. In this case, the compiler treats it as an ordinary automatic variable. The register keyword is a suggestion—not an order! The benefits of the register storage class are largest for variables that are used frequently by the function, such as the counter variable for a loop.

Where the register Keyword Can Be Used
The register keyword can be used only with simple numeric variables, not arrays or structures. Also, it cannot be used with either static or external storage classes. You cannot define a pointer to a register variable.

Topic 5.4: Local Variables and the main() Function

Local Variables and the main() Function

Everything said so far about local variables applies to main(), as well as to all other functions. Strictly speaking, main() is a function like any other. The main() function is called when the program is started from DOS, and control is returned to DOS from main() when the program terminates.

Variables Are Automatic

This means that local variables defined in main() are created when the program begins, and their lifetime is over when the program ends. The notion of a static local variable retaining its value between calls to main() really makes no sense: a variable cannot remain in existence between program executions. Within main(), therefore, there is no difference between automatic and static local variables. You can define a local variable in main() as being static, but it has no effect. Click the Tip button on the toolbar.

DO remember that main() is a function similar in most respects to any other function.

DON'T declare static variables in main() because doing so gains nothing.

Topic 5.5: Which Storage Class Should You Use?

Summary of Storage Classes

When you're deciding which storage class to use for particular variables in your programs, click here to see a table that summarizes the five storage classes available in C.

Table 12.1: C's Five Variable Storage Classes

Storage Class	Keyword	Lifetime	Where Defined	Scope
Automatic	None(1)	Temporary	In a function	Local
Static	static	Permanent(3)	In a function	Local
Register	register	Temporary	In a function	Local

External	None(2)	Permanent	Outside a function	Global (all files)
External	static	Permanent	Outside a function	Global (one file)
(1) The auto keyword is optional.				
(2) The extern keyword is used in functions to declare a static external variable that is defined elsewhere.				
(3) The static variable is permanent after the function is called for the first time.				

Guidelines for Deciding on a Storage Class
When you're deciding on a storage class, you should use an automatic storage class whenever possible and use other classes only when needed. Here are some guidelines to follow:

- Give each variable automatic local storage class to begin with.
- In functions other than main(), make a variable static if its value must be retained between calls to the function.
- If a variable is used by most or all of the program's functions, define it with the external storage class.

Topic 5.6: Local Variables and Blocks

Defining Local Variables Within Blocks
So far, this unit has discussed only variables that are local to a function. This is the primary way local variables are used, but you can define variables that are local to any program block (any section enclosed in braces). When declaring variables within the block, you must remember that the declarations must be first. For an example, see Listing 12.5.

Usage is Not Common
The use of this type of local variable is not common in C programming, and you may never find a need for it. Its most common use is probably when a programmer tries to isolate a problem within a program. You can temporarily isolate sections of code in braces and establish local variables to assist in tracking down the fault. Another advantage is that the variable declaration-initialization can be placed closer to the point where it's used, which can help in understanding the program.Click the Tip button.

DON'T try to put variable definitions at any place within a function other than at the beginning of the function or at the beginning of a block.

DON'T use variables at the beginning of a block unless it makes the program clearer.

DO use variables at the beginning of a block (temporarily) to help track down problems.

Listing 12.5. A program that Demonstrates Defining Local Variables Within a Program Block

Code

```
 1: /* LIST1205.c: Day 12 Listing 12.5 */
 2: /* Demonstrates local variables within blocks. */
 3:
 4: #include <stdio.h>
 5:
 6: int main(void) {
 7:   /* Define a variable local to main(). */
 8:
 9:   int count = 0;
10:
11:   printf("\nOutside the block, count = %d", count);
12:
13:   /* Start a block. */
14:   {
15:   /* Define a variable local to the block. */
16:
17:   int count = 999;
18:   printf("\nWithin the block, count = %d", count);
19:   }
20:
21:   printf("\nOutside the block again, count = %d",
22:          count);
23:   return 0;
24: }
```

Output

```
Outside the block, count = 0
Within the block, count = 999
Outside the block again, count = 0
```

Description

From this program, you can see that the count defined within the block is independent of the count defined outside the block.

Line 9 defines count as a type int variable equal to 0. Because it is declared at the beginning of main(), it can be used throughout the entire main() function. Line 11 shows that count has been initialized to zero by printing a message.

	In lines 14 – 19, a block is declared , and within the block, another count variable is defined as a type int variable. This count variable is initialized to 999 on line 17. Line 18 prints the block's count variable value of 999. Finally, because the block ends in line 19, the print statement in line 21 uses the original count initially declared in line 9 of main().

Topic 5.7: Day 12 Q&A

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

Question 1
If global variables can be used anywhere in the program, why not make all variables global?

Answer
As your programs get bigger, you will begin to declare more and more variables. As stated in the unit, there are limits on the amount of memory available. Variables declared as global take up memory for the entire time the program is running; however, local variables do not. For the most part a local variable takes up memory only while the function to which it is local is active. (A static variable takes up memory from the time it is first used to the end of the program.) Additionally, global variables are subject to unintentional alteration by other functions. If this occurs, the variables may not contain the values you expect them to when they are used in the functions for which they were created.

Question 2
Day 11, "Structures," stated that scope affects a structure instance but not a structure tag or body. Why doesn't scope affect structure tag or body?

Answer
When declaring a structure without instances, you are creating a template. You do not actually declare any variables. It is not until you create an instance of the structure that you declare a variable. For this reason, you can leave a structure body external to any functions with no real effect on external memory. Many programmers put commonly used structure bodies with tags into header files, and then include these header files

when they need to create an instance of the structure. (Header files are covered on Day 20, "Odds and Ends.")

Question 3
How does the computer know the difference between a global and a local variable?

Answer
The answer to this question is beyond the scope of this unit. The important thing to know is that when a local variable is declared with the same name as a global variable, the program temporarily ignores the global variable. It continues to ignore the global variable until the local variable goes out of scope.

Question 4
Can I declare a local variable with a different variable type, yet use the same variable name as a global variable?

Answer
Yes. When you declare a local variable with the same name as a global variable, it is a completely different variable. This means you can make it whatever type you want. You should be careful not to confuse the two, however, when declaring global and local variables with the same name.

Topic 5.8: Day 12 Think About Its

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

```
1 void a_sample_function() {
2   char star = '*'; {
3   int ctr1;
4     for (ctr1 = 0; ctr1 < 25; ctr1++)
5       printf( "*" );
6   }
7     puts( "\nThis is a sample function" );
8     puts( "It has a problem" );
9     for (ctr1 = 0; ctr1 < 25; ctr1++)
10       printf( "%c", star);
11 }
```

Topic 5.9: Day 12 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 declaration for a variable to be placed in a CPU register.

Answer

```
register int x = 0;
```

* Exercise 2

Change Listing 12.2 to prevent the error. Do this without using any external variables. 2.

Answer

Listing 12.2: The Variable x is Not Accessible Within the Function print_value

Code

```
/* LIST1202.c: Day 12 Listing 12.2 */
/* Illustrates variable scope. */

#include <stdio.h>

void print_value(void);

int main(void) {
  int x = 999;

  printf("%d\n", x);
  print_value();
  return 0;
}

void print_value(void) {
  printf("%d\n", x);
}
```

```
/* EXER1202.c: Day 12 Exercise 12.2 */
/* Illustrates variable scope. */

#include <stdio.h>

void print_value(int x);

int main(void) {
  int x = 999;

  printf("\n%d", x);
  print_value(x);
  return 0;
}

void print_value(int x) {
  printf("\n%d", x);
}
```

*** Exercise 3**

Write a program that declares a global variable of type int called var. Initialize var to any value. The program should print the value of var in a function (not main()). Do you need to pass var as a parameter to the function?

Answer

Because you are declaring var as a global, you do not need to pass it as a parameter.

```
/* EXER1203.c: Day 12 Exercise 12.3 */
/* Using a global variable. */

#include <stdio.h>

int var = 99;

void print_value(void);

int main(void) {
  print_value();
  return 0;
}
```

```
void print_value(void) {
  printf("The value is %d", var);
}
```

*** Exercise 4**

Change the program in exercise three. Instead of declaring var as a global variable, change it to a local variable in main(). The program should still print var in a separate function. Do you need to pass var as a parameter to the function?

Answer

Yes, you need to pass the variable var in order to print it in a different function.

```
/* EXER1204.c: Day 12 Exercise 12.4 */
/* Using a local variable */

#include <stdio.h>

void print_value(int var);

int main(void) {
  int var = 99;
  print_value(var);
  return 0;
}

void print_value(int var) {
  printf("The value is %d", var);
}
```

*** Exercise 5**

Write a program that uses a global and a local variable with the same name.

Answer

A program can have a local and global variable with the same name. In such cases, active local variables take precedence.

```
/* EXER1205.c: Day 12 Exercise 12.5 */
```

```
/* Using a global variable */

#include <stdio.h>

int var = 99;

void print_func(void);

int main(void) {
  int var = 77;
  printf("Printing in function with local and global:");
  printf("\nThe value of var is %d", var);
  print_func();
  return 0;
}

void print_func(void) {
  printf("\nPrinting in function only global:");
  printf("\nThe value of var is %d", var);
}
```

*** Exercise 6**

What do you notice that is unusual about the variable declarations within another_sample_function?

```
void another_sample_function() {
  int ctr1;

  for (ctr1 = 0; ctr1 < 25; ctr1++)
      printf( "*" );

  puts( "\nThis is a sample function" ); {
      char star = '*';
      puts( "It has a problem" );
      for (int ctr2 = 0; ctr2 < 25; ctr2++) {
           printf( "%c", star);
      }
  }
}
```

Answer

The variable ctr2 is not declared at the beginning of the block — instead is declared inside the for loop. This used to be unacceptable practice in C programming, but now

it is preferred practice to declare counter variables inside or near the loop where they will be used. This is an exception to the rule of declaring variables at the beginning of the code block.

*** Exercise 7**

BUG BUSTERS: What is wrong with the following code?

```
/* EXER1207.c: Day 12 Exercise 12.7 */
/* Count the number of even numbers from 0 to 100 */

#include <stdio.h>

int main(void) {
  int x = 1;
  static long tally = 99;

  for(x = 0; x < 100; x++)
    if(x%2 == 0)  /* if x is an even number ... */
      tally++;  /* then add 1 to tally!       */
  return 0;
}
```

Answer

This program is actually okay; however, it could be better. There is absolutely no reason to declare the tally variable as static. A static variable in main() is equivalent to a local (automatic) variable in main().

*** Exercise 8**

BUG BUSTERS: Is anything wrong with the following program?

```
/* EXER1208.c: Day 12 Exercise 12.8 */

#include <stdio.h>

void print_function(char star);

int ctr;
```

```
int main(void) {
  char star;

  print_function(star);
  return 0;
}

void print_function(char star) {
  char dash;

  for(ctr = 0; ctr < 25; ctr++) {
    printf("%c%c", star, dash);
  }
}
```

Answer

What is the value of star? What is the value of dash? These two variables were never initialized. Because they are both local variables, each could contain any value. Notice that this program compiles with no errors or warnings, but there is a problem.

There is a second issue that should be brought up about this program. The variable ctr is declared as a global, but it is only used in print_function(). This is not a good assignment. The program would be better if ctr was local variable in print_function().

*** Exercise**
What does the following program print?

```
/* EXER1209.c: Day 12 Exercise 12.9 */

#include <stdio.h>

void print_letter2(void);

int ctr;
char letter1 = 'X';
char letter2 = '=';

int main(void) {
  for(ctr = 0; ctr < 10; ctr++) {
     printf("%c", letter1);
     print_letter2();
  }
```

```
  return 0;
}

void print_letter2(void) {
  for(ctr = 0; ctr < 2; ctr++)
    printf("%c", letter2);
}
```

Answer

This program prints the following pattern forever. See exercise 10.

```
X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==...
```

*** Exercise 1**
BUG BUSTERS: What is wrong with the preceding program? Rewrite it so that it is correct.

Answer

This program poses a problem because of the global scope of ctr. Both the main() function and print_letter2() function use ctr in loops at the same time. Because print_letter2() changes the value, the for loop in main() never completes. This could be fixed in a number of ways. One way is to use two different counter variables. A second way is to change the scope of the counter variable, ctr. It could be declared in both main() and print_letter2() as local variables.

An additional comment on letter1 and letter2. Because each of these is only used in one function, they should be declared as locals. Following is the corrected listing.

```
/* EXER1210.c: Day 12 Exercise 12.10 */

#include <stdio.h>

void print_letter2(void);

int main(void) {
  char letter1 = 'X';
  int ctr;

  for(ctr = 0; ctr < 10; ctr++) {
```

```
      printf("%c", letter1);
      print_letter2();
  }
  return 0;
}

void print_letter2(void) {
  char letter2 = '=';
  int ctr;  /* This is a local variable. */
            /* It is different from ctr in main(). */

  for(ctr = 0; ctr < 2; ctr++)
    printf("%c", letter2);
}
```

Topic 5.10: Day 12 Summary

What Storage Classes Determine
This unit covered C's variable storage classes. Every C variable, whether a simple variable, an array, a structure, or whatever, has a specific storage class that determines 1) its scope or where in the program it's visible and 2) its lifetime or how long the variable persists in memory.

Which Storage Class to Use
Proper use of storage classes is an important aspect of structured programming. By keeping most variables local to the function that uses them, you enhance the independence of functions from each other. A variable should be given automatic storage class unless there is a specific reason to make it external or static.

Unit 6. Day 13 More Program Control

Day 6, "Basic Program Control," introduced some of C's program control statements that govern the execution of other statements in your program. This unit covers more advanced aspects of program control, including the goto statement and some of the more interesting things you can do with loops in your programs.

Today, you learn

- How to use the break and continue statements.
- What infinite loops are and why you might use them.
- What the goto statement is and why you should avoid it.
- How to use the switch statement.
- How to control program exits.
- How to execute functions automatically on program completion.
- How to execute system commands in your program.

Topic 6.1: Ending Loops Early

Exiting Loops When a Condition Occurs

On Day 6, "Basic Program Control," you learned how the for loop, the while loop, and the do...while loop can control program execution. These loop constructions execute a block of C statements never, once, or more than one time, depending on conditions in the program. In all three cases, termination or exit of the loop occurs only when a certain condition occurs.

Ending Loops Early

At times, however, you might want to exert more control over loop execution. The break and continue statements provide this control.

Topic 6.1.1: The break Statement

The break Statement

The break statement can be placed only in the body of a for loop, a while loop, or a do...while loop. (It's valid in a switch statement too, but that topic isn't covered until later in this unit.) When a break statement is encountered, execution exits the loop. Click the Example link.

Example

When a break statement is encountered inside a nested loop, it causes exit of the innermost loop only.

```
for (count = 0; count < 10; count++) {
    if (count == 5)
        break;
}
```

Left to itself, the for loop would execute ten times. On the sixth iteration, however, count is equal to 5, and the break statement executes, causing the for loop to terminate. Execution then passes to the statement immediately following the for loop's closing brace.

Example Program

The program in Listing 13.1 demonstrates break usage.

Multiple break Statements

A loop can contain multiple break statements. Only the first break executed (if any) has any effect. If none are executed, the loop terminates normally (according to its test condition). Figure 13.1 shows the operation of the break statement.

Listing 13.1: Use of the break Statement

```
Code   1: /* LIST1301.c: Day 13 Listing 13.1 */
       2: /* Demonstrates the break statement. */
       3:
       4: #include <stdio.h>
       5:
       6: char s[] = "This is a test string. It
       7: contains two sentences.";
       8:
       9: int main(void) {
      10:   int count;
```

```
11:
12:   printf("\nOriginal string: %s", s);
13:
14:   for (count  = 0; s[count] != '\0'; count++)
15:     if (s[count] == '.') {
16:       s[count+1] = '\0';
17:       break;
18:     }
19:
20:   printf("\nModified string: %s", s);
21:   return 0;
22: }
```

Output

```
Original string: This is a test string. It
contains two sentences.
Modified string: This is a test string.
```

Description

The program extracts the first sentence from a string. It searches the string, character by character, for the first period (which should mark the end of a sentence). This is done in the for loop in lines 14 – 18.

Line 14 starts the for loop, incrementing count to go from character to character in the string, s.

Line 15 checks to see whether the current character in the string is equal to a period.

If it is, line 16 inserts a null character immediately after the period. This, in effect, trims the string.

Once you trim the string, you no longer need to continue the loop, so a break statement on line 17 quickly terminates the loop and sends control to the first line after the loop (line 19).

If no period is found, the string is not altered.

Topic 6.1.2: The continue Statement

The continue Statement

Like the break statement, the continue statement can be placed only in the body of a for loop, a while loop, or a do...while loop. When a continue statement executes, the next iteration of the enclosing loop begins immediately. The statements between the

continue statement and the end of the loop are not executed. The operation of continue is shown in Figure 13.1. Notice how this differs from the operation of a break statement.

A program that uses continue is presented in Listing 13.2. The program accepts a line of input from the keyboard, and then displays it with all lowercase vowels removed.

Listing 13.2: Demonstration of the continue Statement

Code

```
/* LIST1302.c: Day 13 Listing 13.2 */
/* Demonstrates the continue statement. */

#include <stdio.h>

int main(void) {
  /* Declare a buffer for input and a counter */
  /* variable. */

  char buffer[81];
  int ctr;

  /* Input a line of text. */

  puts("Enter a line of text:");
  gets(buffer);

  /* Go through the string, displaying only */
  /* those characters that are not lowercase */
  /* vowels. */

  for (ctr = 0; buffer[ctr] !='\0'; ctr++) {

    /* If the character is a lowercase vowel, */
    /* loop back without displaying it. */

    if (buffer[ctr] == 'a' || buffer[ctr] == 'e' ||
      buffer[ctr] == 'i'
      || buffer[ctr] == 'o' || buffer[ctr] == 'u')
      continue;

    /* If not a vowel, display it. */

    putchar(buffer[ctr]);
  }
  return 0;
}
```

Output

```
Enter a line of text:
This is a line of text
Ths s  ln f txt
```

Description	Although this is not the most practical program, it does use a continue statement effectively. Lines 10 and 11 declare the program's variables. On line 16, buffer[ ] is used to store the string the user enters. On lines 22 – 35 the other variable, ctr, increments through buffer while the for loop searches for vowels. For each letter in the array, an if statement on lines 27 – 30 checks the letter against lowercase vowels. If there is a match, a continue statement executes on line 30, sending control back to line 22, the for statement. If the letter is not a vowel, control passes to the next statement and line 34 is executed. Note that line 34 contains a new library function, putchar(), which displays a single character on the screen at a time.

Topic 6.2: The goto Statement

Unconditional Jump
The goto statement is one of C's *unconditional jump,* or *branching*, statements. When program execution reaches a goto statement, execution immediately jumps, or branches, to the location specified by the goto statement. The statement is *unconditional* because execution always branches when a goto statement is encountered; the branch does not depend on any program conditions (unlike if statements, for example). Click the Tip button on the toolbar.

DO avoid using the goto statement if possible.

DON'T confuse break and continue. break ends a loop, whereas continue starts the next iteration.

Syntax: goto Statement
The syntax of the goto statement is

```
goto target;
```

target is a label statement that identifies the program location where execution is to

branch.

Syntax: label Statement

A *label statement* consists of an identifier followed by a colon and a C statement.

```
location1: a C statement;
```

If you want the label by itself on a line, you can follow it with the null statement (a semicolon by itself).

```
location1: ;
```

A goto statement and its target label must be in the same function, although they can be in different blocks.

Example Program

Take a look at Listing 13.3, a simple program that uses a goto statement.

Listing 13.3: Demonstration of the goto Statement

```
Code  1: /* LIST1303.c: Day 13 Listing 13.3 */
      2: /* Demonstrates the goto statement. */
      3:
      4: #include <stdio.h>
      5:
      6: int main(void) {
      7:   int n;
      8:
      9: start: ;
     10:
     11:   puts("Enter a number between 0 and 10: ");
     12:   scanf("%d", &n);
     13:
     14:   if (n < 0 || n > 10)
     15:     goto start;
     16:   else if (n == 0)
     17:     goto location0;
     18:   else if (n==1)
     19:     goto location1;
     20:   else
     21:     goto location2;
     22:
     23: location0: ;
     24:   puts("You entered 0.");
     25:   goto end;
     26:
     27: location1: ;
     28:   puts("You entered 1.");
```

```
29:   goto end;
30:
31: location2: ;
32:   puts("You entered something between 2 and 10.");
33:
34: end: ;
35: return 0;
36: }
```

Output

```
>list1303
Enter a number between 0 and 10:
1
You entered 1.

>list1303
Enter a number between 0 and 10:
9
You entered something between 2 and 10.
```

Description

This is a simple program that accepts a number between 0 and 10. If the number is not between 0 and 10, the program uses a goto statement on line 15 to go to start, which is on line 9. Otherwise, the program checks on line 16 to see whether the number equals 0. If it does, a goto statement on line 17 sends control to location0 (line 23), which prints a statement on line 24 and executes another goto. The goto on line 25 sends control to end at the end of the program. The program executes the same logic for the value of 1 and all values between 2 and 10 as a whole.

Never Use the goto Statement

The target of a goto statement can come either before or after that statement in the code. The only restriction, as mentioned previously, is that both the goto and the target must be in the same function. They can be in different blocks, however; you can use goto to transfer execution both into and out of loops, such as a for statement. We strongly recommend that you never use the goto statement anywhere in your programs for these two reasons:

• You don't need it. There are no programming tasks that require the goto statement. You always can write the needed code using C's other branching statements.

• It's dangerous. The goto statement may seem like an ideal solution for certain programming problems, but it is easy to abuse. When program execution branches with a goto statement, no record is kept of where the execution came from, so execution can weave willy-nilly in the program. This type of programming is

known in the trade as *spaghetti code*. Click the Note button on the toolbar.

Now, some careful programmers can write perfectly fine programs that use goto. There might be situations where a judicious use of goto is the simplest solution to a programming problem. It is never the only solution, however. If you're going to ignore this warning, at least be careful!

Topic 6.3: Infinite Loops

An Infinite while Loop
What is an infinite loop and why would you want one in your program? An infinite loop is one that, left to its own devices, would run forever. It can be a for loop, a while loop, or a do...while loop. Click the Example link.

Example

Used with break Statements
In the last section, however, you saw that the break statement can be used to exit a loop. Without the break statement, an infinite loop would be useless. With break, you can take advantage of infinite loops.

If you write

```
while (1) {
    /* additional code goes here */
}
```

you create an infinite loop. The condition that the while tests is the constant 1, which is always true and cannot be changed by program execution. Therefore, on its own, the loop never terminates.

An Infinite for Loop or do...while Loop
You also can create an infinite for loop or an infinite do...while loop, as follows. Click the Example link.

Example

The principle remains the same for all three loop types. This section's examples use the while loop.

Used to Test Conditions

An infinite loop can be used to test many conditions and determine whether the loop should terminate. The reason this structure might be used is that it might be difficult to include all the test conditions in parentheses after the while statement. It might be easier to test the conditions individually in the body of the loop, and then exit by executing a break as needed.

```
for (;;) {
    /* additional code goes here */
}
do {
    /* additional code goes here */
} while (1);
```

Used to Create a Menu System

An infinite loop also can create a menu system that directs your program's operation. You may remember from Day 5, "Functions: the Basics," that a program's main() function often serves as a sort of "traffic cop," directing execution among the various functions that do the real work of the program. This is often accomplished by a menu of some kind: the user is presented with a list of choices and makes an entry by selecting one of them. One of the available choices should be to terminate the program. Once a choice is made, one of C's decision statements is used to direct program execution accordingly.

The program in Listing 13.4 demonstrates a menu system.

Listing 13.4: Using an Infinite Loop to Implement a Menu System

```
Code  1: /* LIST1304.c: Day 13 Listing 13.4 */
      2: /* Demonstrates using an infinite loop to */
      3: /* implement a menu system. */
      4:
      5: #include <stdio.h>
      6:
      7: #define DELAY 150000    /* Used in delay loop. */
      8:
      9: int menu(void);
     10: void delay(void);
     11:
     12: int main(void) {
```

```
13:   int choice;
14:
15:   while (1) {
16:
17:   /* Get the user's selection. */
18:
19:   choice = menu();
20:
21:   /* Branch based on the input. */
22:
23:   if (choice == 1) {
24:       puts("\nExecuting choice 1.");
25:       delay();
26:     }
27:
28:   else if (choice == 2) {
29:       puts("\nExecuting choice 2.");
30:       delay();
31:     }
32:
33:   else if (choice == 3) {
34:       puts("\nExecuting choice 3.");
35:       delay();
36:     }
37:
38:   else if (choice == 4) {
39:       puts("\nExecuting choice 4.");
40:       delay();
41:     }
42:
43:   else if (choice == 5)    /* Exit program. */ {
44:       puts("\nExiting program now...");
45:       delay();
46:       break;
47:     }
48:   else {
49:       puts("Invalid choice, try again.");
50:       delay();
51:     }
52:   }
53:   return 0;
54: }
55:
56: int menu(void) {
57: /* Displays a menu and inputs user's selection. */
58:   int reply;
59:
60:   puts("\nEnter 1 for task A.");
61:   puts("Enter 2 for task B.");
62:   puts("Enter 3 for task C.");
63:   puts("Enter 4 for task D.");
64:   puts("Enter 5 to exit program.");
```

```
65:
66:   scanf("%d", &reply);
67:
68:   return reply;
69: }
70:
71: void delay(void) {
72:   long x;
73:   for (x=0; x < DELAY; x++)
74:     ;
75: }
```

Output

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
1

Executing choice 1.

Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.

Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
5
Exiting program now...
```

Description

In Listing 13.4, a function named menu() is called on line 19 and defined on lines 56 – 69. menu() displays a menu on the screen, accepts user input, and returns the input to the main program. In main(), a series of nested if statements tests the returned value and directs execution accordingly. The only thing this program does is display messages on the screen. In a real program, the code would call various functions and perform the selected task.

This program also uses a second function, named delay(). delay() is defined on lines 71 – 75 and really does not do much. Simply stated, the for statement on line 73 loops, doing nothing (line 74). The statement loops DELAY times. This is an effective

	method to pause the program momentarily. If the delay is too short or too long, the defined value of DELAY can be adjusted accordingly. Both Borland and Symantec offer a function similar to delay(), called sleep(). This function pauses program execution for the number of seconds that is passed as its argument. To use sleep(), a program must include the header file TIME.H if you are using the Symantec compiler. You must use DOS.H if you are using a Borland compiler. If you are using either of these compilers, or a compiler that supports sleep(), you could use it instead of delay().

Topic 6.4: The switch Statement

The switch Statement
C's most flexible program control statement is the switch statement, which enables your program to execute different statements based on an expression that can have more than two values. Earlier control statements, such as if, were limited to evaluating an expression that could have only two values: true or false. To control program flow based on more than two values, you had to use multiple, nested if statements, as Listing 13.4 illustrated. The switch statement makes such nesting unnecessary.

Syntax
The general form of the switch statement is as follows:

```
switch (expression) {
    case  constant-1: statement(s);
    case  constant-2: statement(s);
    ...
    case  constant-n: statement(s);
    default: statement(s);
}
```

In this statement, expression is any expression that evaluates to an integer value: type long, int, or char.

Process
The switch statement evaluates `expression` and compares the value against the constants following each case label. Then:

- If a match is found between `expression` and one of the constants, execution is

transferred to the statement that follows the case label.

• If no match is found, execution is transferred to the statement following the optional default label.

• If no match is found and there is no default label, execution passes to the first statement following the switch statement's closing brace. Click the Tip button on the toolbar.

DON'T forget to use break statements if your switch statements need them.

DO use a default case in a switch statement even if you think you have covered all possible cases.

DO use a switch statement instead of an if statement if there are more than two conditions being evaluated for the same variable.

DO line up your case statements so that they are easy to read.

Example Program: Without break

The switch statement is demonstrated by the simple program in Listing 13.5 that displays a message based on the user's input. Compile and run the program, entering 2 at the program prompt. Take a look at the output.

The output is not right, is it? It looks like the switch statement finds the first matching constant, and then executes everything that follows (not just the statements associated with the constant). That's exactly what does happen, though. That's how switch is supposed to work. In effect, it performs a goto to the matching constant.

Listing 13.5: Demonstration of the switch Statement

```
Code   1: /* LIST1305.c: Day 13 Listing 13.5 */
       2: /* Demonstrates the switch statement. */
       3:
       4: #include <stdio.h>
       5:
       6: int main(void) {
       7:   int reply;
       8:
       9:   puts("Enter a number between 1 and 5:");
      10:   scanf("%d", &reply);
```

```
11:
12:   switch(reply) {
13:     case 1:
14:       puts("You entered 1.");
15:     case 2:
16:       puts("You entered 2.");
17:     case 3:
18:       puts("You entered 3.");
19:     case 4:
20:       puts("You entered 4.");
21:     case 5:
22:       puts("You entered 5.");
23:     default:
24:       puts("Out of range, try again.");
25:   }
26:   return 0;
27: }
```

Output

```
Enter a number between 1 and 5:
2
You entered 2.
You entered 3.
You entered 4.
You entered 5.
Out of range, try again.
```

Example Program: With break

To ensure that only the statements associated with the matching constant are executed, include a break statement where needed. Listing 13.6 shows the program rewritten with break statements. Now it functions properly.

Listing 13.6: Correct Use of switch, Including break Statements as Needed

Code

```
 1: /* LIST1306.c: Day 13 Listing 13.6 */
 2: /* Demonstrates the switch statement correctly. */
 3:
 4: #include <stdio.h>
 5:
 6: int main(void) {
 7:   int reply;
 8:
 9:   puts("Enter a number between 1 and 5:");
10:   scanf("%d", &reply);
11:
12:   switch(reply) {
13:     case 0:
14:       break;
15:     case 1: {
```

```
16:       puts("You entered 1.");
17:       break;
18:       }
19:     case 2: {
20:       puts("You entered 2.");
21:       break;
22:       }
23:     case 3: {
24:       puts("You entered 3.");
25:       break;
26:       }
27:     case 4: {
28:       puts("You entered 4.");
29:       break;
30:       }
31:     case 5: {
32:       puts("You entered 5.");
33:       break;
34:       }
35:     default:
36:       puts("Out of range, try again.");
37:   }
38:   return 0;
39: }
```

Output

```
>list1306
Enter a number between 1 and 5:
1
You entered 1.

>list1306
Enter a number between 1 and 5:
6
Out of range, try again.
```

Example Program: Creating a Menu System

One common use of the switch statement is to implement the sort of menu shown in Listing 13.4. Using switch is much better than using the nested if statements, as used in the earlier version. The program in Listing 13.7 uses switch instead of if. Both listings are shown here.

The exit() Function

There's one other new statement in this version: the exit() library function in the statements associated with case 5: on line 48. You cannot use break here, as you did in Listing 13.4's version. Executing a break would merely break out of the switch statement but would not break out of the infinite while loop. As you learn in the next section, the exit() function terminates the program.

Listing 13.4: Using an Infinite Loop to Implement a Menu System

```
Code  1: /* LIST1304.c: Day 13 Listing 13.4 */
      2: /* Demonstrates using an infinite loop to */
      3: /* implement a menu system. */
      4:
      5: #include <stdio.h>
      6:
      7: #define DELAY 150000    /* Used in delay loop. */
      8:
      9: int menu(void);
     10: void delay(void);
     11:
     12: int main(void) {
     13:   int choice;
     14:
     15:   while (1) {
     16:
     17:   /* Get the user's selection. */
     18:
     19:   choice = menu();
     20:
     21:   /* Branch based on the input. */
     22:
     23:   if (choice == 1) {
     24:       puts("\nExecuting choice 1.");
     25:       delay();
     26:     }
     27:
     28:   else if (choice == 2) {
     29:       puts("\nExecuting choice 2.");
     30:       delay();
     31:     }
     32:
     33:   else if (choice == 3) {
     34:       puts("\nExecuting choice 3.");
     35:       delay();
     36:     }
     37:
     38:   else if (choice == 4) {
     39:       puts("\nExecuting choice 4.");
     40:       delay();
     41:     }
     42:
     43:   else if (choice == 5)    /* Exit program. */ {
     44:       puts("\nExiting program now...");
     45:       delay();
     46:       break;
     47:     }
     48:   else {
     49:       puts("Invalid choice, try again.");
```

```
50:       delay();
51:     }
52:   }
53:   return 0;
54: }
55:
56: int menu(void) {
57: /* Displays a menu and inputs user's selection. */
58:   int reply;
59:
60:   puts("\nEnter 1 for task A.");
61:   puts("Enter 2 for task B.");
62:   puts("Enter 3 for task C.");
63:   puts("Enter 4 for task D.");
64:   puts("Enter 5 to exit program.");
65:
66:   scanf("%d", &reply);
67:
68:   return reply;
69: }
70:
71: void delay(void) {
72:   long x;
73:   for (x=0; x < DELAY; x++)
74:     ;
75: }
```

Output

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
1

Executing choice 1.

Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.

Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
5
Exiting program now...
```

Description	In Listing 13.4, a function named menu() is called on line 19 and defined on lines 56 – 69. menu() displays a menu on the screen, accepts user input, and returns the input to the main program. In main(), a series of nested if statements tests the returned value and directs execution accordingly. The only thing this program does is display messages on the screen. In a real program, the code would call various functions and perform the selected task. This program also uses a second function, named delay(). delay() is defined on lines 71 – 75 and really does not do much. Simply stated, the for statement on line 73 loops, doing nothing (line 74). The statement loops DELAY times. This is an effective method to pause the program momentarily. If the delay is too short or too long, the defined value of DELAY can be adjusted accordingly. Both Borland and Symantec offer a function similar to delay(), called sleep(). This function pauses program execution for the number of seconds that is passed as its argument. To use sleep(), a program must include the header file TIME.H if you are using the Symantec compiler. You must use DOS.H if you are using a Borland compiler. If you are using either of these compilers, or a compiler that supports sleep(), you could use it instead of delay().

Listing 13.7: Using the switch Statement to Execute a Menu System

```
Code  1: /* LIST1307.c: Day 13 Listing 13.7 */
      2: /* Demonstrates using an infinite loop and the */
      3: /* switch statement to implement a menu system. */
      4: #include <stdio.h>
      5: #include <stdlib.h>
      6:
      7: #define DELAY 150000
      8:
      9: int menu(void);
     10: void delay(void);
     11:
     12: int main(void) {
     13:   while (1) {
     14:
     15:   /* Get the user's selection and branch based */
     16:   /* on the input. */
     17:
     18:     switch(menu()) {
     19:       case 1: {
```

```
20:         puts("\nExecuting choice 1.");
21:         delay();
22:         break;
23:         }
24:
25:       case 2: {
26:         puts("\nExecuting choice 2.");
27:         delay();
28:         break;
29:         }
30:
31:       case 3: {
32:         puts("\nExecuting choice 3.");
33:         delay();
34:         break;
35:         }
36:
37:       case 4: {
38:         puts("\nExecuting choice 4.");
39:         delay();
40:         break;
41:         }
42:
43:       case 5:    /* Exit program. */ {
44:         puts("\nExiting program now...");
45:         delay();
46:         exit(0);
47:         }
48:       default: {
49:         puts("Invalid choice, try again.");
50:         delay();
51:         }
52:     }
53:   }
54:   return 0;
55: }
56:
57: int menu(void) {
58: /* Displays a menu and inputs user's selection. */
59:   int reply;
60:
61:   puts("\nEnter 1 for task A.");
62:   puts("Enter 2 for task B.");
63:   puts("Enter 3 for task C.");
64:   puts("Enter 4 for task D.");
65:   puts("Enter 5 to exit program.");
66:
67:   scanf("%d", &reply);
68:
69:   return reply;
70: }
71:
```

```
72: void delay(void) {
73:   long x;
74:   for (x=0; x < DELAY; x++)
75:     ;
76: }
```

Output

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C
Enter 4 for task D.
Enter 5 to exit program.
1

Executing choice 1.

Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
5
Exiting program now...
```

Example Program: Another Way to Use the Switch Statement

Having execution "fall through" parts of a switch construction, however, can be useful at times. Say, for example, you want the same groups of statements executed if one of several values is encountered. Simply omit the break statements and list all the case constants before the statements. This is illustrated by the program in Listing 13.8.

Listing 13.8. Another Way to Use the switch Statement

Code

```
1: /* LIST1308.c: Day 13 Listing 13.8 */
2: /* Demonstrates another use of the switch */
3: /* statement. */
4:
5: #include <stdio.h>
6: #include <stdlib.h>
7:
8: int main(void) {
9:   int reply;
```

```
10:
11:   while (1) {
12:     puts("Enter a value between 1 and 10,"
13:          " 0 to exit: ");
14:     scanf("%d", &reply);
15:
16:     switch(reply) {
17:       case 0:
18:         exit(0);
19:       case 1:
20:       case 2:
21:       case 3:
22:       case 4:
23:       case 5: {
24:         puts("You entered 5 or below.");
25:         break;
26:           }
27:       case 6:
28:       case 7:
29:       case 8:
30:       case 9:
31:       case 10: {
32:         puts("You entered 6 or higher.");
33:         break;
34:         }
35:       default:
36:         puts("Between 1 and 10, please!");
37:
38:     }
39:   }
40:   return 0;
41: }
```

Output

```
Enter a value between 1 and 10, 0 to exit:
11
Between 1 and 10, please!
Enter a value between 1 and 10, 0 to exit:
1
You entered 5 or below.
Enter a value between 1 and 10, 0 to exit:
6
You entered 6 or higher.
Enter a value between 1 and 10, 0 to exit:
0
```

Description

This program accepts a value from the keyboard and then states whether the value is 5 or below, 6 or higher, or not between 1 and 10. If the value is zero, line 18 executes a call to the exit() function, thus ending the program.

Topic 6.5: Exiting the Program

Exiting the Program

A C program normally terminates when execution reaches the closing brace of the main() function. However, you can terminate a program at any time by calling the library function exit(). You also can specify one or more functions to be automatically executed at termination.

Topic 6.5.1: The exit() Function

Description

The exit() function terminates program execution and returns control to the operating system. This function takes a single type int argument that is passed back to the operating system to indicate the success or failure of the program.

Syntax

The syntax of the exit() function is

```
exit(status);
```

If status has a value of 0, it indicates that the program terminated normally. A value of 1 indicates that the program terminated with some sort of error (as defined by you, the programmer).

Return Value

The return value is usually ignored. In a DOS system, you can test the return value with a DOS batch file and the `if errorlevel` statement. This is not a course about DOS, however; you need to refer to your DOS documentation if you want to use a program's return value.

Usage

To use the exit() function, a program must include the header file STDLIB.H. This header file also defines two symbolic constants for use as arguments to the exit() function, as follows:

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

Thus, to exit with a return value of 0, call `exit(EXIT_SUCCESS)`; for a return value of 1, call `exit(EXIT_FAILURE)`.

Topic 6.5.2: The atexit() Function (DOS only)

Description
The atexit() function is used to specify, or register, one or more functions that are automatically executed when the program terminates. This function might not be available on non-DOS systems. You can register as many as 32 functions; at program termination, they are executed in reverse order. The last function registered is the first function executed. When all the functions registered by atexit() have been executed, the program terminates and returns control to the operating system.

Syntax
The prototype of the atexit() function is located in the header file STDLIB.H, which must be included in any program that uses atexit(). The prototype reads as follows:

```
int atexit(void (*func)(void));
```

You may not recognize this format. It means that atexit() takes a function pointer as its argument. (Day 15, "More On Pointers," covers function pointers in more detail.) Functions registered with atexit() must have a return type of void. Click the Tip button.

DON'T overuse the atexit() function. Limit its use to those functions that must be executed if there is a problem when the program terminates, such as closing an opened file.

DO use the exit() command to get out of the program if there is a problem.

Usage
Say you want the functions cleanup1() and cleanup2() executed in that order on termination. Here's how you would do it. Click the Example link and then the Listing button.

Example

Listing 13.9: Using the exit() and atexit() Functions

```
Code  1: /* LIST1309.c: Day 13 Listing 13.9 */
      2: /* Demonstrates the exit() and atexit() */
      3: /* functions. */
      4:
      5: #include <stdio.h>
```

```
 6: #include <stdlib.h>
 7: #include <time.h>
 8:
 9: #define DELAY 150000
10:
11: void cleanup(void);
12: void delay(void);
13:
14: int main(void) {
15:   int reply;
16:
17:   /* Register the function to be called at exit. */
18:
19:   atexit(cleanup);
20:
21:   puts("Enter 1 to exit, any other to continue.");
22:   scanf("%d", &reply);
23:
24:   if (reply == 1)
25:     exit(EXIT_SUCCESS);
26:
27:   /* Pretend to do some work. */
28:
29:   for (reply = 0; reply < 5; reply++) {
30:     puts("Working...");
31:     delay();
32:   }
33:   return 0;
34: }
35:
36: void cleanup(void) {
37:   puts("\nPreparing for exit...");
38:   delay();
39: }
40:
41: void delay(void) {
42:   long x;
43:   for (x = 0; x < DELAY; x++)
44:     ;
45: }
```

Output

```
>list1309
Enter 1 to exit, any other to continue.
1

Preparing for exit...

>list1309
Enter 1 to exit, any other to continue.
8
Working...
Working...
```

```
Working...
Working...
Working...

Preparing for exit...
```

Description

This program registers the function cleanup() on line 19 with the atexit() function. When the program terminates, either with the exit() on line 25 or by reaching the end of main(), cleanup() executes.

Why would you need to use a "cleanup" function at program termination? Clearly, the example in Listing 13.9 does nothing useful; it merely serves to demonstrate the function. In real-world programs, however, often there are tasks that need to be performed at termination: closing files and streams, for example, or releasing dynamically allocated memory. (Files and streams are covered on days 14 and 16.) These things may not mean anything to you now, but they are covered later in the course. Then you can see how to use atexit() in a real situation.

The Program

```
01: #include <stdlib.h>
02:
03: void cleanup1(void);
04: void cleanup2(void);
05:
06: main() {
07:     atexit(cleanup2);
08:     atexit(cleanup1);
09:
10: }  /* End of main() */
11:
12: void cleanup1(void) {
13: /* additional code goes here */
14: }
15:
16: void cleanup2(void) {
17: .../* additional code goes here */
18: }
```

Description

When this program runs, line 7 registers cleanup2() with the atexit() function, and line 8 registers cleanup1(). When the program terminates, the registered functions are called in reverse order: cleanup1() is called, followed by cleanup2(), and the program

then terminates. Functions registered with atexit() are executed whether program termination is caused by executing the exit() function or by reaching the end of main().

The program in Listing 13.9 shows you how the exit() and atexit() functions are used.

Topic 6.6: Executing Operating System Commands in a Program

The system() Function

The C standard library includes a function, system(), that enables you to execute operating system commands in a running C program. This can be useful, allowing you to read a disk's directory listing or format a disk without exiting the program. To use the system() function, a program must include the header file STDLIB.H.

Syntax

The format of system() is

```
system(s-com);
```

The argument `s-com` can be either a string constant or a pointer to a string. Click the Example link.

Example

To obtain a directory listing in DOS, you could write either

```
system("dir");
```

or

```
char *command = "dir";
system(command);
```

Program Control Flow

After the operating system command is executed, execution returns to the program at the location immediately following the call to system(). If the command you pass to

system() is not a valid operating system command, you get a `Bad command or file name` error message before returning to the program.

The use of system() is illustrated in Listing 13.10.

Listing 13.10: Using the system() Function to Execute System Commands

Code

```
/* LIST1310.c: Day 13 Listing 13.10 */
/* Demonstrates the system() function. */
/* MS-DOS Application Only */

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

int main(void) {
  /* Declare a buffer to hold input. */

  char input[40];

  while (1) {
    /* Get the user's command. */

    puts("\nInput the desired DOS command, "
         "blank to exit");
    gets(input);

    /* Exit if a blank line was entered. */

    if (input[0] == '\0')
      exit(0);

    /* Execute the command. */

    system(input);
  }
  return 0;
}
```

Output

```
Input the desired DOS command, blank to exit
dir *.bak

Volume in drive E is STACVOL_000
Directory of E:\BOOK\LISTINGS
LIST1414 BAK      1416 08-22-02   5:18p
        1 file(s)       1416 bytes
                    24068096 bytes free

Input the desired DOS command, blank to exit
```

Description Listing 13.10 illustrates the use of system(). Using a while loop

	in lines 13 – 28, this program enables operating system commands. Lines 16 – 18 prompt the user to enter the operating system command. Lines 22 – 23 call exit() to end the program if Enter was pressed without entering a command. Line 27 calls system() with the command entered by the user.

Passing Commands Other Than Operating Commands
The commands that you pass to system() are not limited to simple operating commands, such as listing directories or formatting disks. You also can pass the name of any executable file or batch file—yes, and that program is executed normally. For example, if you pass the argument LIST1309, you would execute the program called LIST1309. When the LIST1309 program terminates, execution passes back to where the system() call was made.

Restrictions
The only restrictions on using system() have to do with memory. When system() is executed, the original program remains loaded in your computer's RAM, and a new copy of the operating system command processor and any program you run are loaded as well. This works only if the computer has sufficient memory. If not, you get an error message.

Topic 6.7: Day 13 Q&A

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

Question 1
Is it better to use a switch statement or a nested loop?

Answer
If you are checking a variable, usually a switch statement is more efficient. Many times it is also easier to read. As a general rule, if there are only two options, go with an if statement. If there are more than two options, a switch statement is more efficient.

Question 2
Why should I avoid a goto statement?

Answer
When you first see a goto statement, it is easy to believe that it could be useful. goto, however, can cause you more problems than it fixes. A goto statement is an unstructured command that takes you to another point in a program. Many debuggers (software that helps you to trace program problems) cannot interrogate a goto properly. goto statements also lead to spaghetti code—code that goes all over the place.

Question 3
Why don't all compilers have the same functions?

Answer
In this unit you should have noticed that not all functions are available from all compilers or for all computer systems. The atexit() function is only available on DOS systems. The sleep() function is provided by Borland and Symantec, but not by Microsoft. Each compiler is different, but most have the same overall functionality and features. Where functions are not available, it is usually possible to create those functions yourself.

The answer to the question is a matter of what each company that produces the compiler believes to be important. This same answer can be applied to word processors or spreadsheets. Not every word processor or spreadsheet has the same functionality as every other package of a similar function.

Question 4
Is it good to use the system() function to execute system functions?

Answer
The system() function might appear to be an easy way to do such things as list the files in a directory; however, caution should be used. Most operating system commands are specific to a particular operating system. If you use a system() call, your code probably won't be portable. If you want to run another program (not an operating system command), you shouldn't have portability problems.

Topic 6.8: Day 13 Think About Its

Think About Its

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

```
switch( answer ) {
    case 'Y': printf("You answered yes");
              break;
    case 'N': printf( "You answered no");

    default: printf("Please try again.");
}
```

```
do {
/* any C statements */
} while ( 1 );
```

Topic 6.9: Day 13 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 code that causes the functions f1(), f2(), and f3() to execute, in that order, on program termination.

Answer

```
atexit(f3);
atexit(f2);
atexit(f1);
```

*** Exercise 2**
BUG BUSTERS: Is anything wrong with the following code? If so, what?

```
switch( choice ) {
```

```
    default:
           printf("You did not choose 1 or 2");
    case 1:
           printf("You answered 1");
           break;
    case 2:
           printf( "You answered 2");
           break;
}
```

Answer

You might think the default needs to go at the bottom of the switch statement; that is not true. The default can go anywhere. There is a problem, however. There should be a break statement at the end of the default case.

*** Exercise 3**
Rewrite exercise 2 using if statements.

Answer

```
if( choice == 1 )
    printf("You answered 1");
else if( choice == 2 )
    printf( "You answered 2");
else
    printf( "You did not choose 1 or 2");
```

*** Exercise 4**
Write a program that works like a calculator. The program should allow for addition, subtraction, multiplication, and division.

Answer

Here is one possible answer.

```
/* EXER1309.C Day 13 Exercise 4 */
/* This program functions like a calculator */
/* Calculations allowed: Addition           */
```

```
/*                      Multiplication     */
/*                      Subtraction        */
/*                      Division           */

#include <stdio.h>
int subtot;
int a, b;
char oper;
void clear_kb(void);

int main(void)  {

/* Input the first number */
  printf("1st: Enter a number between 1 and 100: \n");
  scanf("%d", &a);

  clear_kb();

  subtot = a;

  while (1)  {

/* Input the operator */
    printf("Enter the operator (+, -, /, *, =): \n");
    oper = getchar();

    clear_kb();

    if (oper == '=')
      break;
/* Input the next number */
    printf("Enter a number between 1 and 100: \n");
    scanf("%d", &b);

    clear_kb();

/* Perform calculations */
    switch (oper)  {
      case '+':  {
    subtot = subtot + b;
    break;
  }
      case '-':  {
    subtot = subtot - b;
    break;
  }
      case '*':  {
    subtot = subtot * b;
    break;
```

```
  }
      case '/':  {
    subtot = subtot / b;
    break;
  }
    }
  }
  printf("=%d\n", subtot);
  return 0;
}

/* Function: clear_kb() */
/* Purpose: This function clears the keyboard of extra */
/* characters.*/
/* Returns: Nothing                                    */

void clear_kb(void)  {
  char junk[80];
  gets(junk);
}
```

*** Exercise 5**

Write a program that provides a menu with five different options. The fifth option should quit the program. Each of the other options should execute a system command using the system() function.

Answer

Here is one possible answer.

```
/* EXER1305.C Day 13 Exercise 5 */
/* Displays a menu which processes DOS commands */

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

#define DELAY 300000000

int menu(void);
void delay(void);

int main(void)  {
  while (1)  {
    switch(menu())  {
      case 1:  {
```

```
            system("cls");
            system("time");
            break;
          }
          case 2:  {
            system("cls");
            system("dir *.exe | more");
            delay();
            break;
          }
          case 3:  {
            system("cls");
            system("date");
            break;
          }
          case 4:  {
            system("cls");
            system("format b:");
            break;
          }
          case 5:  {
            exit(0);
            break;
          }
          default:  {
      printf("Invalid choice.  Try again.");
      delay();
      }
        }
      }
    }

    int menu(void)  {
    /* Displays a menu and inputs user's selection */
      int reply;
      system("cls");
      puts("\nEnter the number of any entry");
      puts("1 - System time");
      puts("2 - List EXE files");
      puts("3 - System date");
      puts("4 - Format disk on Drive B");
      puts("5 - Exit");

      scanf("%d", &reply);

      return reply;
    }

    void delay(void)  {
      long x;
      for (x = 0; x < DELAY; x++)
```

```
    ;
}
```

Topic 6.10: Day 13 Summary

Ending Loops Early and Infinite Loops
This unit covered a variety of topics related to program control. You learned about the goto statement and why you should avoid using it in your programs. You saw that the break and continue statements give additional control over the execution of loops and that these statements can be used in conjunction with infinite loops to perform useful programming tasks.

Exiting the Program
This unit also explained how to use the exit() function to control program termination and how atexit() can be used to register functions to be executed automatically on program completion.

Executing Operating System Commands in a Program
Finally, you saw how to use the system() function to execute system commands from within your program.

Unit 7. Day 14 Working with the Screen, Printer, and Keyboard

Almost every program must perform input and output. How well a program handles input and output is often the best judge of a program's usefulness. You've already learned how to perform some basic input and output.

Today, you learn

- How C uses *streams* for input and output.
- Various ways of accepting input from the keyboard.
- Methods of displaying text and numeric data on the screen.
- How to send output to the printer.
- How to redirect program input and output.

Topic 7.1: Streams and C

Streams
Before you get to the details of program input/output, you need to learn about *streams*. All C input/output is done with streams, no matter where input is coming from or where output is going to. As you will see later, this standard way of handling all input and output has definite advantages for the programmer. Of course, this makes it essential that you understand what streams are and how they work. First, however, you need to know exactly what the term *input/output* means.

RAM
As you learned earlier in this course, a C program keeps data in random access memory (RAM) while executing. This data is in the form of variables, structures, and arrays that have been declared by the program. Where did this data come from, and what can the program do with it?

Topic 7.1.1: What Exactly Is Program Input/Output?

Input
Data can come from some location external to the program. Data moved from an external location into RAM, where the program can access it, is called *input*. The keyboard and disk files are the most common sources of program input.

Output
Data also can be sent to a location external to the program; this is called *output*. The most common destinations for output are the screen, a printer, and disk files.

Devices
Input sources and output destinations are collectively referred to as *devices*. The keyboard is a device, the screen is a device, and so on. Some devices (the keyboard) are for input only; others (the screen), for output only; and still others (disk files), for both input and output. This is illustrated in Figure 14.1.

Whatever the device, and whether it's performing input or output, C carries out all input and output operations by means of streams.

Streams
A *stream* is nothing more than a sequence of characters. More exactly, it is a sequence of bytes of data. A sequence of bytes flowing into a program is an input stream; a sequence of bytes flowing out of a program is an output stream.

Major Advantage of Streams
The major advantage of streams is that input/output programming is *device independent*. Programmers don't need to write special input/output functions for each device (keyboard, disk, etc.). The program "sees" input/output as a continuous stream of bytes no matter where the input is coming from or going to.

Topic 7.1.2: What Is a Stream?

Files
Every C stream is connected to a file. In this context, the term *file* does not refer to a disk file. Rather, it is an intermediate step between the stream that your program deals with and the actual physical device being used for input or output. For the most part,

the beginning C programmer doesn't need to be concerned with these files because the details of interactions between streams, files, and devices are taken care of automatically by the C library functions and the operating system.

Topic 7.1.3: Text Versus Binary Streams

Text Streams
C streams fall into two modes: text and binary. A *text* stream consists only of characters, such as text data being sent to the screen. Text streams are organized into lines, which can be up to 255 characters long and are terminated by an end-of-line, or newline, character. Certain characters in a text stream are recognized as having special meaning, such as the newline character. This unit deals with text streams.

Binary Streams
A *binary* stream can handle any sort of data including, but not limited to, text data. Bytes of data in a binary stream are not translated or interpreted in any special way; they are read and written exactly as is. Binary streams are used primarily with disk files, which are covered on Day 16, "Using Disk Files."

Topic 7.1.4: The Predefined Streams

Standard Input/Output Files
ANSI C has three predefined streams (stdin, stdout and stderr) and MS-DOS compilers support two additional streams (stdprn and stdaux). Together these are referred to as the *standard input/output files*. These streams are automatically opened when a C program starts executing, and are closed when the program terminates. The programmer doesn't need to take any special action to make these streams available. The standard streams and the devices they are connected with are listed here. All five of the standard streams are text-mode streams.

stdout and stdin
Whenever you have used the printf() or puts() functions to display text on the screen, you have used the stdout stream. Likewise, when you use gets() or scanf() to read keyboard input, you use the stdin stream. The standard streams are automatically opened, but other streams, such as those used to manipulate information stored on disk, must be opened explicitly. You learn how to do this on Day 16, "Using Disk Files." The remainder of this unit deals with the standard streams.

Table 14.1: The Five Standard Streams		

Name	Streams	Device
stdin	Standard input	Keyboard
stdout	Standard output	Screen
stderr	Standard error	Screen
stdprn (MS-DOS only)	Standard printer	Printer (LPT1:)
stdaux (MS-DOS only)	Standard auxiliary	Serial port (COM1:)

Topic 7.2: C's Stream Functions

Stream Functions
The C standard library has a variety of functions that deal with stream input and output. Most of these functions come in two varieties: one that always uses one of the standard streams, and the other that requires the programmer to specify the stream. These functions are listed in Table 14.2. This table does not list all of C's input/output functions, nor are all of the functions in the table covered in this unit.

Required Header Files
The function perror() may require STDLIB.H. All other functions require STDIO.H; vprintf() and vfprintf() also require STDARG.H. A few other functions require CONIO.H. Click the Tip button.

DO take advantage of the five standard input/output streams that C provides.

DON'T rename or change the five standard streams unnecessarily.

DON'T try to use an input stream such as stdin for an output function such as fprintf().

Table 14.2: The Standard Library's Stream Input/Output Functions		
Version That Uses One of the Standard Streams	**Version That Requires a Stream Name**	**Action**
printf()	fprintf()	Formatted output

vprintf()	vfprintf()	Formatted output with variable argument list
puts()	fputs()	String output
putchar()	putc(),fputc()	Character output
scanf()	fscanf()	Formatted input
gets()	fgets()	String input
getchar()	getc(), fgetc()	Character input
perror()	_	String output to stderr only

Example Program

The short program in Listing 14.1 demonstrates the equivalence of streams.

Listing 14.1: The Equivalence of Streams

Code

```
 1: /* LIST1401.c: Day 14 Listing 14.1 */
 2: /* Demonstrates the equivalence of stream input */
 3: /* and output. */
 4: #include <stdio.h>
 5:
 6: int main(void) {
 7:   char buffer[81];
 8:
 9:   /* Input a line, then immediately output it. */
10:
11:   puts(gets(buffer));
12:   return 0;
13: }
```

Description

On line 11, the gets() function is used to input a line of text from the keyboard (stdin). Because gets() returns a pointer to the string, it can be used as the argument to puts(), which displays the string on the screen (stdout). When run, the program inputs a line of text from the user and then immediately displays the string on the screen.

Topic 7.3: Accepting Keyboard Input

Input Functions

Almost every C program requires some form of input from the keyboard (that is,

from stdin). Input functions are divided into a hierarchy of three levels: character input, line input, and formatted input.

Topic 7.3.1: Character Input

Character Input Functions

The character input functions read input from a stream one character at a time. When called, each of these functions returns the next character in the stream, or EOF if the end of the file has been reached or an error has occurred. EOF is a symbolic constant defined in stdio.h as –1. Character input functions differ in terms of buffering and echoing.

Buffering

Some character input functions are *buffered*. This means that the operating system holds all characters in a temporary storage space until you press Enter, and then the system sends the characters to the stdin stream. Others are *unbuffered*, and each character is sent to stdin as soon as the key is pressed.

Echoing

Some input functions automatically *echo* each character to stdout as it is received. Others do not echo; the character is sent to stdin and not stdout. Because stdout is assigned to the screen, that's where input is echoed.

The uses of buffered, unbuffered, echoing, and non-echoing character input are explained in sections to follow.

Description

The function getchar() obtains the next character from the stream stdin. It provides buffered character input with echo, and its prototype is

```
int getchar(void);
```

Usage

The use of getchar() is demonstrated in Listing 14.2. Notice that the putchar() function, explained in detail later in this unit, simply displays a single character on the screen.

The getchar() function can be used to input entire lines of text, as shown in Listing 14.3. Other input functions are better suited for this task, however, as you learn later in the unit.

Listing 14.2. Demonstration of the getchar() Function

Code

```
 1: /* LIST1402.c: Day 14 Listing 14.2 */
 2: /* Demonstrates the getchar() function. */
 3:
 4: #include <stdio.h>
 5:
 6: int main(void) {
 7:   int ch;
 8:
 9:   while ((ch = getchar()) != '\n')
10:     putchar(ch);
11:   return 0;
12: }
```

Description

Here's what happens when this program executes:

1. On line 9, the getchar() function is called, and waits to receive a character from stdin. Because getchar() is a buffered input function, no characters are received until you press Enter. However, each key you press is echoed on the screen.

2. When you press Enter, all of the characters you entered, including the newline, are sent to stdin by the operating system. The getchar() function returns the characters one at a time, assigning each in turn to ch.

3. Each character is compared with the newline character '\n' and, if not equal, displayed on the screen with putchar(). When a newline is returned by getchar(), the while loop terminates.

Listing 14.3: Using the getchar() Function to Input an Entire Line of Text

Code

```
 1: /* LIST1403.c: Day 14 Listing 14.3 */
 2: /* Using getchar() to input strings. */
 3:
 4: #include <stdio.h>
 5:
 6: #define MAX 80
 7:
 8: int main(void) {
 9:   char ch, buffer[MAX+1];
10:   int x = 0;
11:
12:   while ((ch = getchar()) != '\n' && x < MAX)
13:     buffer[x++] = ch;
```

```
14:
15:   buffer[x] = '\0';
16:
17:   printf("%s", buffer);
18:   return 0;
19: }
```

Description	This program is similar to Listing 14.2 in the way that it uses getchar(). An extra condition has been added to the loop. This time the while loop accepts characters from getchar() until either a newline character is reached or 80 characters are read. Each character is assigned to an array called buffer. After the characters are inputted (MAX is reached), line 15 puts a null on the end of the array so the printf() function on line 17 can print the entered string. One additional note on this program. On line 9, why was buffer declared with a size of MAX + 1 instead of just MAX? By declaring buffer with the size of MAX + 1, the string can be 80 characters plus a null terminator. Don't forget to include a place for the null terminator at the end of your strings.

Description
The getch() function obtains the next character from the stream stdin. It provides unbuffered character input without echo.

Syntax
The prototype for getch() is in the header file conio.h, as follows:

```
int getch(void);
```

Return Value
Because it is unbuffered, getch() returns each character as soon as the key is pressed, without waiting for the user to press Enter. Because getch() does not echo its input, the characters are not displayed on-screen.

Example Programs
The use of getch() is illustrated by the program in Listing 14.4. Using getch() to input an entire line of text is illustrated in Listing 14.5.

Non-ANSI
Caution: getch() is not an ANSI-standard command. This means that your compiler

(and other compilers) might, or might not, support it. getch() is supported by Symantec, Borland, and Microsoft. If you have problems using it, you should check your compiler and see whether it supports getch(). If you are concerned with portability, you should avoid non-ANSI functions.

Listing 14.4: Using the getch() Function

```
Code  1: /* LIST1404.c: Day 14 Listing 14.4 */
      2: /* Demonstrates the getch() function. */
      3: /* MS-DOS Application Only */
      4: /* Non-ANSI Application */
      5:
      6: #include <stdio.h>
      7: #include <conio.h>
      8:
      9: int main(void) {
     10:   int ch;
     11:
     12:   while ((ch = getch()) != '\r')
     13:     putchar(ch);
     14:   return 0;
     15: }
```

Description

When this program runs, getch() returns each character as soon as you press a key—it doesn't wait for you to press Enter. There's no echo, so the only reason that each character is displayed on-screen is the call to putchar(). But why does the program compare each character to \r instead of to \n?

The code \r is the escape sequence for the carriage return character. When you press Enter, the keyboard device sends a carriage return (CR) to stdin. The buffered character input functions automatically translate the carriage return to a newline, so the program must test for \n to determine whether Enter has been pressed. The unbuffered character input functions do not translate, so a carriage return is input as \r and that's what the program must test for.

Listing 14.5: Using the getch() Function to Input an Entire Line

```
Code  1: /* LIST1405.c: Day 14 Listing 14.5 */
      2: /* Using getch() to input strings. */
      3: /* MS-DOS Application Only */
      4: /* Non-ANSI Application */
      5:
      6: #include <stdio.h>
```

```
 7: #include <conio.h>
 8:
 9: #define MAX 80
10:
11: int main(void) {
12:   char ch, buffer[MAX+1];
13:   int x = 0;
14:
15:   while ((ch = getch()) != '\r' && x < MAX)
16:     buffer[x++] = ch;
17:
18:   buffer[x] = '\0';
19:
20:   printf("%s", buffer);
21:   return 0;
22: }
```

Description	Running this program illustrates clearly that getch() does not echo its input. With the exception of getch() substituted for getchar(), this program is identical to Listing 14.3.

Description
The getche() function is unbuffered like getch(), but it echoes each character to stdout, like getchar().

Example Program
Modify the program in Listing 14.4 to use getche() instead of getch(). When the program runs, each key you press is displayed twice on the screen: once as echoed by getche() and once by putchar().

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

Description
The getc() and the fgetc() functions are also character input functions but they do not automatically work with stdin. Rather, they let the program specify the input stream. They are used primarily to read characters from disk files, which Day 16, "Using Disk Files," covers in detail. Click the Tip button.

DO understand the difference between echoed and non-echoed input.

DO understand the difference between buffered and unbuffered input

DON'T use non-ANSI-standard functions if portability is a concern.

Regular Versus Special Keys

So far, you've learned how to input all the regular characters of the IBM PC keyboard: letters, numerals, and punctuation marks. The PC keyboard also has a number of special keys, such as the function keys F1–F10, as well as the arrow and other direction keys on the numeric keypad. You also can type key combinations, such as Ctrl and PgDn; Alt and 1; and Shift and F1. How do you accept input from these special keys in your C programs?

Extended Keys

To do this, you first need to know how these keys and key combinations work. These keys differ from the regular character keys in that they send a pair of values to stdin rather than just one. In each pair, the first value is always the null character, with a numeric value of 0. The second has a numeric value that identifies the key pressed. For example, pressing F1 sends a pair consisting of 0 followed by 59, and pressing the Home key sends a pair consisting of 0 followed by 71. The keys that return a two-character code to stdin are called the *extended keys,* and their codes are called the *extended key codes*.

Extended Key Input

How does a program deal with extended key input? Specifically, how can a program accept only the press of a special key, ignoring other characters? It's simple.

1. Accept characters from stdin, discarding them until a 0, or \0 is received. (To the C compiler, the number 0 and the character \0 are the same.) This signals the first of two values sent by one of these keys.
2. Look at the next value that identifies the key that was pressed.

Example Program

A function that performs this task, ext_key(), is presented in Listing 14.6. The program uses ext_key() to accept extended key presses, ignoring all other input. The second value of the key is displayed on-screen; pressing F1 exits the program. The second value sent by the F1 key is 59, so that's the value the program tests for in its if statement.

Listing 14.6: Accepting Extended Key Input

```
Code  1: /* LIST1406.c: Day 14 Listing 14.6 */
```

```
 2: /* Demonstrates reading extended keys from the */
 3: /* keyboard. */
 4: /* MS-DOS Application Only */
 5: /* Non-ANSI Application */
 6:
 7: #include <stdio.h>
 8: #include <conio.h>
 9:
10: int ext_key(void);
11:
12: int main(void) {
13:   int ch;
14:
15:   puts("Press any extended key; "
16:        "press F1 to exit.");
17:
18:   while (1) {
19:     ch = ext_key();
20:     if (ch == 59)    /* F1? */
21:       break;
22:     else
23:       printf("\nThat key's code has a value of "
24:              "%d.", ch);
25:   }
26:   return 0;
27: }
28:
29: int ext_key(void) {
30:   int ch;
31:
32:   /* Wait until a zero byte comes in. */
33:
34:   while ((ch = getch()) != 0)
35:     ;
36:
37:   /* Return the next character. */
38:
39:   return getch();
40: }
```

Description

As long as characters are entered into the program, main() stays in an infinite while loop. In the loop line 19 makes a call to ext_key(), which is defined on lines 29 – 40. ext_key() has its own while loop that does nothing but look for the character \0, the signal that an extended character has been pressed. Once it obtains the null character, ext_key() sends back the next character by calling getch() again.

Back in main(), line 20 evaluates the next value to see whether it equals 59 (equivalent to F1). If it does, a break exits the

	infinite loop and the program ends. If it was not F1, the program prints the value of the extended key and the loop continues.

CONIO.H

Most compilers make dealing with extended key input easier by defining a set of symbolic constants in the header file. This file is usually CONIO.H. Click here for a table of defined symbolic constants. Then click the Example link.

Example

Creating a Header File with the Defined Constants

You should check your compiler manual to determine if it contains defined constants for the extended keys. If it does not, add the #define statement(s) to the beginning of your program for the specific keys that your program needs. If your compiler does not define the extended keys for you, you will want to create a header file with all of the above defined constants. This way, you can include the file in any program that needs the extended key codes. (On Day 21, "Taking Advantage of Preprocessor Directives and More," you learn specifics about header files.)

Table 14.3: A Subset of the Symbolic Constants for Extended Keys Defined in conio.h of the Symantec Compiler

Constant	Value	Key
`#define _KB_F1`	59	/* Function key F1 */
`#define _KB_HOME`	71	/* Editing key */
`#define _KB_SF1`	84	/* Shift F1 */
`#define _KB_CF1`	94	/* Control F1 */
`#define _KB_AF1`	104	/* Alt F1 */
`#define _KB_CPGUP`	132	/* Control PgUp */
`#define _KB_A1`	120	/* Alt 1 */
`#define _KB_A0`	129	/* Alt 0 */
`#define _KB_AMINUS`	130	/* Alt '–'. */
`#define _KB_APLUS`	131	/* Alt '+'. */

In the Symantec compiler, each of these constants starts with the characters _KB_, and identifies one of the extended keys mnemonically. Its value is equal to the value returned by the key. For example, the constant _KB_F1 is defined as the value 59. Some of these constants and their values are shown in Table 14.3.

Example Program Modified

The function ext_key() can be modified to accept a subset of the extended keys. You can use extended keys to create flexible and user-friendly interfaces, menus, and the like. The program in Listing 14.7 modifies ext_key() to accept function key presses only, and then implements a function-key-driven menu system. Remember that you need to add #DEFINE statements for the constants if your compiler doesn't have them.

Listing 14.7: A Menu System that Responds to Function Key Input

```
Code  1: /* LIST1407.c: Day 14 Listing 14.7 */
      2: /* Demonstrates a function key driven menu */
      3: /* using symbolic constants for extended keys */
      4: /* defined in conio.h of the Symantec compiler. */
      5: /* MS-DOS Application Only */
      6: /* Non-ANSI Application */
      7:
      8: #include <stdio.h>
      9: #include <stdlib.h>
     10: #include <conio.h>
     11: #include <time.h>
     12:
     13: int fnc_key(void);
     14: int menu(void);
     15:
     16: int main(void) {
     17:   /* Set up infinite loop. */
     18:
     19:   while (1) {
     20:     /* Switch based on return value of menu(). */
     21:
     22:     switch (menu()) {
     23:       case _KB_F1:
     24:         puts("Task 1");
     25:         break;
     26:       case _KB_F2:
     27:         puts("Task 2");
     28:         break;
     29:       case _KB_F3:
     30:         puts("Task 3");
     31:         break;
```

```
32:       case _KB_F4:
33:         puts("Task 4");
34:         break;
35:       case _KB_F5:
36:         puts("Task 5");
37:         break;
38:       case _KB_F6:
39:         puts("Task 6");
40:         break;
41:       case _KB_F7:
42:         puts("Task 7");
43:         break;
44:       case _KB_F8:
45:         puts("Task 8");
46:         break;
47:       case _KB_F9:
48:         puts("Task 9");
49:         break;
50:       case _KB_F10:
51:         puts("Exiting program...");
52:         exit(0);
53:     }
54:   }
55:   return 0;
56: }
57:
58: int menu(void) {
59:   /* Display menu choices. */
60:
61:   puts("\nF1 -> task 1");
62:   puts("F2 -> task 2");
63:   puts("F3 -> task 3");
64:   puts("F4 -> task 4");
65:   puts("F5 -> task 5");
66:   puts("F6 -> task 6");
67:   puts("F7 -> task 7");
68:   puts("F8 -> task 8");
69:   puts("F9 -> task 9");
70:   puts("F10 -> exit\n");
71:
72:   /* Get a function key press. */
73:
74:   return (fnc_key());
75: }
76:
77: int fnc_key(void) {
78:   int ch;
79:
80:   while (1) {
81:     /* Wait until a zero byte comes in. */
82:
83:     while ((ch = getch()) != 0)
```

```
     84:       ;
     85:     /* Get the next character. */
     86:
     87:     ch = getch();
     88:
     89:     /* Is it a function key? */
     90:
     91:     if (ch >= _KB_F1 && ch <= _KB_F10)
     92:       return ch;
     93:   }
     94: }
```

Output

```
F1 -> task 1
F2 -> task 2
F3 -> task 3
F4 -> task 4
F5 -> task 5
F6 -> task 6
F7 -> task 7
F8 -> task 8
F9 -> task 9
F10 -> exit

Exiting program...
```

Description This program modifies ext_key() to accept function key presses only, and then implements a function-key-driven menu system.

"Ungetting" a Character

What does "ungetting" a character mean? An example should explain. Click the Example link.

Example

ungetc() Syntax

To "unget" a character, you use the ungetc() library function. Its prototype is

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

Say, for instance, your program is reading characters from an input stream and can detect the end of input only by reading one character too many. For example, you may be inputting digits only, so you know that input has ended when the first non-digit character is encountered. That first non-digit character may be an important part of subsequent data, but it has been removed from the input stream. Is it lost? No, it can be "ungotten" or returned to the input stream where it is then the first character

read by the next input operation on that stream.

Arguments

The argument `c` is the character to be returned. The argument `*file-ptr` specifies the stream that the character is to be returned to, which can be any input stream. For now, simply specify stdin as the second argument: `ungetc(c, stdin);`. The notation `FILE *file-ptr` is used with streams associated with disk files; you learn about this on Day 16, "Using Disk Files."

Restriction

You can "unget" only a single character to a stream between reads, and you cannot "unget" EOF at any time.

Return Value

The function ungetc() returns `c` on success and EOF if the character cannot be returned to the stream.

A Line-Input Function

The line-input functions read a line from an input stream—they read all characters up to the next newline character. The standard library has two line input functions, gets() and fgets(). You were introduced to the gets() function on Day 10, "Characters and Strings." This is a straightforward function, reading a line from stdin and storing it in a string.

Syntax

The function prototype is

```
char *gets(char *s);
```

Description

gets() takes a pointer to type char as its argument, and returns a pointer to type char. The gets() function reads characters from stdin until a newline (`\n`) or end-of-file is encountered; the newline is replaced with a null character, and the string is stored at the location indicated by `s`.

Return Value

The return value is a pointer to the string (the same as `s`). If gets() encounters an error or reads end-of-file before any characters are input, a null pointer is returned.

Allocating Memory

Before calling gets(), you must allocate sufficient memory space to store the string, using the methods covered on Day 10, "Characters and Strings." The function has no

way of knowing whether space for ptr is allocated or not; the string is input and stored starting at ptr in either case. If the space has not been allocated, the string may overwrite other data and cause program errors.

Description

The fgets() library function is similar to gets() in that it reads a line of text from an input stream. It is more flexible because it allows the programmer to specify the specific input stream to use and the maximum number of characters to be input. The fgets() function is often used to input text from disk files, covered on Day 16, "Using Disk Files." To use it for input from stdin, you specify stdin as the input stream.

Syntax

The prototype of fgets() is

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

Arguments

The last parameter `FILE *file-ptr` is used to specify the input stream. For now, simply specify `stdin` as the stream argument.

The pointer `s` indicates where the input string is stored. The argument `num` specifies the maximum number of characters to be input. The fgets() function reads characters from the input stream until a newline or end-of-line is encountered, or n - 1 characters have been read. The newline is included in the string, terminated with a `\0` before it is stored.

Return Value

The return values of fgets() are the same as described earlier for gets().

The First n - 1 Characters

Strictly speaking, fgets() does not input a single line of text (if you define a line as a sequence of characters ending with a newline). It can read less than a full line if the line contains more than n - 1 characters. When used with stdin, execution does not return from fgets() until you press Enter, but only the first n - 1 characters are stored in the string. The newline is included in the string only if it falls within the first n - 1 characters.

Example Program

The program in Listing 14.8 demonstrates fgets().

Listing 14.8: Using the fgets() Function for Keyboard Input

```
Code  1:  /* LIST1408.c: Day 14 Listing 14.8 */
```

```
 2: /* Demonstrates the fgets() function. */
 3:
 4: #include <stdio.h>
 5:
 6: #define MAXLEN 10
 7:
 8: int main(void) {
 9:   char buffer[MAXLEN];
10:
11:   puts("Enter text a line at a time; "
12:        "enter a blank to exit.");
13:
14:   while (1) {
15:     fgets(buffer, MAXLEN, stdin);
16:
17:     if (buffer[0] == '\n')
18:       break;
19:
20:     puts(buffer);
21:   }
22:   return 0;
23: }
```

Output

```
Enter text a line at a time; enter a blank to
exit.
Roses are red
Roses are
red
Violets are blue
Violets a
re blue

Programming in C
Programmi
ng in C

Is for people like you!
Is for pe
ople like
you!
```

Description

When running the program, enter lines of length less than and greater than MAXLEN to see what happens. If a line greater than MAXLEN is entered, the first MAXLEN - 1 characters are read by the first call to fgets(); the remaining characters remain in the keyboard buffer and are read by the next call to fgets().

Topic 7.3.2: Formatted Input

scanf() and fscanf() Functions
The input functions covered until now have simply taken one or more characters from an input stream and put them somewhere in memory. No interpretation or formatting of the input has been done, and you still have no method of assigning multiple character input to numeric variables. How, for example, would you input the value 12.86 from the keyboard and assign it to a type float variable? Enter the scanf() and fscanf() functions. You were introduced to scanf() on Day 7, "Basic Input/Output"; this section explains its use in more detail.

scanf() Versus fscanf()
These two functions are identical except that scanf() always uses stdin, whereas the user can specify the input stream in fscanf(). This section covers scanf(); fscanf() generally is used with disk file input and is covered on Day 16, "Using Disk Files."

A Variable Number of Arguments
The scanf() function takes a variable number of arguments; it requires a minimum of two. The first argument is a format string that uses special characters to tell scanf() how to interpret the input. The second and additional arguments are the addresses of the variable(s) to which the input data is assigned. Click the Example link.

Example

Here's an example:

```
scanf("%d", &x);
```

The first argument `"%d"` is the format string. In this case, `%d` tells scanf() to look for one signed integer value. The second argument uses the address of operator (&) to tell scanf() to assign the input value to the variable x. Now you can look at the format string details.

The Format String
The scanf() format string can contain

- Spaces and tabs, which are ignored (they can be used to make the format string more readable).
- Characters (but not %), which are matched against non-white characters in the input.

• One or more *conversion specifications* (also known as format specifications), which consist of the % character followed by special characters. Generally, the format string contains one conversion specification for each variable.

Conversion Specifications

The only required part of the format string is the conversion specifications. Each conversion specification begins with the % character and contains optional and required components in a certain order. The scanf() function applies the conversion specifications in the format string, in order, to the input fields. An input field is a sequence of non-whitespace characters that ends when the next white space is encountered or when the field width, if specified, is reached.

Conversion Specification Components

The conversion specification components are

Component	Optional/Required	Description
Assignment suppression flag (*)	Optional	Immediately follows the %. If present, this character tells scanf() to perform the conversion corresponding to the current conversion specifier (also known as a format specifier), but to ignore the result (not assign it to any variable).
Field width	Optional	Is a decimal number specifying the width, in characters, of the input field. In other words, the field width specifies how many characters from stdin scanf() should examine for the current conversion. If a field width is not specified, the input field extends to the next white space.
Precision modifier	Optional	A single character that can be `h`, `l`, or `L`. If present, the precision modifier changes the meaning of the following type specifier. Details are given later in the unit.
Type specifier	Required	Is one or more characters that tell scanf() how to interpret the input. The characters are listed and explained here. The Argument column lists the required type

		of the corresponding variable; for example, the type specifier `d` requires int * (a pointer to type int).

Table 14.4: The Type Specifier Characters Used in Conversion Specifiers

Type	Meaning	Argument
`d`	A decimal integer	int *
`i`	An integer in decimal, octal (with leading 0) or hexadecimal (with leading 0X or 0x) notation.	int *
`o`	An integer in octal notation with or without the leading 0.	int *
`u`	An unsigned decimal integer.	unsigned int *
`x`	A hexadecimal integer with or without the leading 0X or 0x.	int *
`c`	One or more characters are read and assigned sequentially to the memory location indicated by the argument. No terminating \0 is added. If a field width argument is not given, 1 character is read. If a field width argument is given, that number of characters, including whitespace (if any), is read.	char *
`s`	A string of non-whitespace characters is read into the specified memory location, and a terminating \0 is added.	char *
`e,f,g`	A floating point number. Numbers can be input in decimal or scientific notation.	float*
`[...]`	A string. Only the characters listed between the brackets are accepted. Input ends as soon as a non-matching character is encountered, the specified field width is reached, or Enter is pressed. To accept the] character, list it first: [ ]...]. A \0 is added at the end of the string.	char *
`[^...]`	The same as [...] except that only characters not listed between the brackets are accepted.	char *
`%`	Literal %: reads the % character. No assignment is	none

	made.	

Precision Modifiers
Before seeing some examples of scanf(), you need to understand the precision modifiers. The precision modifiers are

Modifier	Description
`h`	When placed before the type specifier d, i, o, u, or x, the modifier h specifies that the argument is a pointer to type short instead of type int.
`l`	When placed before the type specifier d, i, o, u, or x, the modifier l specifies that the argument is a pointer to type long. When placed before the type specifier e, f, or g, the modifier l specifies that the argument is a pointer to type double.
`L`	When placed before the type specifier e, f, or g, the modifier L specifies that the argument is a pointer to type long double.

Overview of the Operation of scanf()
Input from scanf() is buffered; no characters are actually received from stdin until the user presses Enter. The entire line of characters then "arrives" from stdin, and is processed, in order, by scanf(). Execution returns from scanf() only when enough input has been received to match the specifications in the format string. Also, scanf() processes only enough characters from stdin to satisfy its format string. Extra, unneeded characters, if any, remain "waiting" in stdin. These characters can cause problems. Take a closer look at the operation of scanf() in order to see how.

Three Situations
When a call to scanf() is executed and the user has entered a single line, you can have three situations. For these examples, assume that `scanf("%d %d", &x, &y);` is being executed; in other words, scanf() is expecting two decimal integers. Click the Example link.

Example

The Third Situation: Potential for Problems
It's this third situation (specifically, those leftover characters) that can cause problems. They remain waiting for as long as your program is running, until the next time the program reads input from stdin. Then the leftover characters are the first ones read,

ahead of any input the user makes at the time. It's clear how this could cause errors! Click the Example link.

Example

The possibilities are

1. The line the user inputs matches the format string. For example, the user enters 12 14 followed by Enter. In this case, there are no problems; scanf() is satisfied, and there are no characters left over in stdin.
2. The line that the user inputs has too few elements to match the format string. For example, the user enters 12 followed by Enter. In this case, scanf() continues to wait for the missing input. Once the input is received, execution continues and there are no characters left over in stdin.
3. The line that the user enters has more elements than required by the format string. For example, the user enters 12 14 16 followed by Enter. In this case, scanf() reads the 12 and the 14, and then returns. The extra characters, the 1 and the 6, are left "waiting" in stdin.

The following code asks the user to input an integer, and then to input a string:

```
puts("Enter your age.");
scanf("%d", &age);
puts("Enter your first name.");
scanf("%s", name);
```

Say, for example, that in response to the first prompt, the user decides to be precise and enters 29.00, and then presses Enter. The first call to scanf() is looking for an integer, so it reads the characters 29 from stdin and assigns the value 29 to the variable age. The characters .00 are left waiting in stdin. The next call to scanf() is looking for a string. It goes to stdin for input and finds .00 waiting there. The result is that the string .00 is assigned to name.

Avoiding the Problem

How can you avoid this problem of leftover characters? If the people who use your programs never make mistakes, that's one solution—but rather impractical!

A better solution is to make sure there are no extra characters waiting in stdin before

prompting the user for input. You can do this by calling gets(), which reads any remaining characters from stdin, up to and including the end of the line. Rather than calling gets() directly from the program, you can put it in a separate function with the descriptive name clear_kb(). This function is shown in Listing 14.9.

Listing 14.9: Removing Extra Characters from stdin to Avoid Errors

Code

```
 1: /* LIST1409.c: Day 14 Listing 14.9 */
 2: /* Removing extra characters from stdin. */
 3: /* MS-DOS Application Only */
 4: /* Non-ANSI Application */
 5:
 6: #include <stdio.h>
 7:
 8: void clear_kb(void);
 9:
10: int main(void) {
11:   int age;
12:   char name[20];
13:
14:   /* Prompt for user's age. */
15:
16:   puts("Enter your age.");
17:   scanf("%d", &age);
18:
19:   /* Clear stdin of any extra characters. */
20:
21:   clear_kb();
22:
23:   /* Now prompt for user's name. */
24:
25:   puts("Enter your first name.");
26:   scanf("%s", name);
27:   /* Display the data. */
28:
29:   printf("Your age is %d.\n", age);
30:   printf("Your name is %s.\n", name);
31:   return 0;
32: }
33:
34: void clear_kb(void) {
35: /* Clears stdin of any waiting characters. */
36:   char junk[80];
37:   gets(junk);
38: }
```

Output

```
Enter your age.
29
Enter your first name.
Bradley Jones
Your age is 29.
```

	`Your name is Bradley.`
Description	When you run the program in Listing 14.9, enter some extra characters after your age, before pressing Enter. Make sure the program ignores them and correctly prompts you for your name. Then modify the program by removing the call to clear_kb(), and run it again. Any extra characters entered on the same line as your age are assigned to name.

Example Program

The best way to become familiar with the operation of the scanf() function is to use it. It's a powerful function but can be a bit confusing at times. Try it and see what happens. The program in Listing 14.10 demonstrates some of the unusual ways to use scanf(). You should compile and run the program, and then do some experimentation by making changes in the scanf() format strings.

User-Defined Input Functions

The scanf() function can be used for most of your input needs, particularly those involving numbers (strings are input more easily with gets()). It is often worthwhile, however, to write your own specialized input functions. You can see some examples of user-defined functions on Day 18, "Getting More from Functions."

DO take advantage of extended characters in your programs. When using extended characters, you should try to be consistent with other programs.

DON'T forget to check the input stream for extra characters.

DO use the gets() and scanf() functions instead of the fgets() and fscanf() functions if you are using the standard input file (stdin) only.

Listing 14.10: Some Ways to Use scanf() for Keyboard Input

```
Code  1: /* LIST1410.c: Day 14 Listing 14.10 */
      2: /* Demonstrates some uses of scanf(). */
      3:
      4: #include <stdio.h>
      5:
      6: void clear_kb(void);
      7:
      8: int main(void) {
      9:   int i1, i2;
     10:   long l1;
```

```
11:   double d1;
12:   char buf1[80], buf2[80];
13:
14:   /* Using the l modifier to enter long  */
15:   /* integers and doubles */
16:
17:   puts("Enter an integer and a floating point "
18:        "number.");
19:   scanf("%ld %lf", &l1, &d1);
20:   printf("You entered %ld and %lf.\n", l1, d1);
21:   puts("The scanf() format string used the l "
22:        "modifier to ");
23:   puts("store your input in a type long and a "
24:        "type double.\n");
25:
26:   clear_kb();
27:
28:   /* Use field width to split input. */
29:
30:   puts("Enter a 5 digit integer (e.g., 54321).");
31:   scanf("%2d%3d", &i1, &i2);
32:
33:   printf("You entered %d and %d.\n", i1, i2);
34:   puts("Note how the field width specifier in ");
35:   puts("the scanf() format string split your ");
36:   puts("input into values.\n");
37:
38:   clear_kb();
39:
40:   /* Using an excluded space to split a line of */
41:   /* input into two strings at the space. */
42:
43:   puts("Enter your first and last names separated "
44:        "by a space. ");
45:   scanf("%[^ ]%s", buf1, buf2);
46:   printf("Your first name is %s\n", buf1);
47:   printf("Your last name is %s\n", buf2);
48:   puts("Note how [^ ] in the scanf() format ");
49:   puts("string, by excluding the space ");
50:   puts("character, caused the input to be split.");
51:   return 0;
52: }
53:
54: void clear_kb(void) {
55: /* Clears stdin of any waiting characters. */
56:   char junk[80];
57:   gets(junk);
58: }
```

Output

```
Enter an integer and a floating point number.
123 45.6789
You entered 123 and 45.678900.
```

```
The scanf() format string used the l modifier to
store your input in a type long and a type double.

Enter a 5 digit integer (e.g., 54321).
54321
You entered 54 and 321.
Note how the field width specifier in
the scanf() format string split your
input into values.

Enter your first and last names separated by a
space.
Peter Aitken
Your first name is Peter
Your last name is Aitken
Note how [^ ] in the scanf() format
string, by excluding the space
character, caused the input to be split.
```

Description

This listing starts by defining several variables on lines 9 – 12 for data input. The program then walks you through the steps of entering various types of data. Lines 14 – 24 have you enter and print long integers and a double.

Line 26 calls the clear_kb() function to clear any unwanted characters from the input stream.

Lines 30 – 31 get the next value, a five-character integer. Because there are width specifiers, the five-digit integer is split into two integers, one that is two characters and one that is three characters.

Line 38 calls clear_kb() to clear the keyboard again.

The final example, lines 40 – 50, use the exclude character. Line 45 uses `"%[^ ]"` which tells scanf() to get a string but to stop at any spaces. This effectively splits the input.

You should take time modifying this listing and entering additional values to see what the results are.

Topic 7.4: Screen Output

Screen Output Functions

Screen output functions are divided into three general categories along the same lines as the input functions: character output, line output, and formatted output. You've been introduced to some of these functions in earlier units. This section covers them all in detail.

Topic 7.4.1: Character Output with putchar(), putc(), and fputc()

Character Output Functions

The C library's character output functions send a single character to a stream. The function putchar() sends its output to stdout (normally the screen). The functions fputc() and putc() send their output to a stream specified in the argument list.

Syntax

The prototype for putchar is located in STDIO.H and reads

```
int putchar(int c);
```

The function writes the character stored in c to stdout. Although the prototype specifies a type int argument, you pass putchar() a type char. You also can pass it a type int as long as its value is appropriate for a character (that is, in the range 0–255).

Return Value

The function returns the character that was just written, or EOF if an error has occurred.

Example Programs

You saw putchar() demonstrated in Listing 14.2. The program in Listing 14.11 displays the characters with ASCII values between 14 and 127.

You also can display strings with the putchar() function as in Listing 14.12, although other functions are better suited for this purpose.

Listing 14.2. Demonstration of the getchar() Function

```
Code  1: /* LIST1402.c: Day 14 Listing 14.2 */
      2: /* Demonstrates the getchar() function. */
      3:
      4: #include <stdio.h>
      5:
      6: int main(void) {
      7:   int ch;
```

```
 8:
 9:   while ((ch = getchar()) != '\n')
10:     putchar(ch);
11:   return 0;
12: }
```

Description

Here's what happens when this program executes:

1. On line 9, the getchar() function is called, and waits to receive a character from stdin. Because getchar() is a buffered input function, no characters are received until you press Enter. However, each key you press is echoed on the screen.

2. When you press Enter, all of the characters you entered, including the newline, are sent to stdin by the operating system. The getchar() function returns the characters one at a time, assigning each in turn to ch.

3. Each character is compared with the newline character '\n' and, if not equal, displayed on the screen with putchar(). When a newline is returned by getchar(), the while loop terminates.

Listing 14.11: The putchar() Function

```
Code  1: /* LIST1411.c: Day 14 Listing 14.11 */
      2: /* Demonstrates putchar(). */
      3:
      4: #include <stdio.h>
      5:
      6: int main(void) {
      7:   int count;
      8:
      9:   for (count = 14; count < 128;)
     10:     putchar(count++);
     11:   return 0;
     12: }
```

Description

This program displays the characters with ASCII values between 14 and 127.

Listing 14.12: Displaying a String with putchar()

```
Code  1: /* LIST1412.c: Day 14 Listing 14.12 */
      2: /* Using putchar() to display strings. */
      3:
```

```
#include <stdio.h>

#define MAXSTRING 80

char message[] = "Displayed with putchar().";
int main(void) {
  int count;

  for (count = 0; count < MAXSTRING; count++) {

    /* Look for the —d of the string. When */
    /* it's found, write a newline character */
    /* and exit the loop. */

    if(message[count] == '\0') {
      putchar('\n');
      break;
    }
    else

    /* If end of string not found, write the */
    /* next character. */

       putchar(message[count]);
  }
  return 0;
}
```

Output	`Displayed with putchar().`
Description	This program displays strings with the putchar() function.

Description

The putc() and fputc() functions perform the same action, sending a single character to a specified stream. putc() is a macro implementation of fputc(). You learn about macros on Day 21, "Taking Advantage of Preprocessor Directives and More"; for now, just stick to fputc().

Syntax of fputc()

Its prototype is

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

Arguments

The `FILE *file-ptr` part of the prototype might puzzle you. You pass fputc() the output stream in this argument. You learn more about this on Day 16, "Using Disk Files." If you specify `stdout` as the stream, fputc() behaves exactly the same as

putchar(). Thus, the following two statements are equivalent:

```
putchar('x');
fputc('x', stdout);
```

Topic 7.4.2: Using puts() and fputs() for String Output

Description

Your programs display strings on the screen more often than they display single characters. The library function puts() displays strings. The function fputs() sends a string to a specified stream; otherwise, it is identical to puts().

Syntax

The prototype for puts() is

```
int puts(char *s);
```

with `*s` as a pointer to the first character of the string that you want displayed. The puts() function displays the entire string up to but not including the terminating null character, adding a newline at the end.

Return Value

Then puts() returns a positive value if successful, EOF on error (remember, EOF is a symbolic constant with the value –1; it is defined in STDIO.H).

Example Program

The puts() function can be used to display any type of string. Its use is demonstrated in Listing 14.13.

Listing 14.13: Using the puts() Function to Display Strings

```
Code  1: /* LIST1413.c: Day 14 Listing 14.13 */
      2: /* Demonstrates puts(). */
      3:
      4: #include <stdio.h>
      5:
      6: /* Declare and initialize an array of pointers. */
      7:
      8: char *messages[5] = {"This", "is", "a", "short",
      9:                      "message." };
     10:
     11: int main(void) {
```

```
12:   int x;
13:
14:   for (x = 0; x < 5; x++)
15:     puts(messages[x]);
16:
17:   puts("And this is the end!");
18:   return 0;
19: }
```

Output

```
This
is
a
short
message.
And this is the end!
```

Description

This listing declares an array of pointers, a subject not covered yet, (but which will be covered tomorrow, Day 15, "More on Pointers"). Lines 14 and 15 print each of the strings stored in the message array.

Topic 7.4.3: Using printf() and fprintf() for Formatted Output

Description

So far, the output functions have displayed characters and strings only. What about numbers? To display numbers, you must use the C library's formatted output functions printf() and fprintf(). These functions also can display strings and characters. You were officially introduced to printf() on Day 7, "Basic Input/Output," and have used it in almost every unit; this section provides the remainder of the details.

printf() Versus fprintf()

The two functions printf() and fprintf() are identical except that printf() always sends output to stdout, whereas fprintf() specifies the output stream. fprintf() is used generally for output to disk files and is covered on Day 16, "Using Disk Files."

A Variable Number of Arguments

The printf() function takes a variable number of arguments, with a minimum of one. The first and only required argument is the format string that tells printf() how to format the output. The optional arguments are variables and expressions whose values you want to display. Take a look at a few simple examples, giving you a feel for printf(), before you really get into the nitty-gritty. Click the Examples link.

Examples

Here are three printf() examples:

- The statement `printf("Hello, world.");` displays the message `Hello, world.` on-screen. This is an example of using printf() with only one argument, the format string. In this case, the format string contains only a literal string to be displayed on the screen.
- The statement `printf("%d", i);` displays the value of the integer variable `i` on the screen. The format string contains only the format specifier `%d`, which tells printf() to display a single decimal integer. The second argument `i` is the name of the variable whose value is to be displayed.
- The statement `printf("%d plus %d equals %d.", a, b, a+b);` displays `2 plus 3 equals 5` on-screen (assuming `a` and `b` are integer variables with the values of `2` and `3`, respectively). This use of printf() has four arguments: a format string that contains literal text as well as format specifiers, two variables, and an expression whose values are to be displayed.

Format String

Now look at the printf() format string in more detail. It can contain

- One or more conversion commands that tell printf() how to display a value in its argument list. A conversion command consists of % followed by one or more characters.
- Characters that are not part of a conversion command and are displayed as is.

The third example's format string is `%d plus %d equals %d`. In this case, the three `%d`s are conversion commands and the remainder of the string, including the spaces, consists ofliteral characters that are displayed directly.

The Conversion Command

Now you can dissect the conversion command. The components of the command are given here and explained below; components in brackets are optional.

```
%[flag][field_width][.[precision]][l]conversion_char
```

The conversion_char is the only required part of a conversion command (other than

the %). The conversion characters (also known as format specifiers) and their meanings are listed here.

Table 14.5: The printf() and fprintf() Conversion Characters	
Conversion Character	**Meaning**
`d, i`	Display a signed integer in decimal notation.
`u`	Display an unsigned integer in decimal notation.
`o`	Display an integer in unsigned octal notation.
`x, X`	Display an integer in unsigned hexadecimal notation. Use x for lowercase output, X for uppercase output.
`c`	Display a single character (the argument gives the character's ASCII code).
`e, E`	Display a float or double in scientific notation (for example, 123.45 is displayed as 1.234500e+002). Six digits are displayed to the right of the decimal point unless another precision is specified (see below). Use e or E to control the case of output.
`f`	Display a float or double in decimal notation (for example, 123.45 is displayed as 123.450000). Six digits are displayed to the right of the decimal point unless another precision is specified.
`g, G`	Use e, E, or f format. The e or E format is used if the exponent is less than -3 or greater than the precision (which defaults to 6. f format is used otherwise). Trailing zeros are truncated.
`n`	Nothing is displayed. The argument corresponding to an n conversion command is a pointer to type int. The printf() function assigns to this variable the number of characters output so far.
`s`	Display a string. The argument is a pointer to char. Characters are displayed until a null character is encountered or the number of characters specified by precision (which defaults to 32767) is displayed. The terminating null character is not output.
`%`	Display the % character.

The last optional part of the printf() format string is the flag, which immediately follows the `%` character. There are four available flags:

Flag	Meaning
`-`	Means that the output is left-justified in its field rather than the default right-justified.
`+`	Means that signed numbers are always displayed with a leading `+` or `-`.
`' '` (space)	Means that positive numbers are preceded by a space.
`#`	Applies only to `x`, `X`, and `o` conversion characters. It specifies that non-zero numbers are displayed with a leading `0X` or `0x` (for `x` and `X`) or a leading `0` (for `o`).

The field-width specifier determines the minimum number of characters output. The field-width specifier can be

- A decimal integer not starting with 0. The output is padded on the left with spaces to fill the designated field width.
- A decimal integer starting with 0. The output is padded on the left with zeros to fill the designated field width.
- The `*` character. The value of the next argument (which must be an int) is used as the field width. For example, if `w` is a type int with a value of `10`, the statement `printf("%*d", w, a);` prints the value of a with a field width of `10`.

If no field width is specified, or if the specified field width is narrower than the output, the output field is just as wide as needed.

The precision specifier consists of a decimal point (`.`) by itself or followed by a

number. A precision specifier applies only to the conversion characters `e E f g G s`. It specifies the number of digits to display to the right of the decimal point or, when used with s, the number of characters to output. If the decimal point is used alone, it specifies a precision of 0.

You can place the `l` modifier just before the conversion character. This modifier applies only to the conversion characters `o, u, x, X, i, d, b`. When applied, this modifier specifies that the argument is a type long rather than a type int. If the `l` modifier is applied to the conversion characters `e, E, f, g, G`, it specifies that the argument is a type double precision number. If an `l` is placed before any other conversion characters, it is ignored.

Contents of the Format String

When you use printf(), the format string can be a string literal enclosed in double quotes in the printf() argument list. It also can be a null-terminated string stored in memory, in which case you pass a pointer to the string to printf(). Click the Example link.

Example

Escape Sequences

As explained on Day 7, "Basic Input/Output," the printf() format string can contain escape sequences that provide special control over the output. Click here for a table that lists the most frequently used escape sequences. For example, including the newline sequence (\n) in a format string causes subsequent output to appear starting on the next screen line.

Table 14.6: The Most Frequently Used Escape Sequences

Sequence	Meaning
`\a`	bell (alert)
`\b`	backspace
`\n`	newline
`\t`	horizontal tab
`\\`	backslash
`\?`	question mark
`\'`	

'	single quote
"	double quote

```
char *fmt = "The answer is %f.";
printf(fmt, x);
```

is equivalent to

```
printf("The answer is %f.", x);
```

Example Program

printf() is somewhat complicated. The best way to learn how to use it is to look at examples and then experiment on your own. The program in Listing 14.14 demonstrates many of the ways you can use printf().

Listing 14.14: Some Ways to Use the printf() Function

```
Code  1: /* LIST1414.c: Day 14 Listing 14.14 */
      2: /* Demonstration of printf(). */
      3:
      4: #include <stdio.h>
      5:
      6: char *m1 = "Binary";
      7: char *m2 = "Decimal";
      8: char *m3 = "Octal";
      9: char *m4 = "Hexadecimal";
     10:
     11: int main(void) {
     12:   float d1 = 10000.123;
     13:   int n;
     14:
     15:   puts("Outputting a number with different "
     16:        "field widths.\n");
     17:
     18:   printf("%5f\n", d1);
     19:   printf("%10f\n", d1);
     20:   printf("%15f\n", d1);
     21:   printf("%20f\n", d1);
     22:   printf("%25f\n", d1);
     23:
     24:   getchar();
     25:
     26:   puts("\nUse the * field width specifier to ");
     27:   puts("obtain field width from a variable in ");
     28:   puts("the argument list.\n");
     29:
```

```
30:   for (n = 5; n <= 25; n +=5)
31:     printf("%*f\n", n, d1);
32:
33:   getchar();
34:
35:   puts("\nInclude leading zeros.\n");
36:
37:   printf("%05f\n", d1);
38:   printf("%010f\n", d1);
39:   printf("%015f\n", d1);
40:   printf("%020f\n", d1);
41:   printf("%025f\n", d1);
42:
43:   getchar();
44:
45:   puts("Display in octal, decimal, and ");
46:   puts("hexadecimal. Use # to precede octal and ");
47:   puts("hex output with 0 and 0X. Use - to left-");
48:   puts("justify each value in its field. First ");
49:   puts("display column labels.\n");
50:
51:   printf("%-15s%-15s%-15s", m2, m3, m4);
52:
53:   for (n = 1; n < 20; n++)
54:     printf("\n%-15d%-#15o%-#15X", n, n, n);
55:
56:   getchar();
57:
58:   puts("\n\nUse the %n conversion command to "
59:        "count characters.\n");
60:
61:   printf("%s%s%s%s%n", m1, m2, m3, m4, &n);
62:   printf("\n\nThe last printf() output %d "
63:          "characters.", n);
64:
65:   getchar();
66:   return 0;
67: }
```

Output

```
Outputting a number with different field widths.

10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047



Use the * field width specifier to
obtain field width from a variable in
the argument list.
```

```
10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047



Include leading zeros.

10000.123047
10000.123047
00010000.123047
0000000010000.123047
000000000000010000.123047

Display in octal, decimal, and
hexadecimal. Use # to precede octal and
hex output with 0 and 0X. Use - to left-
justify each value in its field. First
display column labels.

Decimal        Octal          Hexadecimal
1              01             0X1
2              02             0X2
3              03             0X3
4              04             0X4
5              05             0X5
6              06             0X6
7              07             0X7
8              010            0X8
9              011            0X9
10             012            0XA
11             013            0XB
12             014            0XC
13             015            0XD
14             016            0XE
15             017            0XF
16             020            0X10
17             021            0X11
18             022            0X12
19             023            0X13

Use the %n conversion command to count characters.
BinaryDecimalOctalHexadecimal
The last printf() output 29 characters.
```

Description This program demonstrates many of the ways you can use printf(). Remember that the syntax for the conversion expression is:

%[*flag*][*fieldWidth*][.*precision*][l]*conversionChar*

Lines 18 – 22 display d1 as a float value (f) with different fieldWidth values. Because float values are right-justified by default, leading spaces are inserted. Notice that the fieldWidth only applies to the integer portion of d1 (the decimal portion is determined by the precision, which is 5 by default).

Line 24 clears the stdin of extra characters that may have been entered. getchar() is executed again after each user entry.

In lines 30 – 31 a for statement displays the same thing as lines 18 – 22 did, but instead uses a count variable (n) to increment the field width. In this case the * is used for the fieldWidth value to indicate that there is a second argument (n) to be used as the field width specifier.

The conversion statements in lines 37 – 41 also do the same thing as lines 18 – 22, except they have zeros preceding the fieldWidth value, which results in leading zeros rather than leading spaces.

Line 51 displays three strings from pointers by using the s conversion character. The dash (-) is used as a flag to left-justify the strings, and a field width of 15 characters adds space between them. These will serve as column headers for the values printed by the following for loop.

In lines 53 – 54 the for loop prints the values 1 – 20 in three conversion formats: decimal (d), octal (o), and hexadecimal (X). Note that the octal and hex conversion specifiers are preceded the # flag. The # displays leading zeros for octal and leading OX for hex values.

In lines 61 – 63 the conversion expression is used to display four strings from pointers, but this time there are no spaces between them. Also, the conversion command %n is added at the end (along with an additional argument &n). This command counts the number of characters displayed by the entire printf statement, storing the value at the address specified by n, and displays the statement "The last printf() output *n* characters."

Topic 7.5: Redirection of Input and Output

Redirection
A program that uses stdin and stdout can utilize an operating-system feature called *redirection*.

Usage
Redirection enables you to do the following:

- Output sent to stdout can be sent to a disk file or the printer rather than to the screen.
- Program input from stdin can come from a disk file rather than from the keyboard.

Symbols for Redirection
You don't code redirection into your programs; you specify it on the command line when you run the program. In DOS, the symbols for redirection are < and >. Redirection of output is covered first.

Redirection of Output
Remember your first C program, HELLO.C? It used the printf() library function to display the message `Hello, world` on-screen. As you now know, printf() sends output to stdout, so it can be redirected. When you enter the program name at the DOS prompt, follow it with the > symbol and the name of the new destination. Click the Examples link.

```
hello >destination
```

Examples

Caution
When you redirect output to a disk file, be careful. If the file already exists, the old copy is deleted and replaced with the new file. If the file does not exist, it is created. When redirecting output to a file, you also can use the >> symbol. If the specified destination file already exists, the program output is appended at the end of the file.

Thus, if you enter `hello >prn`, the program output goes to the printer instead of to the screen (`prn` is the DOS name for the printer attached to port LPT1:). If you enter `hello >hello.txt`, the output is placed in a disk file with the name `hello.txt`.

Example Program

The program in Listing 14.15 demonstrates redirection.

Listing 14.15: Program to Demonstrate Redirection of Input and Output

Code

```
 1: /* LIST1415.c: Day 14 Listing 14.15 */
 2: /* Can be used to demonstrate redirection of */
 3: /* stdin and stdout. */
 4:
 5: #include <stdio.h>
 6:
 7: int main(void) {
 8:   char buf[80];
 9:
10:   gets(buf);
11:   printf("The input was: %s", buf);
12:   return 0;
13: }
```

Description

This program accepts a line of input from stdin and then sends the line to stdout, preceding it by `The input was:`. After compiling and linking the program, you run it without redirection (assuming the program is named LIST1415) by entering `LIST1415` at the DOS prompt. If you then enter `C is a powerful language`, the program displays the following on-screen:

```
The input was: C is a powerful language
```

If you run the program by entering `LIST1415 >test.txt`, and make the same entry, there is nothing displayed on-screen. Instead, a file named `test.txt` is created on the disk. Use the `DOS TYPE` (or an equivalent) command to display the contents of the file: `type test.txt` and you see the file contains the one line `The input was: C is a powerful language.`

Similarly, if you had run the program by entering `LIST1415 >prn`, the output line would have been printed on your printer. Run the program again, this time redirecting output to `TEST.TXT` with the >> symbol. Instead of replacing the file, the new output is appended to the end of `TEST.TXT`.

Redirection of Input

Now take a look at redirecting input. First, you need a source file. Use your editor to create a file named `input.txt` that contains the single line `William Shakespeare`. Now, run the previous program by entering the following at the DOS prompt:

```
list1415 <input.txt
```

The program does not wait for you to make an entry at the keyboard. Rather, it immediately displays the message

```
The input was: William Shakespeare
```

on-screen. The stream stdin was redirected to the disk file `input.txt`, so the program's call to gets() reads one line of text from the file rather than the keyboard.

Simultaneous Redirection of Input and Output

You can redirect input and output at the same time. Try running the program with the command

```
list1415 <input.txt >junk.txt
```

to redirect stdin to the file `input.txt` and redirect stdout to `junk.txt`.

Usage

Redirection of stdin and stdout can be useful in certain situations. A sorting program, for example, could sort either keyboard input or the contents of a disk file. Likewise, a mailing list program could display addresses on-screen, send them to the printer for mailing labels, or place them in a file for some other use.

Redirection is a Feature of the Operation System

Please remember that redirection of stdin and stdout is a feature of the operating system and not of the C language itself. It does provide, however, another example of the flexibility of streams.

Topic 7.6: When to Use fprintf()

Description

As mentioned earlier, the library function fprintf() is identical to printf() except that you can specify the stream to which output is sent. The main use for fprintf() is with disk files, covered on Day 16, "Using Disk Files." There are two other uses, as explained here.

Topic 7.6.1: Using stderr

stderr Cannot Be Redirected

One of C's predefined streams is stderr, standard error. A program's error messages traditionally are sent to the stream stderr and not stdout. Why is this?

As you just learned, output to stdout can be redirected to a destination other than the display screen. If stdout is redirected, the user may not be aware of any error messages the program sends to stdout. Unlike stdout, stderr cannot be redirected and is always connected to the screen. By directing error messages to stderr, you can be sure the user always sees them. You do this with fprintf():

```
fprintf(stderr, "An error has occurred.");
```

A User-Defined Function Versus fprintf()

You can write a function to handle error messages, and then call the function when an error occurs rather than calling fprintf(). Click the Example link and then the Tip button.

Example

By using your own function instead of directly calling fprintf(), you provide additional flexibility (one of the advantages of structured programming). For example, in special circumstances you might want a program's error messages to go to the printer or a disk file. All you need do is modify the error_message() function so that output is sent to the desired destination.

DON'T ever try to redirect stderr.

DO use fprintf() to create programs that can print to stdout, stderr, stdprn, or any other stream.

DO use fprintf() with stderr to print error messages to the screen.

DON'T use stderr for purposes other than printing error messages or warnings.

DO create functions such as error_message to make your code more structured and more maintainable.

```
error_message("An error has occurred.");

void error_message(char *msg) {
   fprintf(stderr, msg);
}
```

Topic 7.6.2: Printer Output

stdprn
You send output to your printer by accessing the predefined stream stdprn. On IBM PCs and compatibles, the stream stdprn is connected to the device LPT1: (the first parallel printer port). Listing 14.16 presents a simple example.

Listing 14.16: Sending Output to the Printer

Code

```
 1: /* LIST1416.c: Day 14 Listing 14.16 */
 2: /* Demonstrates printer output. */
 3: /* MS-DOS Application Only */
 4: /* Non-ANSI Application */
 5:
 6: #include <stdio.h>
 7:
 8: int main(void) {
 9:   float f = 2.0134;
10:
11:   fprintf(stdprn, "This message is printed.\n\n");
12:   fprintf(stdprn, "And now some numbers:\n\n");
13:   fprintf(stdprn, "The square of %f is %f.",
14:           f, f*f);
15:
16:   /* Send a form feed. */
17:
18:   fprintf(stdprn, "\f");
19:   return 0;
20: }
```

Output

```
This message is printed.
And now some numbers:
The square of 2.013400 is 4.053780.
```

Note: The previous output is printed by your printer. This output won't appear on your screen.

Description	If your system has a printer connected to port LPT1:, you can compile and run the program in Listing 14.16. It prints three lines on the page. Line 18 sends a "\f" to the printer. \f is the escape sequence for a form feed, the command that causes the printer to advance a page (or, in the case of a laser printer, to eject the current page).

Topic 7.7: Day 14 Q&A

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

Question 1
What happens if I send output to an input device?

Answer
You can write a C program to do this; however, it won't work! For example, if you try to use stdprn with fscanf(), the program compiles into an executable file, but the printer is incapable of sending input, so your program doesn't operate as intended.

Question 2
What happens if I redirect one of the standard streams?

Answer
If you redirect one of the standard streams, you may cause problems later in the program. If you redirect a stream, you must put it back if you need it again in the same program. Many of the functions described in this unit use the standard streams. They all use the same streams, so if you change the stream in one place, you change it for all of the functions. For example, assign stdout equal to stdprn in one of the listings in the unit and see what happens!

Question 3
Is there any danger in using non-ANSI functions in a program?

Answer
Most compilers come with many useful functions that are not ANSI-standard. If you plan on always using that compiler, and not porting your code to other compilers or platforms, there is no problem. When you are going to use other compilers and platforms, you should be concerned with ANSI compatibility.

Question 4
Why can't I always use fprintf() instead of printf()? (Or fscanf() instead of scanf()?)

Answer
If you are using the standard output or input streams, you should use printf() and scanf(). By using these simpler functions, you do not have to bother with any other streams.

Topic 7.8: Day 14 Think About Its

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

Topic 7.9: Day 14 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 statement to print `"Hello World"` to the screen.

Answer

```
printf( "Hello World" );
```

*** Exercise 2**
Use two different C functions to do the same thing the function in exercise one did.

Answer

```
fprintf( stdout, "Hello World" );
puts( "Hello World");
```

*** Exercise 3**
Write a statement to print "Hello Auxiliary Port" to the standard auxiliary port.

Answer

```
fprintf( stdaux, "Hello Auxiliary Port" );
```

*** Exercise 4**
Write a statement that gets a string 30 characters or shorter. If an asterisk is encountered, truncate the string.

Answer

```
char buffer[31];
scanf( "%30[^*]", buffer );
```

*** Exercise 5**
Write a single statement that prints the following:

```
Jack asked, "What is a backslash?"
Jill said, "It is '\'"
```

Answer

```
printf("Jack asked, \"What is a backslash?\"\nJill said, \"It is
\'\\\'\"");
```

*** Exercise 6**
Write a program that redirects a file to the printer one character at a time.

Answer

```
/* EXER1406.C  Day 14 Exercise 6 */
/* MS-DOS Application */
/* This program will allow redirection from the */
/* input file INPUT.TXT to the printer. At the DOS */
/* prompt, type in EXER1406 < INPUT.TXT > PRN */

#include <stdio.h>

#define MAX 80

char c;
int x;

int main(void)  {

  while ((c = getchar()) != '\n' && x < MAX)
    putchar(c);

  return 0;
}
```

*** Exercise 7**

Write a program that prints your C source files. Use redirection to enter the source file, and use fprintf() to do the printing.

Answer

```
/* EXER1407.C Day 14 Exercise 7
 * MS-DOS Application
 * Print a C source file. Use redirection to enter
 * the file and use FPRINTF() to do the printing.
 * Note: To use stdprn you may need to turn ANSI
 * compatibility off in your compiler.
 */

#include <stdio.h>

#define MAX 80

char buffer[MAX];

int main()  {
```

```
  while (1)  {
    if (fgets(buffer,MAX,stdin) == 0)
      break;
    fprintf(stdprn,"%s",buffer);
  }
  return 0;
}
```

*** Exercise 8**

Write a "typing" program that accepts keyboard input, echoing it on the screen, and then reproduces this input on the printer. The program should count lines and advance the paper in the printer to a new page when necessary. Use a function key to terminate the program.

Answer

```
/* EXER1408.C  Day 14 Exercise 8 */
/* Types user's input to the printer */
/* with echoes to the screen */
/* MS-DOS Application */
/* Note: To use stdprn you may need to turn */
/* ANSI compatibility off in your compiler. */

#include <stdio.h>

int count;
int ch;

int main(void)  {

  count = 0;

  while (1)  {
    ch = getchar();

/* Was a function key pressed? */
    if (ch == 0 || ch == 0x0e) {
      ch = getchar();
      if (ch > 58 && ch < 69)
        break;
    }

    else  {
```

```
      if (ch == '\n')
        count++;

/* Eject the page if more than 60 lines printed. */
      if (count >= 60)  {
        fprintf(stdprn,"\f");
          count = 0;
      }
      fprintf(stdprn,"%c",ch);
    }
  }
  return 0;
}
```

Topic 7.10: Day 14 Summary

Streams and C
This was a long day, full of important information on program input and output. You learned how C uses streams, treating all input and output as a sequence of characters. You also learned that ANSI C has three predefined streams: stdin (the keyboard), stdout (the screen) and stderr (the screen) and MS-DOS compilers support two additional streams: stdprn (the printer) and stdaux (the communications port).

Accepting Keyboard Input
Input from the keyboard arrives from the stream stdin. Using C's standard library functions, you can accept keyboard input character by character, a line at a time, or as formatted numbers and strings. Character input can be buffered or unbuffered, echoed or unechoed.

Screen Output
Output to the display screen is normally done with the stdout stream. Like input, program output can be by character, by line, or as formatted numbers and strings. For output to the printer, you use fprintf() to send data to the stream stdprn.

Redirection of Input and Output
When you use stdin and stdout, you can redirect program input and output. Input can come from a disk file rather than the keyboard, and output can go to a disk file or to the printer rather than the display screen.

Using stderr
Finally, you learned why error messages should be sent to the stream stderr instead of stdout. Because stderr is always connected to the display screen, you are assured of

seeing error messages even when the program output is redirected.

Unit 8. Week 2 in Review

8. Week 2 in Review

Week 2 in Review

You have finished your second week of learning how to program in C. By now you should feel comfortable with the C language. You have covered almost all of the basic C commands. Listing R2.1 pulls together many of the topics from the previous week.

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 R2.1: Week Two Review Listing

```
Code  1: /* Program: week2.c                                    */
      2: /* A program to enter information for up to 100 */
      3: /* people.  The program displays a report based */
      4: /* on the numbers entered.                         */
      5:
      6: /* Included files. */
      7:
      8: #include <stdio.h>
      9: #include <conio.h>
     10:
     11: /* Defined constants. */
     12:
     13: #define MAX 100
     14: #define YES 1
     15: #define NO  0
     16:
     17: /* Variables. */
     18:
D11  19: struct record {
D10  20:   char fname[15+1];  /* First name + NULL   */
     21:   char lname[20+1];  /* Last name + NULL    */
     22:   char phone[9+1];   /* Phone number + NULL */
     23:   long income;       /* Incomes             */
     24:   int  month;        /* Birthday month      */
     25:   int  day;          /* Birthday day        */
```

```
     26:   int  year;          /* Birthday year       */
     27: };
     28:
D11  29: struct record list[MAX];  /* Declare structure. */
     30:
D12  31: int last_entry = 0;  /* Total number of entries. */
     32:
     33: /* Function prototypes. */
     34:
     35: void get_data(void);
     36: void display_report(void);
D14  37: void clear_kb(void);
     38: int continue_function(void);
D12  39: int cont = YES;
     40: int ch;
     41: char buf[80];
     42:
     43: /* Start of program. */
     44:
     45: int main(void) {
     46:
     47:   while (cont == YES) {
     48:
     49:     printf("\n");
     50:     printf("\n     MENU");
     51:     printf("\n   ========\n");
     52:     printf("\n1. Enter names");
     53:     printf("\n2. Display report");
     54:     printf("\n3. Quit");
     55:     printf("\n\nEnter Selection ==> ");
     56:
D10  57:     gets(buf);
     58:     if (buf[1] == '\0')
     59:       ch = buf[0] - 0x30;
     60:     else
     61:       ch = 0;
     62:
D13  63:     switch(ch) {
     64:       case 1: get_data();
     65:         break;
     66:       case 2: display_report();
     67:         break;
     68:       case 3: printf("\n\nThank you for using "
     69:                      "this program!");
     70:         cont = NO;
     71:         break;
     72:       default:  printf("\n\nInvalid choice, "
     73:                        "Please select 1 to 3!");
     74:         break;
     75:     }
     76:   }
     77:   return 0;
```

```
     78: }
     79:
     80: /* Function: get_data()                              */
     81: /* Purpose: This function gets the data from        */
     82: /* the user. It continues to get data until         */
     83: /* either 100 people are entered, or the user       */
     84: /* chooses not to continue. Returns: Nothing        */
     85: /* Notes: This allows 0/0/0/ to be entered for      */
     86: /* birthdays in case the user is unsure.  It        */
     87: /* also allows for 31 days in each month.           */
     88:
     89: void get_data(void) {
     90:
     91:   int cont;
     92:
     93:   for (cont = YES; last_entry < MAX && cont == YES;
     94:        last_entry++) {
     95:
     96:     printf("\n\nEnter information for Person %d.",
     97:            last_entry + 1);
     98:
D10  99:     printf("\n\nEnter first name: ");
    100:     gets(list[last_entry].fname);
    101:
D11 102:     printf("\nEnter last name: ");
    103:     gets(list[last_entry].lname);
    104:
D08 105:     printf("\nEnter phone in 123-4567 format: ");
    106:     gets(list[last_entry].phone);
    107:
D11 108:     printf("\nEnter yearly income (whole $s): ");
    109:     scanf("%ld", &list[last_entry].income);
    110:
    111:     printf("\nEnter Birthday:");
    112:
    113:     do {
    114:       printf("\n\tMonth (0-12): ");
    115:       scanf("%d", &list[last_entry].month);
    116:     } while (list[last_entry].month < 0 ||
    117:              list[last_entry].month > 12);
    118:
    119:     do {
    120:       printf("\n\tDay (0-31): ");
    121:       scanf("%d", &list[last_entry].day);
    122:     } while (list[last_entry].day < 0 ||
    123:              list[last_entry].day > 31);
    124:
    125:     do {
    126:        printf("\n\tYear (1800-1999): ");
    127:        scanf("%d", &list[last_entry].year);
    128:     } while (list[last_entry].year != 0 &&
    129:             (list[last_entry].year < 1800 ||
```

```
    130:                 list[last_entry].year > 1999));
    131:
    132:       cont = continue_function();
    133:     }
    134:     if (last_entry == MAX)
    135:       printf("\n\nMaximum number of names has been "
    136:              "entered!\n");
    137:  }
    138:
    139:  /* Function: display_report()                    */
    140:  /* Purpose: This function displays a report to */
    141:  /* the screen. Returns: Nothing                 */
    142:  /* Notes: More information could be displayed. */
    143:  /* Change stdout to stdprn to Print report.    */
    144:
    145:  void display_report(void) {
    146:
D12 147:    long month_total = 0, grand_total = 0;
    148:    /* For totals. */
    149:    int x, y;
    150:
D14 151:    fprintf(stdout, "\n\n");
    152:    fprintf(stdout, "\n          REPORT");
    153:    fprintf(stdout, "\n         ========");
    154:
    155:    for (x = 0; x <= 12; x++) {
    156:       /* For each month, including 0 */
    157:
    158:       month_total = 0;
    159:       for (y = 0; y < last_entry; y++) {
    160:         if (list[y].month == x) {
D14 161:           fprintf(stdout, "\n\t%s %s %s %ld",
    162:                   list[y].fname,
D11 163:                   list[y].lname, list[y].phone,
    164:                   list[y].income);
    165:                   month_total += list[y].income;
    166:         }
    167:       }
    168:       fprintf(stdout, "\nTotal for month %d is %ld",
    169:               x, month_total);
    170:       grand_total += month_total;
    171:     }
    172:     fprintf(stdout, "\n\nReport totals:");
    173:     fprintf(stdout, "\nTotal Income is %ld",
    174:             grand_total);
    175:     if (last_entry)
    176:       fprintf(stdout, "\nAverage Income is %ld",
    177:               grand_total/last_entry);
    178:
    179:     fprintf(stdout, "\n\n*** End of Report ***");
    180:  }
    181:
```

```
    182: /* Function: continue_function()                */
    183: /* Purpose: This function asks the user if they  */
    184: /* want to continue. Returns: YES - if user      */
    185: /* wants to continue, NO - if user wants to quit */
    186:
    187: int continue_function(void) {
    188:   char ch;
    189:
    190:   clear_kb();
    191:
    192:   printf("\n\nDo you wish to continue? "
    193:          "(Y)es/(N)o:");
D14 194:   ch = getchar();
    195:
    196:   while (ch != 'n' && ch != 'N' &&
    197:          ch != 'y' && ch != 'Y') {
    198:      printf("\n%c is invalid!", ch);
    199:      printf("\n\nPlease enter \'N\' to Quit or "
    200:             "\'Y\' to Continue: ");
D14 201:      ch = getchar();
    202:    }
    203:
D14 204:    clear_kb();
    205:
    206:    if (ch == 'n' || ch == 'N')
    207:       return(NO);
    208:    else
    209:       return(YES);
    210: }
    211:
    212: /* Function: clear_kb()                          */
    213: /* Purpose: This function clears the keyboard   */
    214: /* of extra characters. Returns: Nothing        */
    215:
    216: void clear_kb(void) {
D09 217:   char junk[80];
D10 218:   gets(junk);
    219: }
```

Description

It appears that as you learn about C, your programs grow larger. This program resembles the one presented after your first week of programming in C, but this program changes a few of the tasks and adds a few more. Like week one's review, you can enter up to 100 sets of information. The data entered is information about people. You should notice that this program can display the report while you enter information. With the other program, you could not print the report until you had finished entering the data.

You also should notice the addition of a structure used to save the data. The structure is defined on lines 19 – 27 of this program.

Structures often are used to group similar data (Day 11, "Structures"). This program groups all the data for each person into a structure named record. Much of this data should look familiar; however, there are a few new items being tracked. Lines 20 – 22 contain three arrays, or strings, of characters to hold the first name, last name, and phone number. Notice that each of these strings is declared with `a+1` in its array size. You should remember from Day 10, "Characters and Strings," that this extra spot holds the NULL character that signifies the end of a string.

This program demonstrates proper use of variable scope (Day 12, "Variable Scope"). Lines 29 and 31 contain two global variables. Line 31 uses an int called last_entry to hold the number of people that have been entered. This is similar to the variable ctr used in the review at the end of week one. The other global variable is list[MAX], an array of record structures. Local variables are used in each of the functions throughout the program. Of special note are the variables month_total, grand_total, x, and y on lines 147 – 149 in display_report(). In the week one review, these were global variables. Because these variables only apply to the display_report(), they are better placed as local variables.

An additional program control statement, the switch statement (Day 13, "More Program Control") is used on lines 63 – 75. Using a switch statement instead of several if...else statements makes the code easier to follow. Lines 64 – 75 execute various tasks based on a menu choice. Notice that the default statement is also included in case you enter a value that is not a valid menu option.

Looking at the get_data() function, you should notice that there are some additional changes from the week one review. Lines 99 – 100 prompt for a string. Line 100 uses the gets() function (Day 14, "Working with the Screen, Printer, and Keyboard") to accept a person's first name. The gets() function gets a string and places the value in list[last_entry].fname. You should remember from Day 11, "Structures," that this places the first name into fname, a member of the structure list.

display_report() has been modified, using fprintf() instead of printf() to display the information. The reason for this change is simple. If you want the report to go to the printer instead of the

	screen, on each fprintf() statement, change stdout to stdprn. (Though you may need to turn off ANSI compatibility so your compiler will recognize stdprn.) Day 14, "Working with the Screen, Printer, and Keyboard," covered fprintf(), stdout, and stdprn. Remember that stdout and stdprn are streams that output to the screen and the printer respectively. continue_function() on lines 187 – 210 also has been modified. You now respond to the question with Y or N instead of 0 or 1. This is more friendly. Also notice that the clear_kb() function from Listing 13.9 has been added on line 204 to remove any extra characters that the user entered. This program uses what you learned in your first two weeks of teaching yourself C. As you can see, many of the concepts from the second week make your C programs more functional and coding in C easier. Week three continues to build on these concepts.

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 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

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)
```

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. Here are 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 (all in 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
```