4

# Character Strings and Formatted Input/Output

You will learn about the following in this chapter:

- Function:

  `strlen()`

- Keywords:

  `const`

- Character strings
- How character strings are created and stored
- How you can use `scanf()` and `printf()` to read and display character strings
- How to use the `strlen()` function to measure string lengths
- The C preprocessor's `#define` directive and ANSI C's `const` modifier for creating symbolic constants

This chapter concentrates on input and output. You'll add personality to your programs by making them interactive and using character strings. You will also take a more detailed look at those two handy C input/output functions, `printf()` and `scanf()`. With these two functions, you have the program tools you need to communicate with users and to format output to meet your needs and tastes. Finally, you'll take a quick look at an important C facility, the C preprocessor, and learn how to define and use symbolic constants.

## Introductory Program

By now, you probably expect a sample program at the beginning of each chapter, so Listing 4.1 is a program that engages in a dialog with the user. To add a little variety, the code uses the newer comment style.

Listing 4.1 **The `talkback.c` Program**

```
// talkback.c -- nosy, informative program
#include <stdio.h>
#include <string.h>      // for strlen() prototype
#define DENSITY 62.4     // human density in lbs per cu ft
int main()
{
    float weight, volume;
    int size, letters;
    char name[40];        // name is an array of 40 chars

    printf("Hi! What's your first name?\n");
    scanf("%s", name);
    printf("%s, what's your weight in pounds?\n", name);
    scanf("%f", &weight);
    size = sizeof name;
    letters = strlen(name);
    volume = weight / DENSITY;
    printf("Well, %s, your volume is %2.2f cubic feet.\n",
            name, volume);
    printf("Also, your first name has %d letters,\n",
            letters);
    printf("and we have %d bytes to store it.\n", size);

    return 0;
}
```

Running `talkback.c` produces results such as the following:

```
Hi! What's your first name?
Christine
Christine, what's your weight in pounds?
154
Well, Christine, your volume is 2.47 cubic feet.
Also, your first name has 9 letters,
and we have 40 bytes to store it.
```

Here are the main new features of this program:

- It uses an *array* to hold a *character string*. Here, someone's name is read into the array, which, in this case, is a series of 40 consecutive bytes in memory, each able to hold a single character value.
- It uses the `%s` *conversion specification* to handle the input and output of the string. Note that `name`, unlike `weight`, does not use the `&` prefix when used with `scanf()`. (As you'll see later, both `&weight` and `name` are addresses.)

- It uses the C preprocessor to define the symbolic constant `DENSITY` to represent the value `62.4`.
- It uses the C function `strlen()` to find the length of a string.

The C approach might seem a little complex compared with the input/output of, say, BASIC. However, this complexity buys a finer control of I/O and greater program efficiency, and it's surprisingly easy once you get used to it.

Let's investigate these new ideas.

## Character Strings: An Introduction

A *character string* is a series of one or more characters. Here is an example of a string:

```
"Zing went the strings of my heart!"
```

The double quotation marks are not part of the string. They inform the compiler that they enclose a string, just as single quotation marks identify a character.

### Type `char` Arrays and the Null Character

C has no special variable type for strings. Instead, strings are stored in an array of type `char`. Characters in a string are stored in adjacent memory cells, one character per cell, and an array consists of adjacent memory locations, so placing a string in an array is quite natural (see Figure 4.1).

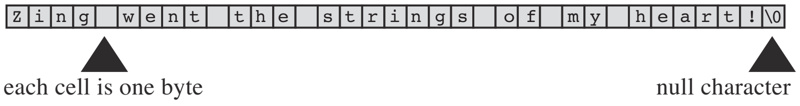


Figure 4.1 A string in an array.

Note that Figure 4.1 shows the character `\0` in the last array position. This is the *null character*, and C uses it to mark the end of a string. The null character is not the digit zero; it is the nonprinting character whose ASCII code value (or equivalent) is `0`. Strings in C are always stored with this terminating null character. The presence of the null character means that the array must have at least one more cell than the number of characters to be stored. So when the preceding program said it had 40 bytes to store the string, that meant it could hold up to 39 characters in addition to the null character.

Now just what is an array? You can think of an array as several memory cells in a row. If you prefer more formal language, an array is an ordered sequence of data elements of one type. This example creates an array of 40 memory cells, or *elements*, each of which can store one `char`-type value by using this declaration:

```
char name[40];
```

The brackets after `name` identify it as an array. The `40` within the brackets indicates the number of elements in the array. The `char` identifies the type of each element (see Figure 4.2).

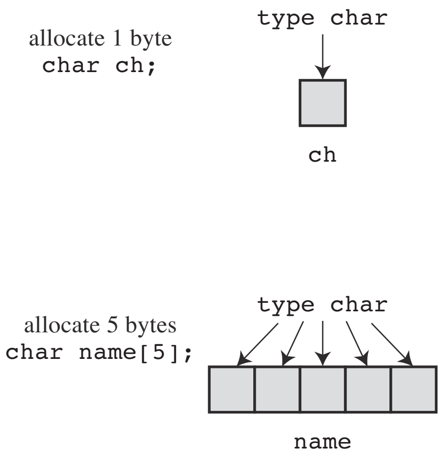


Figure 4.2 Declaring a variable versus declaring an array.

Using a character string is beginning to sound complicated! You have to create an array, place the characters of a string into an array, one by one, and remember to add `\0` at the end. Fortunately, the computer can take care of most of the details itself.

## Using Strings

Try the program in Listing 4.2 to see how easy it really is to use strings.

Listing 4.2 **The `praise1.c` Program**

```
/* praise1.c -- uses an assortment of strings */
#include <stdio.h>
#define PRAISE "You are an extraordinary being."
int main(void)
{
    char name[40];

    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello, %s. %s\n", name, PRAISE);

    return 0;
}
```

The `%s` tells `printf()` to print a string. The `%s` appears twice because the program prints two strings: the one stored in the `name` array and the one represented by `PRAISE`. Running `praise1.c` should produce an output similar to this:

```
What's your name? Angela Plains
Hello, Angela. You are an extraordinary being.
```

You do not have to put the null character into the `name` array yourself. That task is done for you by `scanf()` when it reads the input. Nor do you include a null character in the *character string constant* `PRAISE`. We'll explain the `#define` statement soon; for now, simply note that the double quotation marks that enclose the text following `PRAISE` identify the text as a string. The compiler takes care of putting in the null character.

Note (and this is important) that `scanf()` just reads Angela Plains's first name. After `scanf()` starts to read input, it stops reading at the first *whitespace* (blank, tab, or newline) it encounters. Therefore, it stops scanning for `name` when it reaches the blank between `Angela` and `Plains`. In general, `scanf()` is used with `%s` to read only a single word, not a whole phrase, as a string. C has other input-reading functions, such as `fgets()`, for handling general strings. Later chapters will explore string functions more fully.

### Strings Versus Characters

The string constant `"x"` is not the same as the character constant `'x'`. One difference is that `'x'` is a basic type (`char`), but `"x"` is a derived type, an array of `char`. A second difference is that `"x"` really consists of two characters, `'x'` and `'\0'`, the null character (see Figure 4.3).

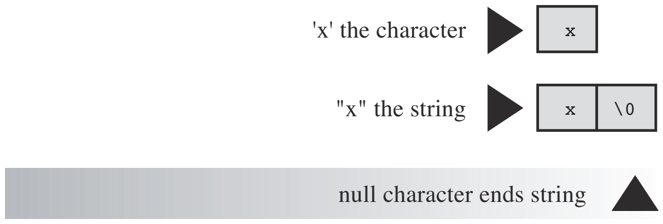


Figure 4.3 The character `'x'` and the string `"x"`.

## The `strlen()` Function

The previous chapter unleashed the `sizeof` operator, which gives the size of things in bytes. The `strlen()` function gives the length of a string in characters. Because it takes one byte to hold one character, you might suppose that both would give the same result when applied to a string, but they don't. Add a few lines to the example, as shown in Listing 4.3, and see why.

Listing 4.3 **The `praise2.c` Program**

```
/* praise2.c */
// try the %u or %lu specifiers if your implementation
// does not recognize the %zd specifier
#include <stdio.h>
#include <string.h>      /* provides strlen() prototype */
#define PRAISE "You are an extraordinary being."
int main(void)
{
    char name[40];

    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello, %s. %s\n", name, PRAISE);
    printf("Your name of %zd letters occupies %zd memory cells.\n",
           strlen(name), sizeof name);
    printf("The phrase of praise has %zd letters ",
           strlen(PRAISE));
    printf("and occupies %zd memory cells.\n", sizeof PRAISE);

    return 0;
}
```

If you are using a pre-ANSI C compiler, you might have to remove the following line:

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

The `string.h` file contains function prototypes for several string-related functions, including `strlen()`. Chapter 11, "Character Strings and String Functions," discusses this header file more fully. (By the way, some pre-ANSI Unix systems use `strings.h` instead of `string.h` to contain declarations for string functions.)

More generally, C divides the C function library into families of related functions and provides a header file for each family. For example, `printf()` and `scanf()` belong to a family of standard input and output functions and use the `stdio.h` header file. The `strlen()` function joins several other string-related functions, such as functions to copy strings and to search through strings, in a family served by the `string.h` header.

Notice that Listing 4.3 uses two methods to handle long `printf()` statements. The first method spreads one `printf()` statement over two lines. (You can break a line between arguments to `printf()` but not in the middle of a string—that is, not between the quotation marks.) The second method uses two `printf()` statements to print just one line. The newline character (`\n`) appears only in the second statement. Running the program could produce the following interchange:

```
What's your name? Serendipity Chance
Hello, Serendipity. You are an extraordinary being.
```

```
Your name of 11 letters occupies 40 memory cells.
The phrase of praise has 31 letters and occupies 32 memory cells.
```

See what happens. The array name has 40 memory cells, and that is what the `sizeof` operator reports. Only the first 11 cells are needed to hold Serendipity, however, and that is what `strlen()` reports. The twelfth cell in the array name contains the null character, and its presence tells `strlen()` when to stop counting. Figure 4.4 illustrates this concept with a shorter string.

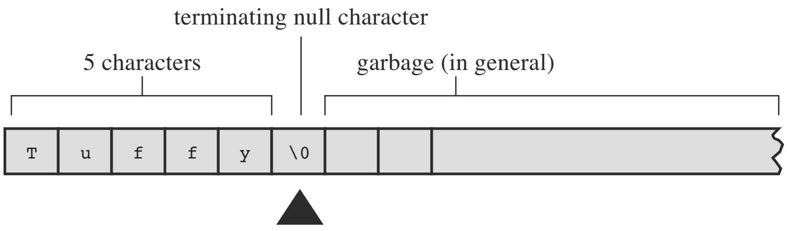


Figure 4.4 The `strlen()` function knows when to stop.

When you get to `PRAISE`, you find that `strlen()` again gives you the exact number of characters (including spaces and punctuation) in the string. The `sizeof` operator gives you a number one larger because it also counts the invisible null character used to end the string. The program didn't tell the computer how much memory to set aside to store the phrase. It had to count the number of characters between the double quotes itself.

As mentioned in Chapter 3, "Data and C," the C99 and C11 standards use a `%zd` specifier for the type used by the `sizeof` operator. This also applies for type returned by `strlen()`. For earlier versions of C you need to know the actual type returned by `sizeof` and `strlen()`; typically that would be `unsigned` or `unsigned long`.

One other point: The preceding chapter used `sizeof` with parentheses, but this example doesn't. Whether you use parentheses depends on whether you want the size of a type or the size of a particular quantity. Parentheses are required for types but are optional for particular quantities. That is, you would use `sizeof(char)` or `sizeof(float)` but can use `sizeof name` or `sizeof 6.28`. However, it is all right to use parentheses in these cases, too, as in `sizeof (6.28)`.

The last example used `strlen()` and `sizeof` for the rather trivial purpose of satisfying a user's potential curiosity. Actually, however, `strlen()` and `sizeof` are important programming tools. For example, `strlen()` is useful in all sorts of character-string programs, as you'll see in Chapter 11.

Let's move on to the `#define` statement.

## Constants and the C Preprocessor

Sometimes you need to use a constant in a program. For example, you could give the circumference of a circle as follows:

```
circumference = 3.14159 * diameter;
```

Here, the constant 3.14159 represents the world-famous constant pi (π). To use a constant, just type in the actual value, as in the example. However, there are good reasons to use a *symbolic constant* instead. That is, you could use a statement such as the following and have the computer substitute in the actual value later:

```
circumference = pi * diameter;
```

Why is it better to use a symbolic constant? First, a name tells you more than a number does. Compare the following two statements:

```
owed = 0.015 * housevalue;
owed = taxrate * housevalue;
```

If you read through a long program, the meaning of the second version is plainer.

Also, suppose you have used a constant in several places, and it becomes necessary to change its value. After all, tax rates do change. Then you only need to alter the definition of the symbolic constant, rather than find and change every occurrence of the constant in the program.

Okay, how do you set up a symbolic constant? One way is to declare a variable and set it equal to the desired constant. You could write this:

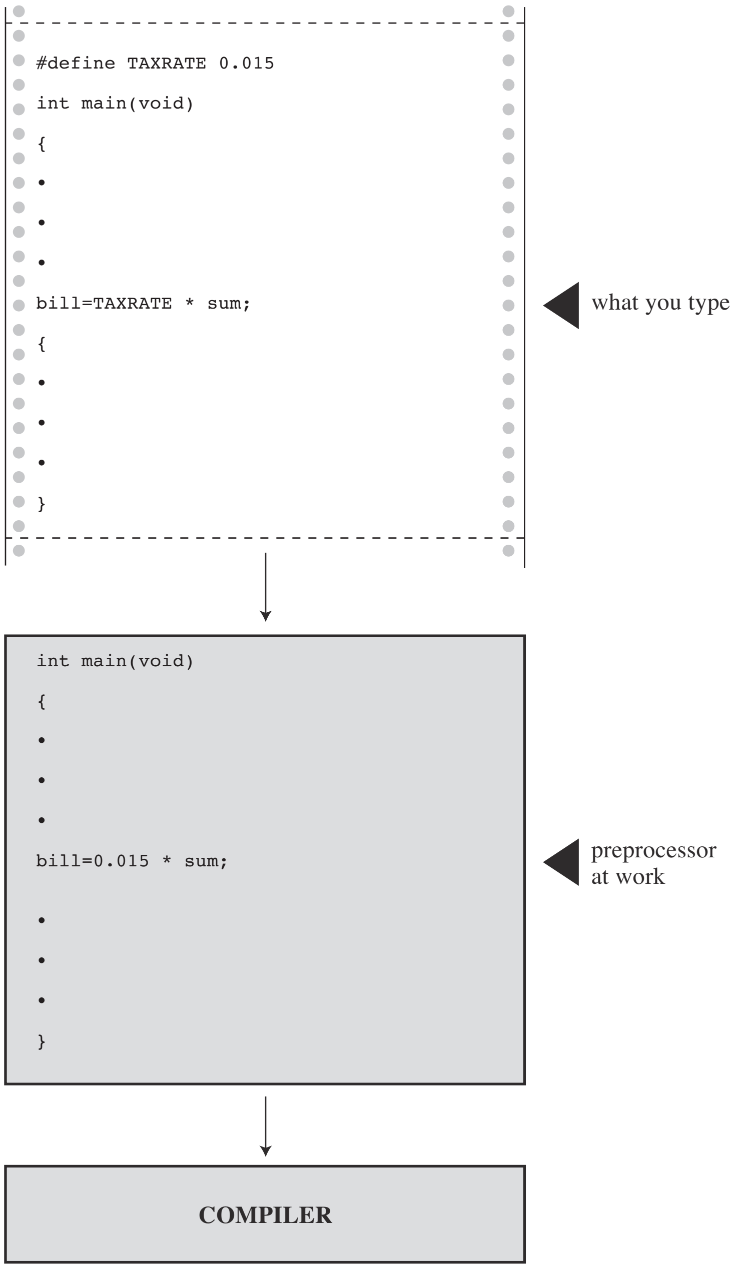
```
float taxrate;
taxrate = 0.015;
```

This provides a symbolic name, but `taxrate` is a variable, so your program might change its value accidentally. Fortunately, C has a couple better ideas.

The original better idea is the C preprocessor. In Chapter 2, "Introducing C," you saw how the preprocessor uses `#include` to incorporate information from another file. The preprocessor also lets you define constants. Just add a line like the following at the top of the file containing your program:

```
#define TAXRATE 0.015
```

When your program is compiled, the value `0.015` will be substituted everywhere you have used `TAXRATE`. This is called a *compile-time substitution*. By the time you run the program, all the substitutions have already been made (see Figure 4.5). Such defined constants are often termed *manifest constants*.



```
#define TAXRATE 0.015
int main(void)
{
•
•
•
bill=TAXRATE * sum;
{
•
•
•
}
```

```
int main(void)
{
•
•
•
bill=0.015 * sum;
•
•
•
}
```

Figure 4.5 What you type versus what is compiled.

Note the format. First comes `#define`. Next comes the symbolic name (`TAXRATE`) for the constant and then the value (`0.015`) for the constant. (Note that this construction does not use the = sign.) So the general form is as follows:

```
#define NAME value
```

You would substitute the symbolic name of your choice for *`NAME`* and the appropriate value for *`value`*. No semicolon is used because this is a substitution mechanism handled by the preprocessor, not a C statement. Why is `TAXRATE` capitalized? It is a sensible C tradition to type constants in uppercase. Then, when you encounter one in the depths of a program, you know immediately that it is a constant, not a variable. Capitalizing constants is just another technique to make programs more readable. Your programs will still work if you don't capitalize the constants, but capitalizing them is a reasonable habit to cultivate.

Other, less common, naming conventions include prefixing a name with a `c_` or `k_` to indicate a constant, producing names such as `c_level` or `k_line`.

The names you use for symbolic constants must satisfy the same rules that the names of variables do. You can use uppercase and lowercase letters, digits, and the underscore character. The first character cannot be a digit. Listing 4.4 shows a simple example.

Listing 4.4 **The `pizza.c` Program**

```
/* pizza.c -- uses defined constants in a pizza context */
#include <stdio.h>
#define PI 3.14159
int main(void)
{
    float area, circum, radius;

    printf("What is the radius of your pizza?\n");
    scanf("%f", &radius);
    area = PI * radius * radius;
    circum = 2.0 * PI *radius;
    printf("Your basic pizza parameters are as follows:\n");
    printf("circumference = %1.2f, area = %1.2f\n", circum,
           area);
    return 0;
}
```

The `%1.2f` in the `printf()` statement causes the printout to be rounded to two decimal places. Of course, this program may not reflect your major pizza concerns, but it does fill a small niche in the world of pizza programs. Here is a sample run:

```
What is the radius of your pizza?
6.0
Your basic pizza parameters are as follows:
circumference = 37.70, area = 113.10
```

The `#define` statement can be used for character and string constants, too. Just use single quotes for the former and double quotes for the latter. The following examples are valid:

```
#define BEEP '\a'
#define TEE 'T'
#define ESC '\033'
#define OOPS "Now you have done it!"
```

Remember that everything following the symbolic name is substituted for it. Don't make this common error:

```
/* the following is wrong */
#define TOES = 20
```

If you do this, `TOES` is replaced by `= 20`, not just `20`. In that case, a statement such as

```
digits = fingers + TOES;
```

is converted to the following misrepresentation:

```
digits = fingers + = 20;
```

## The `const` Modifier

C90 added a second way to create symbolic constants—using the `const` keyword to convert a declaration for a variable into a declaration for a constant:

```
const int MONTHS = 12;    // MONTHS a symbolic constant for 12
```

This makes `MONTHS` into a read-only value. That is, you can display `MONTHS` and use it in calculations, but you cannot alter the value of `MONTHS`. This newer approach is more flexible than using `#define`; it lets you declare a type, and it allows better control over which parts of a program can use the constant. Chapter 12, "Storage Classes, Linkage, and Memory Management," discusses this and other uses of `const`.

Actually, C has yet a third way to create symbolic constants, and that is the `enum` facility discussed in Chapter 14, "Structures and Other Data Forms."

## Manifest Constants on the Job

The C header files `limits.h` and `float.h` supply detailed information about the size limits of integer types and floating types, respectively. Each file defines a series of manifest constants that apply to your implementation. For instance, the `limits.h` file contains lines similar to the following:

```
#define INT_MAX    +32767
#define INT_MIN    -32768
```

These constants represent the largest and smallest possible values for the int type. If your system uses a 32-bit int, the file would provide different values for these symbolic constants. The file defines minimum and maximum values for all the integer types. If you include the limits.h file, you can use code such as the following:

```
printf("Maximum int value on this system = %d\n", INT_MAX);
```

If your system uses a 4-byte int, the limits.h file that comes with that system would provide definitions for INT_MAX and INT_MIN that match the limits of a 4-byte int. Table 4.1 lists some of the constants found in limits.h.

Table 4.1 **Some Symbolic Constants from limits.h**

| Symbolic Constant | Represents |
|---|---|
| CHAR_BIT | Number of bits in a char |
| CHAR_MAX | Maximum char value |
| CHAR_MIN | Minimum char value |
| SCHAR_MAX | Maximum signed char value |
| SCHAR_MIN | Minimum signed char value |
| UCHAR_MAX | Maximum unsigned char value |
| SHRT_MAX | Maximum short value |
| SHRT_MIN | Minimum short value |
| USHRT_MAX | Maximum unsigned short value |
| INT_MAX | Maximum int value |
| INT_MIN | Minimum int value |
| UINT_MAX | Maximum unsigned int value |
| LONG_MAX | Maximum long value |
| LONG_MIN | Minimum long value |
| ULONG_MAX | Maximum unsigned long value |
| LLONG_MAX | Maximum long long value |
| LLONG_MIN | Minimum long long value |
| ULLONG_MAX | Maximum unsigned long long value |

Similarly, the float.h file defines constants such as FLT_DIG and DBL_DIG, which represent the number of significant figures supported by the float type and the double type. Table 4.2 lists some of the constants found in float.h. (You can use a text editor to open and inspect the `float.h` header file your system uses.) This example relates to the `float` type. Equivalent constants are defined for types `double` and `long double`, with `DBL` and `LDBL` substituted for `FLT` in the name. (The table assumes the system represents floating-point numbers in terms of powers of 2.)

Table 4.2 **Some Symbolic Constants from `float.h`**

| Symbolic Constant | Represents |
|---|---|
| `FLT_MANT_DIG` | Number of bits in the mantissa of a `float` |
| `FLT_DIG` | Minimum number of significant decimal digits for a `float` |
| `FLT_MIN_10_EXP` | Minimum base-10 negative exponent for a `float` with a full set of significant figures |
| `FLT_MAX_10_EXP` | Maximum base-10 positive exponent for a `float` |
| `FLT_MIN` | Minimum value for a positive `float` retaining full precision |
| `FLT_MAX` | Maximum value for a positive `float` |
| `FLT_EPSILON` | Difference between 1.00 and the least float value greater than 1.00 |

Listing 4.5 illustrates using data from `float.h` and `limits.h`. (Note that a compiler that doesn't fully support the C99 standard might not accept the `LLONG_MIN` identifier.)

Listing 4.5 **The `defines.c` Program**

```
// defines.c -- uses defined constants from limit.h and float.
#include <stdio.h>
#include <limits.h>    // integer limits
#include <float.h>     // floating-point limits
int main(void)
{
    printf("Some number limits for this system:\n");
    printf("Biggest int: %d\n", INT_MAX);
    printf("Smallest long long: %lld\n", LLONG_MIN);
    printf("One byte = %d bits on this system.\n", CHAR_BIT);
    printf("Largest double: %e\n", DBL_MAX);
    printf("Smallest normal float: %e\n", FLT_MIN);
    printf("float precision = %d digits\n", FLT_DIG);
    printf("float epsilon = %e\n", FLT_EPSILON);

    return 0;
}
```

Here is the sample output:

```
Some number limits for this system:
Biggest int: 2147483647
Smallest long long: -9223372036854775808
One byte = 8 bits on this system.
Largest double: 1.797693e+308
Smallest normal float: 1.175494e-38
float precision = 6 digits
float epsilon = 1.192093e-07
```

The C preprocessor is a useful, helpful tool, so take advantage of it when you can. We'll show you more applications as you move along through this book.

## Exploring and Exploiting `printf()` and `scanf()`

The functions `printf()` and `scanf()` enable you to communicate with a program. They are called *input/output functions*, or *I/O functions* for short. They are not the only I/O functions you can use with C, but they are the most versatile. Historically, these functions, like all other functions in the C library, were not part of the definition of C. C originally left the implementation of I/O up to the compiler writers; this made it possible to better match I/O to specific machines. In the interests of compatibility, various implementations all came with versions of `scanf()` and `printf()`. However, there were occasional discrepancies between implementations. The C90 and C99 standards describe standard versions of these functions, and we'll follow that standard.

Although `printf()` is an output function and `scanf()` is an input function, both work much the same, each using a control string and a list of arguments. We will show you how these work, first with `printf()` and then with `scanf()`.

### The `printf()` Function

The instructions you give `printf()` when you ask it to print a variable depend on the variable type. For example, we have used the `%d` notation when printing an integer and the `%c` notation when printing a character. These notations are called *conversion specifications* because they specify how the data is to be converted into displayable form. We'll list the conversion specifications that the ANSI C standard provides for `printf()` and then show how to use the more common ones. Table 4.3 presents the conversion specifiers and the type of output they cause to be printed.

Table 4.3 **Conversion Specifiers and the Resulting Printed Output**

| **Conversion** | **Output Specification** |
|---|---|
| `%a` | Floating-point number, hexadecimal digits and p-notation (C99/C11). |
| `%A` | Floating-point number, hexadecimal digits and P-notation (C99/C11). |

| Conversion | Output Specification |
|---|---|
| `%c` | Single character. |
| `%d` | Signed decimal integer. |
| `%e` | Floating-point number, e-notation. |
| `%E` | Floating-point number, e-notation. |
| `%f` | Floating-point number, decimal notation. |
| `%g` | Use `%f` or `%e`, depending on the value. The `%e` style is used if the exponent is less than −4 or greater than or equal to the precision. |
| `%G` | Use `%f` or `%E`, depending on the value. The `%E` style is used if the exponent is less than −4 or greater than or equal to the precision. |
| `%i` | Signed decimal integer (same as `%d`). |
| `%o` | Unsigned octal integer. |
| `%p` | A pointer. |
| `%s` | Character string. |
| `%u` | Unsigned decimal integer. |
| `%x` | Unsigned hexadecimal integer, using hex digits `0f`. |
| `%X` | Unsigned hexadecimal integer, using hex digits `0F`. |
| `%%` | Prints a percent sign. |

## Using `printf()`

Listing 4.6 contains a program that uses some of the conversion specifications.

Listing 4.6 **The `printout.c` Program**

```
/* printout.c -- uses conversion specifiers */
#include <stdio.h>
#define PI 3.141593
int main(void)
{
    int number = 7;
    float pies = 12.75;
    int cost = 7800;

    printf("The %d contestants ate %f berry pies.\n", number,
           pies);
    printf("The value of pi is %f.\n", PI);
    printf("Farewell! thou art too dear for my possessing,\n");
    printf("%c%d\n", '$', 2 * cost);

    return 0;
}
```

The output, of course, is

```
The 7 contestants ate 12.750000 berry pies.
The value of pi is 3.141593.
Farewell! thou art too dear for my possessing,
$15600
```

This is the format for using `printf()`:

```
printf(Control-string, item1, item2,...);
```

*`Item1`*, *`item2`*, and so on, are the items to be printed. They can be variables or constants, or even expressions that are evaluated first before the value is printed. *`Control-string`* is a character string describing how the items are to be printed. As mentioned in Chapter 3, the control string should contain a conversion specifier for each item to be printed. For example, consider the following statement:

```
printf("The %d contestants ate %f berry pies.\n", number,
       pies);
```

*`Control-string`* is the phrase enclosed in double quotes. This particular control string contains two conversion specifiers corresponding to `number` and `pies`—the two items to be displayed. Figure 4.6 shows another example of a `printf()`statement.

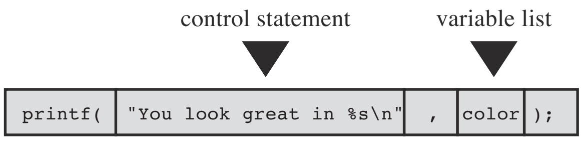


Figure 4.6 Arguments for `printf()`.

Here is another line from the example:

```
printf("The value of pi is %f.\n", PI);
```

This time, the list of items has just one member—the symbolic constant `PI`.

As you can see in Figure 4.7, *`Control-string`* contains two distinct forms of information:

- Characters that are actually printed
- Conversion specifications

**Caution**

Don't forget to use one conversion specification for each item in the list following *`Control-string`*. Woe unto you should you forget this basic requirement! Don't do the following:

```
printf("The score was Squids %d, Slugs %d.\n", score1);
```

Here, there is no value for the second `%d`. The result of this faux pas depends on your system, but at best you will get partial nonsense.

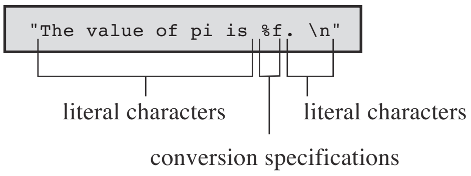


Figure 4.7 Anatomy of a control string.

If you want to print only a phrase, you don't need any conversion specifications. If you just want to print data, you can dispense with the running commentary. Each of the following statements from Listing 4.6 is quite acceptable:

```
printf("Farewell! thou art too dear for my possessing,\n");
printf("%c%d\n", '$', 2 * cost);
```

In the second statement, note that the first item on the print list was a character constant rather than a variable and that the second item is a multiplication. This illustrates that `printf()` uses values, be they variables, constants, or expressions.

Because the `printf()` function uses the `%` symbol to identify the conversion specifications, there is a slight problem if you want to print the `%` sign itself. If you simply use a lone `%` sign, the compiler thinks you have bungled a conversion specification. The way out is simple—just use two `%` symbols, as shown here:

```
pc = 2*6;
printf("Only %d%% of Sally's gribbles were edible.\n", pc);
```

The following output would result:

```
Only 12% of Sally's gribbles were edible.
```

## Conversion Specification Modifiers for `printf()`

You can modify a basic conversion specification by inserting modifiers between the `%` and the defining conversion character. Tables 4.4 and 4.5 list the characters you can place there legally. If you use more than one modifier, they should be in the same order as they appear in Table

4.4. Not all combinations are possible. The table reflects the C99 additions; your implementation may not support all the options shown here.

Table 4.4 **The `printf()` Modifiers**

| **Modifier** | **Meaning** |
|---|---|
| flag | The five flags (`-`, `+`, space, `#`, and `0`) are described in Table 4.5. Zero or more flags may be present. |
| | Example: `"%-10d"`. |
| digit(s) | The minimum field width. A wider field will be used if the printed number or string won't fit in the field. |
| | Example: `"%4d"`. |
| *.digit(s)* | Precision. For `%e`, `%E`, and `%f` conversions, the number of digits to be printed to the right of the decimal. For `%g` and `%G` conversions, the maximum number of significant digits. For `%s` conversions, the maximum number of characters to be printed. For integer conversions, the minimum number of digits to appear; leading zeros are used if necessary to meet this minimum. Using only `.` implies a following zero, so `%.f` is the same as `%.0f`. |
| | Example: `"%5.2f"` prints a `float` in a field five characters wide with two digits after the decimal point. |
| `h` | Used with an integer conversion specifier to indicate a `short int` or `unsigned short int` value. |
| | Examples: `"%hu"`, `"%hx"`, and `"%6.4hd"`. |
| `hh` | Used with an integer conversion specifier to indicate a `signed char` or `unsigned char` value. |
| | Examples: `"%hhu"`, `"%hhx"`, and `"%6.4hhd"`. |
| `j` | Used with an integer conversion specifier to indicate an `intmax_t` or `uintmax_t` value; these are types defined in `stdint.h`. |
| | Examples: `"%jd"` and `"%8jX"`. |
| `l` | Used with an integer conversion specifier to indicate a `long int` or `unsigned long int`. |
| | Examples: `"%ld"` and `"%8lu"`. |
| `ll` | Used with an integer conversion specifier to indicate a `long long int` or `unsigned long long int`. (C99). |
| | Examples: `"%lld"` and `"%8llu"`. |
| `L` | Used with a floating-point conversion specifier to indicate a `long double` value. |
| | Examples: `"%Lf"` and `"%10.4Le"`. |

| Modifier | Meaning |
|---|---|
| `t` | Used with an integer conversion specifier to indicate a `ptrdiff_t` value. This is the type corresponding to the difference between two pointers. (C99). |
| | Examples: `"%td"` and `"%12ti"`. |
| `z` | Used with an integer conversion specifier to indicate a `size_t` value. This is the type returned by `sizeof`. (C99).<br>Examples: `"%zd"` and `"%12zx"`. |

**Note Type Portability**

The `sizeof` operator, recall, returns the size, in bytes, of a type or value. This should be some form of integer, but the standard only provides that it should be an unsigned integer. Thus it could be `unsigned int`, `unsigned long`, or even `unsigned long long`. So, if you were to use `printf()` to display a `sizeof` expression, you might use `%u` on one system, `%lu` one another, and `%llu` on a third. This means you would need to research the correct usage for your system and that you might need to alter your program if you move it to a different system. Well, it would have meant that except that C provides help to make the type more portable. First, the `stddef.h` header file (included when you include `stdio.h`) defines `size_t` to be whatever the type your system uses for `sizeof`; this is called the underlying type. Second, `printf()` uses the `z` modifier to indicate the corresponding type for printing. Similarly, C defines the `ptrdiff_t` type and `t` modifier to indicate whatever underlying signed integer type the system used for the difference between two addresses.

**Note Conversion of `float` Arguments**

There are conversion specifiers to print the floating types `double` and `long double`. However, there is no specifier for `float`. The reason is that `float` values were automatically converted to type `double` before being used in an expression or as an argument under K&R C. ANSI C (or later), in general, does not automatically convert `float` to `double`. To protect the enormous number of existing programs that assume `float` arguments are converted to `double`, however, all `float` arguments to `printf()`—as well as to any other C function not using an explicit prototype—are still automatically converted to `double`. Therefore, under either K&R C or ANSI C, no special conversion specifier is needed for displaying type `float`.

Table 4.5 **The `printf()` Flags**

| Flag | Meaning |
|---|---|
| `-` | The item is left-justified; that is, it is printed beginning at the left of the field. |
| | Example: `"%-20s"`. |

| Flag | Meaning |
|---|---|
| + | Signed values are displayed with a plus sign, if positive, and with a minus sign, if negative. |
| | Example: `"%+6.2f"`. |
| *space* | Signed values are displayed with a leading space (but no sign) if positive and with a minus sign if negative. A + flag overrides a space. |
| | Example: `"% 6.2f"`. |
| # | Use an alternative form for the conversion specification. Produces an initial `0` for the `%o` form and an initial `0x` or `0X` for the `%x` or `%X` form, respectively. For all floating-point forms, # guarantees that a decimal-point character is printed, even if no digits follow. For `%g` and `%G` forms, it prevents trailing zeros from being removed. |
| | Examples: `"%#o"`, `"%#8.0f"`, and `"%+#10.3E"`. |
| `0` | For numeric forms, pad the field width with leading zeros instead of with spaces. This flag is ignored if a – flag is present or if, for an integer form, a precision is specified. Examples: `"%010d"` and `"%08.3f"`. |

## Examples Using Modifiers and Flags

Let's put these modifiers to work, beginning with a look at the effect of the field width modifier on printing an integer. Consider the program in Listing 4.7.

Listing 4.7 **The `width.c` Program**

```
/* width.c -- field widths */
#include <stdio.h>
#define PAGES 959
int main(void)
{
    printf("*%d*\n", PAGES);
    printf("*%2d*\n", PAGES);
    printf("*%10d*\n", PAGES);
    printf("*%-10d*\n", PAGES);

    return 0;
}
```

Listing 4.7 prints the same quantity four times using four different conversion specifications. It uses an asterisk (*) to show you where each field begins and ends. The output looks as follows:

```
*959*
*959*
*       959*
*959       *
```

The first conversion specification is `%d` with no modifiers. It produces a field with the same width as the integer being printed. This is the default option; that is, it's what's printed if you don't give further instructions. The second conversion specification is `%2d`. This should produce a field width of 2, but because the integer is three digits long, the field is expanded automatically to fit the number. The next conversion specification is `%10d`. This produces a field 10 spaces wide, and, indeed, there are seven blanks and three digits between the asterisks, with the number tucked into the right end of the field. The final specification is `%-10d`. It also produces a field 10 spaces wide, and the `-` puts the number at the left end, just as advertised. After you get used to it, this system is easy to use and gives you nice control over the appearance of your output. Try altering the value for `PAGES` to see how different numbers of digits are printed.

Now look at some floating-point formats. Enter, compile, and run the program in Listing 4.8.

Listing 4.8 **The `floats.c` Program**

```
// floats.c -- some floating-point combinations
#include <stdio.h>

int main(void)
{
    const double RENT = 3852.99;  // const-style constant

    printf("*%f*\n", RENT);
    printf("*%e*\n", RENT);
    printf("*%4.2f*\n", RENT);
    printf("*%3.1f*\n", RENT);
    printf("*%10.3f*\n", RENT);
    printf("*%10.3E*\n", RENT);
    printf("*%+4.2f*\n", RENT);
    printf("*%010.2f*\n", RENT);

    return 0;
}
```

This time, the program uses the keyword `const` to create a symbolic constant. The output is

```
*3852.990000*
*3.852990e+03*
*3852.99*
*3853.0*
*  3852.990*
* 3.853E+03*
```

```
*+3852.99*
*0003852.99*
```

The example begins with the default version, `%f`. In this case, there are two defaults—the field width and the number of digits to the right of the decimal. The second default is six digits, and the field width is whatever it takes to hold the number.

Next is the default for `%e`. It prints one digit to the left of the decimal point and six places to the right. We're getting a lot of digits! The cure is to specify the number of decimal places to the right of the decimal, and the next four examples in this segment do that. Notice how the fourth and the sixth examples cause the output to be rounded off. Also, the sixth example uses `E` instead of `e`.

Finally, the + flag causes the result to be printed with its algebraic sign, which is a plus sign in this case, and the `0` flag produces leading zeros to pad the result to the full field width. Note that in the specifier `%010.2f`, the first `0` is a flag, and the remaining digits before the period (`10`) specify the field width.

You can modify the `RENT` value to see how variously sized values are printed. Listing 4.9 demonstrates a few more combinations.

Listing 4.9 **The `flags.c` Program**

```
/* flags.c -- illustrates some formatting flags */
#include <stdio.h>
int main(void)
{
    printf("%x %X %#x\n", 31, 31, 31);
    printf("**%d**% d**% d**\n", 42, 42, -42);
    printf("**%5d**%5.3d**%05d**%05.3d**\n", 6, 6, 6, 6);

    return 0;
}
```

The output looks as follows:

```
1f 1F 0x1f
**42** 42**-42**
**    6**  006**00006**  006**
```

First, `1f` is the hex equivalent of 31. The `x` specifier yields `1f`, and the `X` specifier yields `1F`. Using the # flag provides an initial `0x`.

The second line of output illustrates how using a space in the specifier produces a leading space for positive values, but not for negative values. This can produce a pleasing output because positive and negative values with the same number of significant digits are printed with the same field widths.

The third line illustrates how using a precision specifier (`%5.3d`) with an integer form produces enough leading zeros to pad the number to the minimum value of digits (three, in this case). Using the `0` flag, however, pads the number with enough leading zeros to fill the whole field width. Finally, if you provide both the `0` flag and the precision specifier, the `0` flag is ignored.

Now let's examine some of the string options. Consider the example in Listing 4.10.

Listing 4.10 **The `stringf.c` Program**

```
/* stringf.c -- string formatting */
#include <stdio.h>
#define BLURB "Authentic imitation!"
int main(void)
{
    printf("[%2s]\n", BLURB);
    printf("[%24s]\n", BLURB);
    printf("[%24.5s]\n", BLURB);
    printf("[%-24.5s]\n", BLURB);

    return 0;
}
```

Here is the output:

```
[Authentic imitation!]
[    Authentic imitation!]
[                   Authe]
[Authe                   ]
```

Notice how, for the `%2s` specification, the field is expanded to contain all the characters in the string. Also notice how the precision specification limits the number of characters printed. The `.5` in the format specifier tells `printf()` to print just five characters. Again, the `-` modifier left-justifies the text.

## Using What You Just Learned

Okay, you've seen some examples. Now, how would you set up a statement to print something having the following form?

```
The NAME family just may be $XXX.XX dollars richer!
```

Here, `NAME` and `XXX.XX` represent values that will be supplied by variables in the program—say, `name[40]` and `cash`.

One solution is

```
printf("The %s family just may be $%.2f richer!\n",name,cash);
```

## What Does a Conversion Specification Convert?

Let's take a closer look at what a conversion specification converts. It converts a value stored in the computer in some binary format to a series of characters (a string) to be displayed. For example, the number 76 may be stored internally as binary 01001100. The `%d` conversion specifier converts this to the characters `7` and `6`, displaying `76`. The `%x` conversion converts the same value (`01001100`) to the hexadecimal representation `4c`. The `%c` converts the same value to the character representation `L`.

The term *conversion* is probably somewhat misleading because it might suggest that the original value is replaced with a converted value. Conversion specifications are really translation specifications; `%d` means "translate the given value to a decimal integer text representation and print the representation."

### Mismatched Conversions

Naturally, you should match the conversion specification to the type of value being printed. Often, you have choices. For instance, if you want to print a type `int` value, you can use `%d`, `%x`, or `%o`. All these specifiers assume that you are printing a type `int` value; they merely provide different representations of the value. Similarly, you can use `%f`, `%e`, or `%g` to represent a type `double` value.

What if you mismatch the conversion specification to the type? You've seen in the preceding chapter that mismatches can cause problems. This is a very important point to keep in mind, so Listing 4.11 shows some more examples of mismatches within the integer family.

Listing 4.11 **The `intconv.c` Program**

```
/* intconv.c -- some mismatched integer conversions */
#include <stdio.h>
#define PAGES 336
#define WORDS 65618
int main(void)
{
    short num = PAGES;
    short mnum = -PAGES;

    printf("num as short and unsigned short:  %hd %hu\n", num,
            num);
    printf("-num as short and unsigned short: %hd %hu\n", mnum,
            mnum);
    printf("num as int and char: %d %c\n", num, num);
    printf("WORDS as int, short, and char: %d %hd %c\n",
            WORDS, WORDS, WORDS);
   return 0;
}
```

Our system produces the following results:

```
num as short and unsigned short:  336 336
-num as short and unsigned short: -336 65200
num as int and char: 336 P
WORDS as int, short, and char: 65618 82 R
```

Looking at the first line, you can see that both `%hd` and `%hu` produce `336` as output for the variable `num`; no problem there. On the second line, the `%u` (unsigned) version of `mnum` came out as `65200`, however, not as the `336` you might have expected; this results from the way that signed `short int` values are represented on our reference system. First, they are 2 bytes in size. Second, the system uses a method called the *two's complement* to represent signed integers. In this method, the numbers 0 to 32767 represent themselves, and the numbers 32768 to 65535 represent negative numbers, with 65535 being –1, 65534 being –2, and so forth. Therefore, `–336` is represented by `65536 - 336`, or `65200`. So 65200 represents –336 when interpreted as a signed `int` and represents 65200 when interpreted as an unsigned `int`. Be wary! One number can be interpreted as two different values. Not all systems use this method to represent negative integers. Nonetheless, there is a moral: Don't expect a `%u` conversion to simply strip the sign from a number.

The third line shows what happens if you try to convert a value greater than 255 to a character. On this system, a `short int` is 2 bytes and a `char` is 1 byte. When `printf()` prints 336 using `%c`, it looks at only 1 byte out of the 2 used to hold 336. This truncation (see Figure 4.8) amounts to dividing the integer by 256 and keeping just the remainder. In this case, the remainder is 80, which is the ASCII value for the character *P*. More technically, you can say that the number is interpreted *modulo 256*, which means using the remainder when the number is divided by 256.

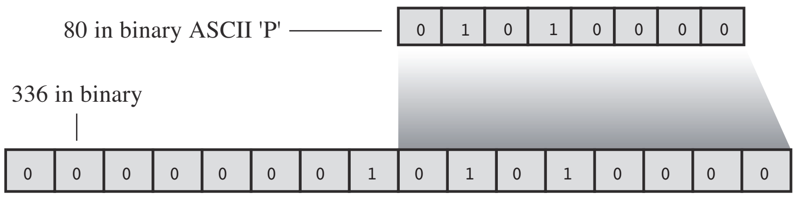


Figure 4.8 Reading 336 as a character.

Finally, we tried printing an integer (65618) larger than the maximum `short int` (32767) allowed on our system. Again, the computer does its modulo thing. The number 65618, because of its size, is stored as a 4-byte `int` value on our system. When we print it using the `%hd` specification, `printf()` uses only the last 2 bytes. This corresponds to using the remainder after dividing by 65536. In this case, the remainder is 82. A remainder between 32767 and 65536 would be printed as a negative number because of the way negative numbers are stored. Systems with different integer sizes would have the same general behavior, but with different numerical values.

When you start mixing integer and floating types, the results are more bizarre. Consider, for example, Listing 4.12.

Listing 4.12 **The `floatcnv.c` Program**

```
/* floatcnv.c -- mismatched floating-point conversions */
#include <stdio.h>
int main(void)
{
    float n1 = 3.0;
    double n2 = 3.0;
    long n3 = 2000000000;
    long n4 = 1234567890;

    printf("%.1e %.1e %.1e %.1e\n", n1, n2, n3, n4);
    printf("%ld %ld\n", n3, n4);
    printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);

    return 0;
}
```

On one system, Listing 4.12 produces the following output:

```
3.0e+00 3.0e+00 3.1e+46 1.7e+266
2000000000 1234567890
0 1074266112 0 1074266112
```

The first line of output shows that using a `%e` specifier does not convert an integer to a floating-point number. Consider, for example, what happens when you try to print `n3` (type `long`) using the `%e` specifier. First, the `%e` specifier causes `printf()` to expect a type `double` value, which is an 8-byte value on this system. When `printf()` looks at `n3`, which is a 4-byte value on this system, it also looks at the adjacent 4 bytes. Therefore, it looks at an 8-byte unit in which the actual n3 is embedded. Second, it interprets the bits in this unit as a floating-point number. Some bits, for example, would be interpreted as an exponent. So even if `n3` had the correct number of bits, they would be interpreted differently under `%e` than under `%ld`. The net result is nonsense.

The first line also illustrates what we mentioned earlier—that `float` is converted to `double` when used as arguments to `printf()`. On this system, `float` is 4 bytes, but `n1` was expanded to 8 bytes so that `printf()` would display it correctly.

The second line of output shows that `printf()` can print `n3` and `n4` correctly if the correct specifier is used.

The third line of output shows that even the correct specifier can produce phony results if the `printf()` statement has mismatches elsewhere. As you might expect, trying to print a floating-point value with an `%ld` specifier fails, but here, trying to print a type `long` using `%ld` fails! The problem lies in how C passes information to a function. The exact details of this failure are implementation dependent, but the sidebar "Passing Arguments" discusses a representative system.

## Passing Arguments

The mechanics of argument passing depend on the implementation. This is how argument passing works on one system. The function call looks as follows:

```
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

This call tells the computer to hand over the values of the variables `n1`, `n2`, `n3`, and `n4` to the computer. Here's one common way that's accomplished. The program places the values in an area of memory called the *stack*. When the computer puts these values on the stack, it is guided by the types of the variables, not by the conversion specifiers. Consequently, for `n1`, it places 8 bytes on the stack (`float` is converted to `double`). Similarly, it places 8 more bytes for `n2`, followed by 4 bytes each for `n3` and `n4`. Then control shifts to the `printf()` function. This function reads the values off the stack but, when it does so, it reads them according to the conversion specifiers. The `%ld` specifier indicates that `printf()` should read 4 bytes, so `printf()` reads the first 4 bytes in the stack as its first value. This is just the first half of `n1`, and it is interpreted as a `long` integer. The next `%ld` specifier reads 4 more bytes; this is just the second half of `n1` and is interpreted as a second `long` integer (see Figure 4.9). Similarly, the third and fourth instances of `%ld` cause the first and second halves of `n2` to be read and to be interpreted as two more `long` integers, so although we have the correct specifiers for `n3` and `n4`, `printf()` is reading the wrong bytes.

```
float n1;  /* passed as type double */
double n2;
long n3, n4;
...
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

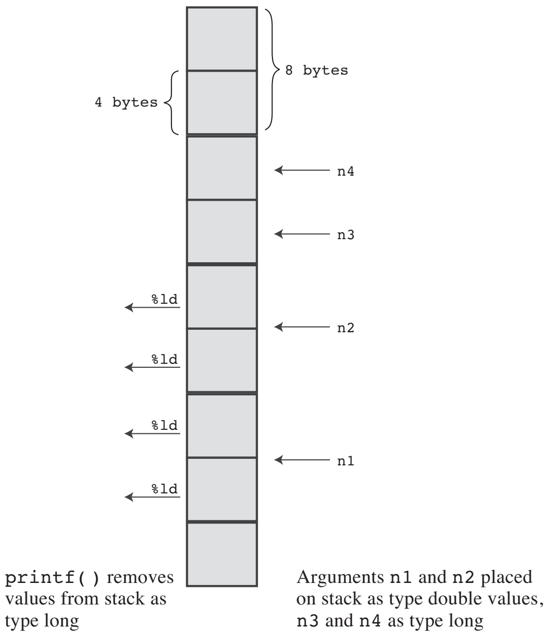

8 bytes
4 bytes
n4
n3
%ld
n2
%ld
%ld
n1
%ld
printf() removes values from stack as type long
Arguments n1 and n2 placed on stack as type double values, n3 and n4 as type long

Figure 4.9 Passing arguments.

## The Return Value of `printf()`

As mentioned in Chapter 2, a C function generally has a return value. This is a value that the function computes and returns to the calling program. For example, the C library contains a `sqrt()` function that takes a number as an argument and returns its square root. The return value can be assigned to a variable, can be used in a computation, can be passed as an argument—in short, it can be used like any other value. The `printf()` function also has a return value; it returns the number of characters it printed. If there is an output error, `printf()` returns a negative value. (Some ancient versions of `printf()` have different return values.)

The return value for `printf()` is incidental to its main purpose of printing output, and it usually isn't used. One reason you might use the return value is to check for output errors. This is more commonly done when writing to a file rather than to a screen. If a full CD or DVD prevented writing from taking place, you could then have the program take some appropriate action, such as beeping the terminal for 30 seconds. However, you have to know about the `if` statement before doing that sort of thing. The simple example in Listing 4.13 shows how you can determine the return value.

Listing 4.13 **The `prntval.c` Program**

```
/* prntval.c -- finding printf()'s return value */
#include <stdio.h>
int main(void)
{
    int bph2o = 212;
    int rv;

    rv = printf("%d F is water's boiling point.\n", bph2o);
    printf("The printf() function printed %d characters.\n",
             rv);
    return 0;
}
```

The output is as follows:

```
212 F is water's boiling point.
The printf() function printed 32 characters.
```

First, the program used the form `rv = printf(...);` to assign the return value to `rv`. This statement therefore performs two tasks: printing information and assigning a value to a variable. Second, note that the count includes all the printed characters, including the spaces and the unseen newline character.

## Printing Long Strings

Occasionally, `printf()` statements are too long to put on one line. Because C ignores whitespace (spaces, tabs, newlines) except when used to separate elements, you can spread a statement over several lines, as long as you put your line breaks between elements. For example, Listing 4.13 used two lines for a statement.

```
printf("The printf() function printed %d characters.\n",
          rv);
```

The line is broken between the comma element and `rv`. To show a reader that the line was being continued, the example indents the `rv`. C ignores the extra spaces.

However, you cannot break a quoted string in the middle. Suppose you try something like the following:

```
printf("The printf() function printed %d
          characters.\n", rv);
```

C will complain that you have an illegal character in a string constant. You can use `\n` in a string to symbolize the newline character, but you can't have the actual newline character generated by the Enter (or Return) key in a string.

If you do have to split a string, you have three choices, as shown in Listing 4.14.

Listing 4.14 **The `longstrg.c` Program**

```
/* longstrg.c -- printing long strings */
#include <stdio.h>
int main(void)
{
    printf("Here's one way to print a ");
    printf("long string.\n");
    printf("Here's another way to print a \
long string.\n");
    printf("Here's the newest way to print a "
          "long string.\n");      /* ANSI C */
    return 0;
}
```

Here is the output:

```
Here's one way to print a long string.
Here's another way to print a long string.
Here's the newest way to print a long string.
```

Method 1 is to use more than one `printf()` statement. Because the first string printed doesn't end with a `\n` character, the second string continues where the first ends.

Method 2 is to terminate the end of the first line with a backslash/return combination. This causes the text onscreen to start a new line without a newline character being included in the string. The effect is to continue the string over to the next line. However, the next line has to start at the far left, as shown. If you indent that line, say, five spaces, those five spaces become part of the string.

Method 3, which ANSI C introduced, is string concatenation. If you follow one quoted string constant with another, separated only by whitespace, C treats the combination as a single string, so the following three forms are equivalent:

```
printf("Hello, young lovers, wherever you are.");
printf("Hello, young "    "lovers" ", wherever you are.");
printf("Hello, young lovers"
       ", wherever you are.");
```

With all these methods, you should include any required spaces in the strings: `"young" "lovers"` becomes `"younglovers"`, but the combination `"young " "lovers"` is `"young lovers"`.

## Using `scanf()`

Now let's go from output to input and examine the `scanf()` function. The C library contains several input functions, and `scanf()` is the most general of them, because it can read a variety of formats. Of course, input from the keyboard is text because the keys generate text characters: letters, digits, and punctuation. When you want to enter, say, the integer 2014, you type the characters `2 0 1` and `4`. If you want to store that as a numerical value rather than as a string, your program has to convert the string character-by-character to a numerical value; that is what `scanf()` does! It converts string input into various forms: integers, floating-point numbers, characters, and C strings. It is the inverse of `printf()`, which converts integers, floating-point numbers, characters, and C strings to text that is to be displayed onscreen.

Like `printf()`, `scanf()` uses a control string followed by a list of arguments. The control string indicates the destination data types for the input stream of characters. The chief difference is in the argument list. The `printf()` function uses variable names, constants, and expressions. The `scanf()` function uses pointers to variables. Fortunately, you don't have to know anything about pointers to use the function. Just remember these simple rules:

- If you use `scanf()` to read a value for one of the basic variable types we've discussed, precede the variable name with an `&`.
- If you use `scanf()` to read a string into a character array, don't use an `&`.

Listing 4.15 presents a short program illustrating these rules.

**Listing 4.15 The `input.c` Program**

```
// input.c -- when to use &
#include <stdio.h>
int main(void)
{
    int age;             // variable
    float assets;        // variable
    char pet[30];        // string

    printf("Enter your age, assets, and favorite pet.\n");
    scanf("%d %f", &age, &assets); // use the & here
    scanf("%s", pet);              // no & for char array
    printf("%d $%.2f %s\n", age, assets, pet);

    return 0;
}
```

Here is a sample exchange:

```
Enter your age, assets, and favorite pet.
38
92360.88 llama
38 $92360.88 llama
```

The `scanf()` function uses whitespace (newlines, tabs, and spaces) to decide how to divide the input into separate fields. It matches up consecutive conversion specifications to consecutive fields, skipping over the whitespace in between. Note how this sample run spread the input over two lines. You could just as well have used one or five lines, as long as you had at least one newline, space, or tab between each entry:

```
Enter your age, assets, and favorite pet.
  42

     2121.45

    guppy
42 $2121.45 guppy
```

The only exception to this is the `%c` specification, which reads the very next character, even if that character is whitespace. We'll return to this topic in a moment.

The `scanf()` function uses pretty much the same set of conversion-specification characters as `printf()` does. The main difference is that `printf()` uses `%f`, `%e`, `%E`, `%g`, and `%G` for both type `float` and type `double`, whereas `scanf()` uses them just for type `float`, requiring the `l` modifier for `double`. Table 4.6 lists the main conversion specifiers as described in the C99 standard.

Table 4.6 **ANSI C Conversion Specifiers for `scanf()`**

| **Conversion Specifier** | **Meaning** |
|---|---|
| `%c` | Interpret input as a character. |
| `%d` | Interpret input as a signed decimal integer. |

| Conversion Specifier | Meaning |
|---|---|
| `%e`, `%f`, `%g`, `%a` | Interpret input as a floating-point number (`%a` is C99). |
| `%E`, `%F`, `%G`, `%A` | Interpret input as a floating-point number (`%A` is C99). |
| `%i` | Interpret input as a signed decimal integer. |
| `%o` | Interpret input as a signed octal integer. |
| `%p` | Interpret input as a pointer (an address). |
| `%s` | Interpret input as a string. Input begins with the first non-whitespace character and includes everything up to the next whitespace character. |
| `%u` | Interpret input as an unsigned decimal integer. |
| `%x`, `%X` | Interpret input as a signed hexadecimal integer. |

You also can use modifiers in the conversion specifiers shown in Table 4.6. The modifiers go between the percent sign and the conversion letter. If you use more than one in a specifier, they should appear in the same order as shown in Table 4.7.

Table 4.7 **Conversion Modifiers for `scanf()`**

| Modifier | Meaning |
|---|---|
| `*` | Suppress assignment (see text). |
| | Example: `"%*d"`. |
| digit(s) | Maximum field width. Input stops when the maximum field width is reached or when the first whitespace character is encountered, whichever comes first. |
| | Example: `"%10s"`. |
| `hh` | Read an integer as a `signed char` or `unsigned char`. |
| | Examples: `"%hhd"` `"%hhu"`. |
| `ll` | Read an integer as a `long long` or unsigned `long long (C99)`. |
| | Examples: `"%lld"` `"%llu"`. |
| `h`, `l`, or `L` | `"%hd"` and `"%hi"` indicate that the value will be stored in a short `int`. `"%ho"`, `"%hx"`, and `"%hu"` indicate that the value will be stored in an `unsigned short int`. `"%ld"` and `"%li"` indicate that the value will be stored in a `long`. `"%lo"`, `"%lx"`, and `"%lu"` indicate that the value will be stored in `unsigned long`. `"%le"`, `"%lf"`, and `"%lg"` indicate that the value will be stored in type `double`. Using `L` instead of `l` with `e`, `f`, and `g` indicates that the value will be stored in type `long double`. In the absence of these modifiers, `d`, `i`, `o`, and `x` indicate type `int`, and `e`, `f`, and `g` indicate type `float`. |

| Modifier | Meaning |
|---|---|
| `j` | When followed by an integer specifier, indicates using the `intmax_t` or `uintmax_t` type (C99). |
| | Examples: `"%jd" "%ju"`. |
| `z` | When followed by an integer specifier, indicates using the type returned by `sizeof` (C99). |
| | Examples: `"%zd" "%zo"`. |
| `t` | When followed by an integer specifier, indicates using the type used to represent the difference between two pointers (C99). Examples: `"%td" "%tx"`. |

As you can see, using conversion specifiers can be involved, and these tables have omitted some of the features. The omitted features primarily facilitate reading selected data from highly formatted sources, such as punched cards or other data records. Because this book uses `scanf()` primarily as a convenient means for feeding data to a program interactively, it won't discuss the more esoteric features.

### The `scanf()` View of Input

Let's look in more detail at how `scanf()` reads input. Suppose you use a `%d` specifier to read an integer. The `scanf()` function begins reading input a character at a time. It skips over whitespace characters (spaces, tabs, and newlines) until it finds a non-whitespace character. Because it is attempting to read an integer, `scanf()` expects to find a digit character or, perhaps, a sign (+ or –). If it finds a digit or a sign, it saves that character and then reads the next character. If that is a digit, it saves the digit and reads the next character. `scanf()` continues reading and saving characters until it encounters a nondigit. It then concludes that it has reached the end of the integer. `scanf()` places the nondigit back into the input. This means that the next time the program goes to read input, it starts at the previously rejected, nondigit character. Finally, `scanf()` computes the numerical value corresponding to the digits (and possible sign) it read and places that value in the specified variable.

If you use a field width, `scanf()` halts at the field end or at the first whitespace, whichever comes first.

What if the first non-whitespace character is, say, an `A` instead of a digit? Then `scanf()` stops right there and places the `A` (or whatever) back in the input. No value is assigned to the specified variable, and the next time the program reads input, it starts at the `A` again. If your program has only `%d` specifiers, `scanf()` will never get past that `A`. Also, if you use a `scanf()` statement with several specifiers, C requires the function to stop reading input at the first failure.

Reading input using the other numeric specifiers works much the same as the `%d` case. The main difference is that `scanf()` may recognize more characters as being part of the number. For instance, the `%x` specifier requires that `scanf()` recognize the hexadecimal digits a–f and A–F. Floating-point specifiers require `scanf()` to recognize decimal points, e-notation, and the new p-notation.

If you use an `%s` specifier, any character other than whitespace is acceptable, so `scanf()` skips whitespace to the first non-whitespace character and then saves up non-whitespace characters until hitting whitespace again. This means that `%s` results in `scanf()` reading a single word—that is, a string with no whitespace in it. If you use a field width, `scanf()` stops at the end of the field or at the first whitespace, whichever comes first. You can't use the field width to make `scanf()` read more than one word for one `%s` specifier. A final point: When `scanf()` places the string in the designated array, it adds the terminating `'\0'` to make the array contents a C string.

If you use a `%c` specifier, all input characters are fair game. If the next input character is a space or a newline, a space or a newline is assigned to the indicated variable; whitespace is not skipped.

Actually, `scanf()` is not the most commonly used input function in C. It is featured here because of its versatility (it can read all the different data types), but C has several other input functions, such as `getchar()` and `fgets()`, that are better suited for specific tasks, such as reading single characters or reading strings containing spaces. We will cover some of these functions in Chapter 7, "C Control Statements: Branching and Jumps"; Chapter 11, "Character Strings and String Functions"; and Chapter 13, "File Input/Output." In the meantime, if you need an integer, decimal fraction, a character, or a string, you can use `scanf()`.

### Regular Characters in the Format String

The `scanf()` function does enable you to place ordinary characters in the format string. Ordinary characters other than the space character must be matched exactly by the input string. For example, suppose you accidentally place a comma between two specifiers:

```
scanf("%d,%d", &n, &m);
```

The `scanf()` function interprets this to mean that you will type a number, type a comma, and then type a second number. That is, you would have to enter two integers as follows:

```
88,121
```

Because the comma comes immediately after the `%d` in the format string, you would have to type it immediately after the `88`. However, because `scanf()` skips over whitespace preceding an integer, you could type a space or newline after the comma when entering the input. That is,

```
88, 121
```

and

```
88,
121
```

also would be accepted.

A space in the format string means to skip over any whitespace before the next input item. For instance, the statement

```
scanf("%d ,%d", &n, &m);
```

would accept any of the following input lines:

```
88,121
88  ,121
88 ,  121
```

Note that the concept of "any whitespace" includes the special cases of no whitespace.

Except for `%c`, the specifiers automatically skip over whitespace preceding an input value, so `scanf("%d%d", &n, &m)` behaves the same as `scanf("%d %d", &n, &m)`. For `%c`, adding a space character to the format string does make a difference. For example, if `%c` is preceded by a space in the format string, `scanf()` does skip to the first non-whitespace character. That is, the command `scanf("%c", &ch)` reads the first character encountered in input, and `scanf(" %c", &ch)` reads the first non-whitespace character encountered.

### The `scanf()` Return Value

The `scanf()` function returns the number of items that it successfully reads. If it reads no items, which happens if you type a nonnumeric string when it expects a number, `scanf()` returns the value `0`. It returns `EOF` when it detects the condition known as "end of file." (`EOF` is a special value defined in the `stdio.h` file. Typically, a `#define` directive gives `EOF` the value `–1`.) We'll discuss end of file in Chapter 6, "C Control Statements: Looping," and make use of `scanf()`'s return value later in the book. After you learn about `if` statements and `while` statements, you can use the `scanf()` return value to detect and handle mismatched input.

## The `*` Modifier with `printf()` and `scanf()`

Both `printf()` and `scanf()` can use the `*` modifier to modify the meaning of a specifier, but they do so in dissimilar fashions. First, let's see what the `*` modifier can do for `printf()`.

Suppose that you don't want to commit yourself to a field width in advance but rather you want the program to specify it. You can do this by using `*` instead of a number for the field width, but you also have to add an argument to tell what the field width should be. That is, if you have the conversion specifier `%*d`, the argument list should include a value for `*` *and* a value for `d`. The technique also can be used with floating-point values to specify the precision as well as the field width. Listing 4.16 is a short example showing how this works.

Listing 4.16 **The `varwid.c` Program**

```
/* varwid.c -- uses variable-width output field */
#include <stdio.h>
```

```
int main(void)
{
    unsigned width, precision;
    int number = 256;
    double weight = 242.5;

    printf("Enter a field width:\n");
    scanf("%d", &width);
    printf("The number is :%*d:\n", width, number);
    printf("Now enter a width and a precision:\n");
    scanf("%d %d", &width, &precision);
    printf("Weight = %*.*f\n", width, precision, weight);
    printf("Done!\n");

    return 0;
}
```

The variable `width` provides the `field width`, and `number` is the number to be printed. Because the `*` precedes the `d` in the specifier, `width` comes before `number` in `printf()`'s argument list. Similarly, `width` and `precision` provide the formatting information for printing `weight`. Here is a sample run:

```
Enter a field width:
6
The number is :   256:
Now enter a width and a precision:
8 3
Weight =  242.500
Done!
```

Here, the reply to the first question was `6`, so `6` was the field width used. Similarly, the second reply produced a width of `8` with `3` digits to the right of the decimal. More generally, a program could decide on values for these variables after looking at the value of `weight`.

The `*` serves quite a different purpose for `scanf()`. When placed between the `%` and the specifier letter, it causes that function to skip over corresponding input. Listing 4.17 provides an example.

### Listing 4.17 The `skip2.c` Program

```
/* skiptwo.c -- skips over first two integers of input */
#include <stdio.h>
int main(void)
{
    int n;

    printf("Please enter three integers:\n");
    scanf("%*d %*d %d", &n);
    printf("The last integer was %d\n", n);

    return 0;
}
```

The `scanf()` instruction in Listing 4.17 says, “Skip two integers and copy the third into `n`.” Here is a sample run:

```
Please enter three integers:
2013 2014 2015
The last integer was 2015
```

This skipping facility is useful if, for example, a program needs to read a particular column of a file that has data arranged in uniform columns.

## Usage Tips for `printf()`

Specifying fixed field widths is useful when you want to print columns of data. Because the default field width is just the width of the number, the repeated use of, say,

```
printf("%d %d %d\n", val1, val2, val3);
```

produces ragged columns if the numbers in a column have different sizes. For example, the output could look like the following:

```
12 234 1222
4 5 23
22334 2322 10001
```

(This assumes that the value of the variables has been changed between `print` statements.)

The output can be cleaned up by using a sufficiently large fixed field width. For example, using

```
printf("%9d %9d %9d\n", val1, val2, val3);
```

yields the following:

```
   12        234       1222
    4          5         23
22334       2322      10001
```

Leaving a blank between one conversion specification and the next ensures that one number never runs into the next, even if it overflows its own field. This is so because the regular characters in the control string, including spaces, are printed.

On the other hand, if a number is to be embedded in a phrase, it is often convenient to specify a field as small or smaller than the expected number width. This makes the number fit in without unnecessary blanks. For example,

```
printf("Count Beppo ran %.2f miles in 3 hours.\n", distance);
```

might produce

```
Count Beppo ran 10.22 miles in 3 hours.
```

Changing the conversion specification to `%10.2f` would give you the following:

```
Count Beppo ran      10.22 miles in 3 hours.
```

**Locale Choices**

The United States and many other parts of the world use a period to separate the integer part of a decimal value from the fractional part, as in 3.14159. But many other parts of the world use a comma instead, as in 3,14159. You may have noticed that the `printf()` and `scanf()` specifiers don't seem to offer the comma format. But C hasn't ignored the rest of the world. As outlined in Appendix B, Section V, "The Standard ANSI C Library with C99 Additions," C supports the concept of a *locale*. This gives a C program the option of choosing a particular locale. For example, it might specify a Netherlands locale, and `printf()` and `scanf()` would use the local convention (a comma, in this case) when displaying and reading floating-point values. Also, once you specified that environment, you would use the comma convention for numbers appearing in your code:

```
double pi = 3,14159;  // Netherlands locale
```

The C standard requires but two locales: `"C"` and `""`. By default, programs use the `"C"` locale which, basically, is U.S. usage. The `""` locale stands for a local locale in use on your system. In principle, it could be the same as the `"C"` locale. In practice, operating systems such as Unix, Linux, and Windows offer long lists of locale choices. However, they might not offer the same lists.

## Key Concepts

The C `char` type represents a single character. To represent a sequence of characters, C uses the character string. One form of string is the character constant, in which the characters are enclosed in double quotation marks; `"Good luck, my friend"` is an example. You can store a string in a character array, which consists of adjacent bytes in memory. Character strings, whether expressed as a character constant or stored in a character array, are terminated by a hidden character called the *null* character.

It's a good idea to represent numerical constants in a program symbolically, either by using `#define` or the keyword `const`. Symbolic constants make a program more readable and easier to maintain and modify.

The standard C input and output functions `scanf()` and `printf()` use a system in which you have to match type specifiers in the first argument to values in the subsequent arguments. Matching, say, an `int` specifier such as `%d` to a `float` value produces odd results. You have to exert care to match the number and type of specifiers to the rest of the function arguments. For `scanf()`, remember to prefix variables' names with the address operator (`&`).

Whitespace characters (tabs, spaces, and newlines) play a critical role in how `scanf()` views input. Except when in the `%c` mode (which reads just the next character), `scanf()` skips over whitespace characters to the first non-whitespace character when reading input. It then keeps reading characters either until encountering whitespace or until encountering a character that doesn't fit the type being read. Let's consider what happens if we feed the identical input line to several different `scanf()` input modes. Start with the following input line:

```
-13.45e12#  0
```

First, suppose we use the `%d` mode; `scanf()` would read the three characters (`−13`) and stop at the period, leaving the period as the next input character. `scanf()` then would convert the character sequence `−13` into the corresponding integer value and store that value in the destination `int` variable. Next, reading the same line in the `%f` mode, `scanf()` would read the `−13.45E12` characters and stop at the # symbol, leaving it as the next input character. It then would convert the character sequence `−13.45E12` into the corresponding floating-point value and store that value in the destination `float` variable. Reading the same line in the `%s` mode, `scanf()` would read `−13.45E12#`, stopping at the space, leaving it as the next input character. It then would store the character codes for these 10 characters into the destination character array, appending a null character at the end. Finally, reading the same line using the `%c` specifier, `scanf()` would read and store the first character, in this case a space.

## Summary

A string is a series of characters treated as a unit. In C, strings are represented by a series of characters terminated by the null character, which is the character whose ASCII code is 0. Strings can be stored in character arrays. An array is a series of items, or elements, all of the same type. To declare an array called `name` that has 30 elements of type `char`, do the following:

```
char name[30];
```

Be sure to allot a number of elements sufficient to hold the entire string, including the null character.

String constants are represented by enclosing the string in double quotes: `"This is an example of a string"`.

The `strlen()` function (declared in the `string.h` header file) can be used to find the length of a string (not counting the terminating null character). The `scanf()` function, when used with the `%s` specifier, can be used to read in single-word strings.

The C preprocessor searches a source code program for preprocessor directives, which begin with the # symbol, and acts upon them before the program is compiled. The `#include` directive causes the processor to add the contents of another file to your file at the location of the directive. The `#define` directive lets you establish manifest constants—that is, symbolic representations for constants. The `limits.h` and `float.h` header files use `#define` to define a set of constants representing various properties of integer and floating-point types. You also can use the `const` modifier to create symbolic constants.

The `printf()` and `scanf()` functions provide versatile support for input and output. Each uses a control string containing embedded conversion specifiers to indicate the number and type of data items to be read or printed. Also, you can use the conversion specifiers to control the appearance of the output: field widths, decimal places, and placement within a field.

## Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. Run Listing 4.1 again, but this time give your first and last name when it asks you for your first name. What happens? Why?

2. Assuming that each of the following examples is part of a complete program, what will each one print?

   a.

   ```
   printf("He sold the painting for $%2.2f.\n", 2.345e2);
   ```

   b.

   ```
   printf("%c%c%c\n", 'H', 105, '\41');
   ```

   c.

   ```
   #define Q "His Hamlet was funny without being vulgar."

      printf("%s\nhas %d characters.\n", Q, strlen(Q));
   ```

   d.

   ```
   printf("Is %2.2e the same as %2.2f?\n", 1201.0, 1201.0);
   ```

3. In Question 2c, what changes could you make so that string `Q` is printed out enclosed in double quotation marks?

4. It's find the error time!

   ```
   define B booboo
   define X 10
   main(int)
   {
      int age;
      char name;
   ```

```
   printf("Please enter your first name.");
   scanf("%s", name);
   printf("All right, %c, what's your age?\n", name);
   scanf("%f", age);
   xp = age + X;
   printf("That's a %s! You must be at least %d.\n", B, xp);
   rerun 0;
}
```

5. Suppose a program starts as follows:

```
#define BOOK "War and Peace"
int main(void)
{
   float cost =12.99;
   float percent = 80.0;
```

   Construct a `printf()` statement that uses `BOOK`, `cost`, and `percent` to print the following:

```
This copy of "War and Peace" sells for $12.99.
That is 80% of list.
```

6. What conversion specification would you use to print each of the following?
   a. A decimal integer with a field width equal to the number of digits
   b. A hexadecimal integer in the form 8A in a field width of 4
   c. A floating-point number in the form 232.346 with a field width of 10
   d. A floating-point number in the form 2.33e+002 with a field width of 12
   e. A string left-justified in a field of width 30

7. Which conversion specification would you use to print each of the following?
   a. An `unsigned long` integer in a field width of 15
   b. A hexadecimal integer in the form 0x8a in a field width of 4
   c. A floating-point number in the form 2.33E+02 that is left-justified in a field width of 12
   d. A floating-point number in the form +232.346 in a field width of 10
   e. The first eight characters of a string in a field eight characters wide

8. What conversion specification would you use to print each of the following?
   a. A decimal integer having a minimum of four digits in a field width of 6
   b. An octal integer in a field whose width will be given in the argument list
   c. A character in a field width of 2
   d. A floating-point number in the form +3.13 in a field width equal to the number of characters in the number
   e. The first five characters in a string left-justified in a field of width 7

9. For each of the following input lines, provide a `scanf()` statement to read it. Also declare any variables or arrays used in the statement.
   a. 101
   b. 22.32 8.34E−09
   c. linguini
   d. catch 22
   e. catch 22 (but skip over catch)

10. What is whitespace?

11. What's wrong with the following statement and how can you fix it?

```
printf("The double type is %z bytes..\n", sizeof (double));
```

12. Suppose that you would rather use parentheses than braces in your programs. How well would the following work?

```
#define ( {
#define ) }
```

## Programming Exercises

1. Write a program that asks for your first name, your last name, and then prints the names in the format *last, first*.

2. Write a program that requests your first name and does the following with it:
   a. Prints it enclosed in double quotation marks
   b. Prints it in a field 20 characters wide, with the whole field in quotes and the name at the right end of the field

   c. Prints it at the left end of a field 20 characters wide, with the whole field enclosed in quotes

   d. Prints it in a field three characters wider than the name

3. Write a program that reads in a floating-point number and prints it first in decimal-point notation and then in exponential notation. Have the output use the following formats (the number of digits shown in the exponent may be different for your system):

   a. The input is `21.3` or `2.1e+001`.

   b. The input is `+21.290` or `2.129E+001`.

4. Write a program that requests your height in inches and your name, and then displays the information in the following form:

```
Dabney, you are 6.208 feet tall
```

   Use type `float`, and use `/` for division. If you prefer, request the height in centimeters and display it in meters.

5. Write a program that requests the download speed in megabits per second (Mbs) and the size of a file in megabytes (MB). The program should calculate the download time for the file. Note that in this context one byte is eight bits. Use type `float`, and use `/` for division. The program should report all three values (download speed, file size, and download time) showing two digits to the right of the decimal point, as in the following:

```
At 18.12 megabits per second, a file of 2.20 megabytes
downloads in 0.97 seconds.
```

6. Write a program that requests the user's first name and then the user's last name. Have it print the entered names on one line and the number of letters in each name on the following line. Align each letter count with the end of the corresponding name, as in the following:

```
Melissa Honeybee
      7        8
```

   Next, have it print the same information, but with the counts aligned with the beginning of each name.

```
Melissa Honeybee
7       8
```

7. Write a program that sets a type `double` variable to 1.0/3.0 and a type `float` variable to 1.0/3.0. Display each result three times—once showing four digits to the right of the decimal, once showing 12 digits to the right of the decimal, and once showing 16 digits

to the right of the decimal. Also have the program include `float.h` and display the values of `FLT_DIG` and `DBL_DIG`. Are the displayed values of 1.0/3.0 consistent with these values?

8. Write a program that asks the user to enter the number of miles traveled and the number of gallons of gasoline consumed. It should then calculate and display the miles-per-gallon value, showing one place to the right of the decimal. Next, using the fact that one gallon is about 3.785 liters and one mile is about 1.609 kilometers, it should convert the mile-per-gallon value to a liters-per-100-km value, the usual European way of expressing fuel consumption, and display the result, showing one place to the right of the decimal. Note that the U. S. scheme measures the distance traveled per amount of fuel (higher is better), whereas the European scheme measures the amount of fuel per distance (lower is better). Use symbolic constants (using `const` or `#define`) for the two conversion factors.