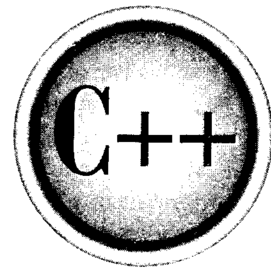
The Complete Reference



Chapter 8

C-Style Console I/O

C++ supports two complete I/O systems. The first it inherits from C. The second is the object-oriented I/O system defined by C++. This and the next chapter discuss the C-like I/O system. (Part Two examines C++ I/O.) While you will probably want to use the C++ I/O system for most new projects, C-style I/O is still quite common, and knowledge of its features is fundamental to a complete understanding of C++.

In C, input and output are accomplished through library functions. There are both console and file I/O functions. Technically, there is little distinction between console I/O and file I/O, but conceptually they are in very different worlds. This chapter examines in detail the console I/O functions. The next chapter presents the file I/O system and describes how the two systems relate.

With one exception, this chapter covers only console I/O functions defined by Standard C++. Standard C++ does not define any functions that perform various screen control operations (such as cursor positioning) or that display graphics, because these operations vary widely between machines. Nor does it define any functions that write to a window or dialog box under Windows. Instead, the console I/O functions perform only TTY-based output. However, most compilers include in their libraries screen control and graphics functions that apply to the specific environment in which the compiler is designed to run. And, of course, you may use C++ to write Windows programs, but keep in mind that the C++ language does not directly define functions that perform these tasks.

The Standard C I/O functions all use the header file **stdio.h**. C++ programs can also use the C++-style header **<cstdio>**.

This chapter refers to the console I/O functions as performing input from the keyboard and output to the screen. However, these functions actually have the standard input and standard output of the system as the target and/or source of their I/O operations. Furthermore, standard input and standard output may be redirected to other devices. These concepts are covered in Chapter 9.

An Important Application Note

Part One of this book uses the C-like I/O system because it is the only style of I/O that is defined for the C subset of C++. As explained, C++ also defines its own object-oriented I/O system. For most C++ applications, you will want to use the C++-specific I/O system, not the C I/O system described in this chapter. However, an understanding of C-based I/O is important for the following reasons:

- At some point in your career you may be called upon to write code that is restricted to the C subset. In this case, you will need to use the C-like I/O functions.
- For the foreseeable future, C and C++ will coexist. Also, many programs will be hybrids of both C and C++ code. Further, it will be common for C programs to be "upgraded" into C++ programs. Thus, knowledge of both the C and the C++

I/O system will be necessary. For example, in order to change the C-style I/O functions into their C++ object-oriented equivalents, you will need to know how both the C and C++ I/O systems operate.

- An understanding of the basic principles behind the C-like I/O system is crucial to an understanding of the C++ object-oriented I/O system. (Both share the same general concepts.)
- In certain situations (for example, in very short programs), it may be easier to use C's non-object-oriented approach to I/O than it is to use the object-oriented I/O defined by C++.

In addition, there is an unwritten rule that any C++ programmer must also be a C programmer. If you don't know how to use the C I/O system, you will be limiting your professional horizons.

Reading and Writing Characters

The simplest of the console I/O functions are **getchar( )**, which reads a character from the keyboard, and **putchar( )**, which prints a character to the screen. The **getchar( )** function waits until a key is pressed and then returns its value. The key pressed is also automatically echoed to the screen. The **putchar( )** function writes a character to the screen at the current cursor position. The prototypes for **getchar( )** and **putchar( )** are shown here:

int getchar(void);
int putchar(int *c*);

As its prototype shows, the **getchar( )** function is declared as returning an integer. However, you can assign this value to a **char** variable, as is usually done, because the character is contained in the low-order byte. (The high-order byte is normally zero.) **getchar( )** returns **EOF** if an error occurs.

In the case of **putchar( )**, even though it is declared as taking an integer parameter, you will generally call it using a character argument. Only the low-order byte of its parameter is actually output to the screen. The **putchar( )** function returns the character written, or **EOF** if an error occurs. (The **EOF** macro is defined in **stdio.h** and is generally equal to –1.)

The following program illustrates **getchar( )** and **putchar( )**. It inputs characters from the keyboard and displays them in reverse case—that is, it prints uppercase as lowercase and lowercase as uppercase. To stop the program, enter a period.

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

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

  printf("Enter some text (type a period to quit).\n");
  do {
    ch = getchar();

    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);

    putchar(ch);
  } while (ch != '.');

  return 0;
}
```

A Problem with getchar( )

There are some potential problems with **getchar( )**. Normally, **getchar( )** is implemented in such a way that it buffers input until ENTER is pressed. This is called *line-buffered* input; you have to press ENTER before anything you typed is actually sent to your program. Also, since **getchar( )** inputs only one character each time it is called, line-buffering may leave one or more characters waiting in the input queue, which is annoying in interactive environments. Even though Standard C/C++ specify that **getchar( )** can be implemented as an interactive function, it seldom is. Therefore, if the preceding program did not behave as you expected, you now know why.

Alternatives to getchar( )

getchar( ) might not be implemented by your compiler in such a way that it is useful in an interactive environment. If this is the case, you might want to use a different function to read characters from the keyboard. Standard C++ does not define any function that is guaranteed to provide interactive input, but virtually all C++ compilers do. Although these functions are not defined by Standard C++, they are commonly used since **getchar( )** does not fill the needs of most programmers.

Two of the most common alternative functions, **getch( )** and **getche( )**, have these prototypes:

```
int getch(void);
int getche(void);
```

For most compilers, the prototypes for these functions are found in the header file **conio.h**. For some compilers, these functions have a leading underscore. For example, in Microsoft's Visual C++, they are called **_getch( )** and **_getche( )**.

The **getch( )** function waits for a keypress, after which it returns immediately. It does not echo the character to the screen. The **getche( )** function is the same as **getch( )**, but the key is echoed. You will frequently see **getche( )** or **getch( )** used instead of **getchar( )** when a character needs to be read from the keyboard in an interactive program. However, if your compiler does not support these alternative functions, or if **getchar( )** is implemented as an interactive function by your compiler, you should substitute **getchar( )** when necessary.

For example, the previous program is shown here using **getch( )** instead of **getchar( )**:

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

int main(void)
{
  char ch;

  printf("Enter some text (type a period to quit).\n");
  do {
    ch = getch();

    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);

    putchar(ch);
  } while (ch != '.');

  return 0;
}
```

When you run this version of the program, each time you press a key, it is immediately transmitted to the program and displayed in reverse case. Input is no longer line-buffered. While the code in this book will not make further use of **getch( )** or **getche( )**, they may be useful in the programs that you write.

Note *At the time of this writing, when using Microsoft's Visual C++ compiler,* ***_getche( )*** *and* ***_getch( )*** *are not compatible with the standard C/C++ input functions, such as* ***scanf( )*** *or* ***gets( )***. *Instead, you must use special versions of the standard functions, such as* ***cscanf( )*** *or* ***cgets( )***. *You will need to examine the Visual C++ documentation for details.*

Reading and Writing Strings

The next step up in console I/O, in terms of complexity and power, are the functions **gets( )** and **puts( )**. They enable you to read and write strings of characters.

The **gets( )** function reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. You may type characters at the keyboard until you press ENTER. The carriage return does not become part of the string; instead, a null terminator is placed at the end and **gets( )** returns. In fact, you cannot use **gets( )** to return a carriage return (although **getchar( )** can do so). You can correct typing mistakes by using the backspace key before pressing ENTER. The prototype for **gets( )** is

char *gets(char *str*);

where *str* is a character array that receives the characters input by the user. **gets( )** also returns *str*. The following program reads a string into the array **str** and prints its length:

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

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

  gets(str);
  printf("Length is %d", strlen(str));

  return 0;
}
```

You need to be careful when using **gets( )** because it performs no boundary checks on the array that is receiving input. Thus, it is possible for the user to enter more characters than the array can hold. While **gets( )** is fine for sample programs and simple utilities that only you will use, you will want to avoid its use in commercial code. One alternative is the **fgets( )** function described in the next chapter, which allows you to prevent an array overrun.

The **puts( )** function writes its string argument to the screen followed by a newline. Its prototype is:

int puts(const char *str*);

puts( ) recognizes the same backslash codes as **printf( )**, such as '\t' for tab. A call to **puts( )** requires far less overhead than the same call to **printf( )** because **puts( )**

can only output a string of characters—it cannot output numbers or do format conversions. Therefore, **puts( )** takes up less space and runs faster than **printf( )**. For this reason, the **puts( )** function is often used when it is important to have highly optimized code. The **puts( )** function returns **EOF** if an error occurs. Otherwise, it returns a nonnegative value. However, when writing to the console, you can usually assume that no error will occur, so the return value of **puts( )** is seldom monitored. The following statement displays **hello**:

```
puts("hello");
```

Table 8-1 summarizes the basic console I/O functions.

The following program, a simple computerized dictionary, demonstrates several of the basic console I/O functions. It prompts the user to enter a word and then checks to see if the word matches one in its built-in database. If a match is found, the program prints the word's meaning. Pay special attention to the indirection used in this program. If you have any trouble understanding it, remember that the **dic** array is an array of pointers to strings. Notice that the list must be terminated by two nulls.

Function	Operation
getchar( )	Reads a character from the keyboard; waits for carriage return.
getche( )	Reads a character with echo; does not wait for carriage return; not defined by Standard C/C++, but a common extension.
getch( )	Reads a character without echo; does not wait for carriage return; not defined by Standard C/C++, but a common extension.
putchar( )	Writes a character to the screen.
gets( )	Reads a string from the keyboard.
puts( )	Writes a string to the screen.

Table 8-1. *The Basic I/O Functions*

```
/* A simple dictionary. */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/* list of words and meanings */
char  *dic[][40] = {
  "atlas", "A volume of maps.",
  "car", "A motorized vehicle.",
  "telephone", "A communication device.",
  "airplane", "A flying machine.",
  "", ""  /* null terminate the list */
};

int main(void)
{
  char word[80], ch;
  char **p;

  do {
    puts("\nEnter word: ");
    scanf("%s", word);

    p = (char **)dic;

    /* find matching word and print its meaning */
    do {
      if(!strcmp(*p, word)) {
        puts("Meaning:");
        puts(*(p+1));
        break;
      }
      if(!strcmp(*p, word)) break;
      p = p + 2;  /* advance through the list */
    } while(*p);
    if(!*p) puts("Word not in dictionary.");
    printf("Another? (y/n): ");
    scanf(" %c%*c", &ch);
  } while(toupper(ch) != 'N');

  return 0;
}
```

Formatted Console I/O

The functions **printf( )** and **scanf( )** perform formatted output and input—that is, they can read and write data in various formats that are under your control. The **printf( )** function writes data to the console. The **scanf( )** function, its complement, reads data from the keyboard. Both functions can operate on any of the built-in data types, including characters, strings, and numbers.

printf( )

The prototype for **printf( )** is

int printf(const char **control_string*, ...);

The **printf( )** function returns the number of characters written or a negative value if an error occurs.

The *control_string* consists of two types of items. The first type is composed of characters that will be printed on the screen. The second type contains format specifiers that define the way the subsequent arguments are displayed. A format specifier begins with a percent sign and is followed by the format code. There must be exactly the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order from left to right. For example, this **printf( )** call

```
printf("I like %c%s", 'C', "++ very much!");
```

displays

```
I like C++ very much!
```

The **printf( )** function accepts a wide variety of format specifiers, as shown in Table 8-2.

Code	Format
%c	Character
%d	Signed decimal integers

Table 8-2. ***printf( )*** *Format Specifiers*

Code	Format
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %F, whichever is shorter
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument must be a pointer to an integer. This specifier causes the number of characters written so far to be put into that integer.
%%	Prints a % sign

Table 8-2. ***printf( )*** *Format Specifiers* **(continued)**

Printing Characters

To print an individual character, use **%c**. This causes its matching argument to be output, unmodified, to the screen.

To print a string, use **%s**.

Printing Numbers

You may use either **%d** or **%i** to indicate a signed decimal number. These format specifiers are equivalent; both are supported for historical reasons.

To output an unsigned value, use **%u**.

The **%f** format specifier displays numbers in floating point.

The %**e** and %**E** specifiers tell **printf( )** to display a **double** argument in scientific notation. Numbers represented in scientific notation take this general form:

x.dddddE+/-yy

If you want to display the letter "E" in uppercase, use the %**E** format; otherwise use %**e**.

You can tell **printf( )** to use either %**f** or %**e** by using the %**g** or %**G** format specifiers. This causes **printf( )** to select the format specifier that produces the shortest output. Where applicable, use %**G** if you want "E" shown in uppercase; otherwise, use %**g**. The following program demonstrates the effect of the %**g** format specifier:

```
#include <stdio.h>

int main(void)
{
  double f;

  for(f=1.0; f<1.0e+10; f=f*10)
    printf("%g ", f);

  return 0;
}
```

It produces the following output.

```
1 10 100 1000 10000 100000 1e+006 1e+007 1e+008 1e+009
```

You can display unsigned integers in octal or hexadecimal format using %**o** and %**x**, respectively. Since the hexadecimal number system uses the letters A through F to represent the numbers 10 through 15, you can display these letters in either upper- or lowercase. For uppercase, use the %**X** format specifier; for lowercase, use %**x**, as shown here:

```
#include <stdio.h>

int main(void)
{
  unsigned num;

  for(num=0; num<255; num++) {
    printf("%o ", num);
    printf("%x ", num);
```

```
    printf("%X\n", num);
  }

  return 0;
}
```

Displaying an Address

If you wish to display an address, use **%p**. This format specifier causes **printf( )** to display a machine address in a format compatible with the type of addressing used by the computer. The next program displays the address of **sample**:

```
#include <stdio.h>

int sample;

int main(void)
{
  printf("%p", &sample);

  return 0;
}
```

The %n Specifier

The **%n** format specifier is different from the others. Instead of telling **printf( )** to display something, it causes **printf( )** to load the variable pointed to by its corresponding argument with a value equal to the number of characters that have been output. In other words, the value that corresponds to the **%n** format specifier must be a pointer to a variable. After the call to **printf( )** has returned, this variable will hold the number of characters output, up to the point at which the **%n** was encountered. Examine this program to understand this somewhat unusual format code.

```
#include <stdio.h>

int main(void)
{
  int count;

  printf("this%n is a test\n", &count);
  printf("%d", count);
```

```
  return 0;
}
```

This program displays **this is a test** followed by the number 4. The **%n** format specifier is used primarily to enable your program to perform dynamic formatting.

Format Modifiers

Many format specifiers may take modifiers that alter their meaning slightly. For example, you can specify a minimum field width, the number of decimal places, and left justification. The format modifier goes between the percent sign and the format code. These modifiers are discussed next.

The Minimum Field Width Specifier

An integer placed between the % sign and the format code acts as a *minimum field width specifier*. This pads the output with spaces to ensure that it reaches a certain minimum length. If the string or number is longer than that minimum, it will still be printed in full. The default padding is done with spaces. If you wish to pad with 0's, place a 0 before the field width specifier. For example, **%05d** will pad a number of less than five digits with 0's so that its total length is five. The following program demonstrates the minimum field width specifier:

```
#include <stdio.h>

int main(void)
{
  double item;

  item = 10.12304;

  printf("%f\n", item);
  printf("%10f\n", item);
  printf("%012f\n", item);

  return 0;
}
```

This program produces the following output:

```
10.123040
 10.123040
00010.123040
```

The minimum field width modifier is most commonly used to produce tables in which the columns line up. For example, the next program produces a table of squares and cubes for the numbers between 1 and 19:

```
#include <stdio.h>

int main(void)
{
  int i;

  /* display a table of squares and cubes */
  for(i=1; i<20; i++)
    printf("%8d %8d %8d\n", i, i*i, i*i*i);

  return 0;
}
```

A sample of its output is shown here:

```
       1        1        1
       2        4        8
       3        9       27
       4       16       64
       5       25      125
       6       36      216
       7       49      343
       8       64      512
       9       81      729
      10      100     1000
      11      121     1331
      12      144     1728
      13      169     2197
      14      196     2744
      15      225     3375
      16      256     4096
      17      289     4913
      18      324     5832
      19      361     6859
```

The Precision Specifier

The *precision specifier* follows the minimum field width specifier (if there is one). It consists of a period followed by an integer. Its exact meaning depends upon the type of data it is applied to.

When you apply the precision specifier to floating-point data using the **%f**, **%e**, or **%E** specifiers, it determines the number of decimal places displayed. For example, **%10.4f** displays a number at least ten characters wide with four decimal places.

When the precision specifier is applied to **%g** or **%G**, it specifies the number of significant digits.

Applied to strings, the precision specifier specifies the maximum field length. For example, **%5.7s** displays a string at least five and not exceeding seven characters long. If the string is longer than the maximum field width, the end characters will be truncated.

When applied to integer types, the precision specifier determines the minimum number of digits that will appear for each number. Leading zeros are added to achieve the required number of digits.

The following program illustrates the precision specifier:

```
#include <stdio.h>

int main(void)
{
  printf("%.4f\n", 123.1234567);
  printf("%3.8d\n", 1000);
  printf("%10.15s\n", "This is a simple test.");

  return 0;
}
```

It produces the following output:

```
123.1235
00001000
This is a simpl
```

Justifying Output

By default, all output is right-justified. That is, if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force output to be left-justified by placing a minus sign directly after the %. For example, **%–10.2f** left-justifies a floating-point number with two decimal places in a 10-character field.

The following program illustrates left justification:

```
#include <stdio.h>

int main(void)
{
  printf("right-justified:%8d\n", 100);
  printf("left-justified:%-8d\n", 100);
```

```
    return 0;
}
```

Handling Other Data Types

There are two format modifiers that allow **printf( )** to display **short** and **long** integers. These modifiers may be applied to the **d**, **i**, **o**, **u**, and **x** type specifiers. The **l** (*ell*) modifier tells **printf( )** that a **long** data type follows. For example, **%ld** means that a **long int** is to be displayed. The **h** modifier instructs **printf( )** to display a **short** integer. For instance, **%hu** indicates that the data is of type **short unsigned int**.

The **l** and **h** modifiers can also be applied to the **n** specifier, to indicate that the corresponding argument is a pointer to a long or short integer, respectively.

If your compiler fully complies with Standard C++, then you can use the **l** modifier with the **c** format to indicate a wide-character. You can also use the **l** modifier with the **s** format to indicate a wide-character string.

The **L** modifier may prefix the floating-point specifiers **e**, **f**, and **g**, and indicates that a **long double** follows.

The * and # Modifiers

The **printf( )** function supports two additional modifiers to some of its format specifiers: * and #.

Preceding **g**, **G**, **f**, **E**, or **e** specifiers with a # ensures that there will be a decimal point even if there are no decimal digits. If you precede the **x** or **X** format specifier with a **#**, the hexadecimal number will be printed with a **0x** prefix. Preceding the **o** specifier with # causes the number to be printed with a leading zero. You cannot apply # to any other format specifiers.

Instead of constants, the minimum field width and precision specifiers may be provided by arguments to **printf( )**. To accomplish this, use an * as a placeholder. When the format string is scanned, **printf( )** will match the * to an argument in the order in which they occur. For example, in Figure 8-1, the minimum field width is 10, the precision is 4, and the value to be displayed is **123.3**.

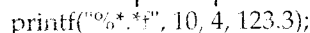


Figure 8-1. *How the * is matched to its value*

The following program illustrates both # and *:

```
#include <stdio.h>

int main(void)
{
  printf("%x %#x\n", 10, 10);
  printf("%*.*f", 10, 4, 1234.34);

  return 0;
}
```

scanf( )

scanf( ) is the general-purpose console input routine. It can read all the built-in data types and automatically convert numbers into the proper internal format. It is much like the reverse of **printf( )**. The prototype for **scanf( )** is

int scanf(const char **control_string*, ...);

The **scanf( )** function returns the number of data items successfully assigned a value. If an error occurs, **scanf( )** returns **EOF**. The *control_string* determines how values are read into the variables pointed to in the argument list.

The control string consists of three classifications of characters:

- Format specifiers
- White-space characters
- Non-white-space characters

Let's take a look at each of these now.

Format Specifiers

The input format specifiers are preceded by a % sign and tell **scanf( )** what type of data is to be read next. These codes are listed in Table 8-3. The format specifiers are matched, in order from left to right, with the arguments in the argument list. Let's look at some examples.

Inputting Numbers

To read an integer, use either the **%d** or **%i** specifier. To read a floating-point number represented in either standard or scientific notation, use **%e**, **%f**, or **%g**.

You can use **scanf( )** to read integers in either octal or hexadecimal form by using the **%o** and **%x** format commands, respectively. The **%x** may be in either upper- or lowercase.

Code	Meaning
%c	Read a single character.
%d	Read a decimal integer.
%i	Read an integer in either decimal, octal, or hexadecimal format.
%e	Read a floating-point number.
%f	Read a floating-point number.
%g	Read a floating-point number.
%o	Read an octal number.
%s	Read a string.
%x	Read a hexadecimal number.
%p	Read a pointer.
%n	Receives an integer value equal to the number of characters read so far.
%u	Read an unsigned decimal integer.
%[ ]	Scan for a set of characters.
%%	Read a percent sign.

Table 8-3. ***scanf( )*** *Format Specifiers*

Either way, you may enter the letters "A" through "F" in either case when entering hexadecimal numbers. The following program reads an octal and hexadecimal number:

```
#include <stdio.h>

int main(void)
{
  int i, j;

  scanf("%o%x", &i, &j);
  printf("%o %x", i, j);

  return 0;
}
```

The **scanf( )** function stops reading a number when the first nonnumeric character is encountered.

Inputting Unsigned Integers

To input an unsigned integer, use the **%u** format specifier. For example,

```
unsigned num;
scanf("%u", &num);
```

reads an unsigned number and puts its value into **num**.

Reading Individual Characters Using scanf( )

As explained earlier in this chapter, you can read individual characters using **getchar( )** or a derivative function. You can also use **scanf( )** for this purpose if you use the **%c** format specifier. However, like most implementations of **getchar( )**, **scanf( )** will generally line-buffer input when the **%c** specifier is used. This makes it somewhat troublesome in an interactive environment.

Although spaces, tabs, and newlines are used as field separators when reading other types of data, when reading a single character, white-space characters are read like any other character. For example, with an input stream of "**x y**," this code fragment

```
scanf("%c%c%c", &a, &b, &c);
```

returns with the character **x** in **a**, a space in **b**, and the character **y** in **c**.

Reading Strings

The **scanf( )** function can be used to read a string from the input stream using the **%s** format specifier. Using **%s** causes **scanf( )** to read characters until it encounters a white-space character. The characters that are read are put into the character array pointed to by the corresponding argument and the result is null terminated. As it applies to **scanf( )**, a white-space character is either a space, a newline, a tab, a vertical tab, or a form feed. Unlike **gets( )**, which reads a string until a carriage return is typed, **scanf( )** reads a string until the first white space is entered. This means that you cannot use **scanf( )** to read a string like "this is a test" because the first space terminates the reading process. To see the effect of the **%s** specifier, try this program using the string "hello there".

```
#include <stdio.h>

int main(void)
```

```
{
  char str[80];

  printf("Enter a string: ");
  scanf("%s", str);
  printf("Here's your string: %s", str);

  return 0;
}
```

The program responds with only the "hello" portion of the string.

Inputting an Address

To input a memory address, use the **%p** format specifier. This specifier causes **scanf( )** to read an address in the format defined by the architecture of the CPU. For example, this program inputs an address and then displays what is at that memory address:

```
#include <stdio.h>

int main(void)
{
  char *p;

  printf("Enter an address: ");
  scanf("%p", &p);
  printf("Value at location %p is %c\n", p, *p);

  return 0;
}
```

The %n Specifier

The **%n** specifier instructs **scanf( )** to assign the number of characters read from the input stream at the point at which the **%n** was encountered to the variable pointed to by the corresponding argument.

Using a Scanset

The **scanf( )** function supports a general-purpose format specifier called a scanset. A *scanset* defines a set of characters. When **scanf( )** processes a scanset, it will input characters as long as those characters are part of the set defined by the scanset. The characters read will be assigned to the character array that is pointed to by the scanset's corresponding

argument. You define a scanset by putting the characters to scan for inside square brackets. The beginning square bracket must be prefixed by a percent sign. For example, the following scanset tells **scanf( )** to read only the characters X, Y, and Z.

```
%[XYZ]
```

When you use a scanset, **scanf( )** continues to read characters, putting them into the corresponding character array until it encounters a character that is not in the scanset. Upon return from **scanf( )**, this array will contain a null-terminated string that consists of the characters that have been read. To see how this works, try this program:

```
#include <stdio.h>

int main(void)
{
  int i;
  char str[80], str2[80];

  scanf("%d%[abcdefg]%s", &i, str, str2);
  printf("%d %s %s", i, str, str2);

  return 0;
}
```

Enter **123abcdtye** followed by ENTER. The program will then display **123 abcd tye**. Because the "t" is not part of the scanset, **scanf( )** stops reading characters into **str** when it encounters the "t." The remaining characters are put into **str2**.

You can specify an inverted set if the first character in the set is a ^. The ^ instructs **scanf( )** to accept any character that is *not* defined by the scanset.

In most implementations you can specify a range using a hyphen. For example, this tells **scanf( )** to accept the characters A through Z:

```
%[A-Z]
```

One important point to remember is that the scanset is case sensitive. If you want to scan for both upper- and lowercase letters, you must specify them individually.

Discarding Unwanted White Space

A white-space character in the control string causes **scanf( )** to skip over one or more leading white-space characters in the input stream. A white-space character is either a

space, a tab, vertical tab, form feed, or a newline. In essence, one white-space character in the control string causes **scanf( )** to read, but not store, any number (including zero) of white-space characters up to the first non-white-space character.

Non-White-Space Characters in the Control String

A non-white-space character in the control string causes **scanf( )** to read and discard matching characters in the input stream. For example, **"%d,%d"** causes **scanf( )** to read an integer, read and discard a comma, and then read another integer. If the specified character is not found, **scanf( )** terminates. If you wish to read and discard a percent sign, use %% in the control string.

You Must Pass scanf( ) Addresses

All the variables used to receive values through **scanf( )** must be passed by their addresses. This means that all arguments must be pointers to the variables used as arguments. Recall that this is one way of creating a call by reference, and it allows a function to alter the contents of an argument. For example, to read an integer into the variable **count**, you would use the following **scanf( )** call:

```
scanf("%d", &count);
```

Strings will be read into character arrays, and the array name, without any index, is the address of the first element of the array. So, to read a string into the character array **str**, you would use

```
scanf("%s", str);
```

In this case, **str** is already a pointer and need not be preceded by the **&** operator.

Format Modifiers

As with **printf( )**, **scanf( )** allows a number of its format specifiers to be modified.

The format specifiers can include a maximum field length modifier. This is an integer, placed between the % and the format specifier, that limits the number of characters read for that field. For example, to read no more than 20 characters into **str**, write

```
scanf("%20s", str);
```

If the input stream is greater than 20 characters, a subsequent call to input begins where this call leaves off. For example, if you enter

ABCDEFGHIJKLMNOPQRSTUVWXYZ

as the response to the **scanf( )** call in this example, only the first 20 characters, or up to the "T," are placed into **str** because of the maximum field width specifier. This means that the remaining characters, UVWXYZ, have not yet been used. If another **scanf( )** call is made, such as

```
scanf("%s", str);
```

the letters UVWXYZ are placed into **str**. Input for a field may terminate before the maximum field length is reached if a white space is encountered. In this case, **scanf( )** moves on to the next field.

To read a long integer, put an **l** (*ell*) in front of the format specifier. To read a short integer, put an **h** in front of the format specifier. These modifiers can be used with the **d**, **i**, **o**, **u**, **x**, and **n** format codes.

By default, the **f**, **e**, and **g** specifiers instruct **scanf( )** to assign data to a **float**. If you put an **l** (*ell*) in front of one of these specifiers, **scanf( )** assigns the data to a **double**. Using an **L** tells **scanf( )** that the variable receiving the data is a **long double**.

Suppressing Input

You can tell **scanf( )** to read a field but not assign it to any variable by preceding that field's format code with an *. For example, given

```
scanf("%d%*c%d", &x, &y);
```

you could enter the coordinate pair **10,10**. The comma would be correctly read, but not assigned to anything. Assignment suppression is especially useful when you need to process only a part of what is being entered.