# C I/O Functions: printf and scanf

The C library functions used to display data to the screen (printf) and read data from the keyboard (scanf) can be used in C++ programs. These functions correspond to cout and cin, respectively. Use of printf and scanf is described in this document. The C getchar() function's use during input error handling is also included. An include directive for the header file **cstdio** should be placed in C++ programs using these functions.

**Output**

The printf function displays data to the screen. Its general form is:

```
printf    (    format string     [ , value1, value2 . . . ]    );
```

Printf has two types of arguments. The first argument is always the format string. The format string provides printf with text to display and/or conversion specification(s). A conversion specification (see table to right) indicates the data type of a value to display and optional formatting information. The letters in specifications are lowercase. The 'l' for long's '%ld' is a lowercase 'L', not a one. Formatting is covered in the next section of this document.

| Data Type | Conversion Specification |
|---|---|
| char | `%c` |
| int / short | `%d` |
| long | `%ld` |
| float / double | `%f` |
| long double | `%Lf` |
| string constant / C string | `%s` |

Following the format string, zero, one, or multiple additional arguments provide printf with data to display. A comma should be placed between arguments. A printf example is given below.

```
printf("The paperback %s has %d pages and costs $%f.",
   "The Big Dog", 230, 6.99);
```
Output: `The paperback The Big Dog has 230 pages and costs $6.990000.`

The format string is **"The paperback %s has %d pages and costs $%f."** The additional, data value arguments are **"The Big Dog"**, **230**, and **6.99**.

Each conversion specification in the format string corresponds to a subsequent argument. The data types of the specifications and arguments should also match. In the example, the specifications indicate C string, integer, and floating-point values are to be displayed. There are therefore three subsequent values having these data types, in this order. If the format string does not contain any conversion specifications, it will be the printf statement's only argument.

Here are a few more printf examples and their output.

```
int number = 3;                  double taxRate = .065;
char middleInitial = 'D';        char lastName[8] = "Queue";

printf("Hello!");
```
Output: `Hello!`

```
printf("The magic number is %d", number);
```
Output: `The magic number is 3`

```
printf("%f is the current sales tax rate", taxRate);
```
Output: `0.065000 is the current sales tax rate`

```
printf("The name is %c. %c. %s", 'P', middleInitial, lastName);
```
Output: `The name is P. D. Queue`

If a double or single quotation mark is a character in the format string to display, precede it with a backslash (i.e., `\"` or `\'`). If a percent sign is to be displayed, precede it by another percent sign (`%%`).

**Output Formatting**

The output displayed often needs to be formatted to appear as desired. Options for formatting output are described below. Formatting impacts only the appearance of data on the screen.

Newlines

The newline escape sequence (\n) serves the same purpose in a printf statement that it serves in a cout statement: it moves the cursor to the start of the next line on the screen. Here is an example.

```
printf("\nOne\ntwo\nThree\n");
Output:  One
         Two
         Three
```

The cursor would have been moved down a line before **One** was displayed and moved to the line below **Three** after **Three** was displayed.

Width (equivalent to C++'s "setw()" manipulator)

An integer, called a width, can be placed in a conversion specification to indicate the number of spaces to allot to a displayed value. By default, a value will use the number of spaces needed to display it completely. The value will be right justified within the space allotted by a width and, if it does not fill the indicated width, preceded by spaces. Here are a few examples. The width value should be placed where indicated.

```
printf("***%3d***", 7);
Output:  ***  7***

printf("%5c", 'z');
Output:       z
```

```
printf("Price = %10f", 2.75);
Output:  Price =   2.750000
```

If the value to display is larger than the conversion specification's width, the entire value will still be displayed. The value will "fill" the space indicated by the width and then continue to the right. It is therefore important to select a width large enough to accommodate the size of the largest expected value to display. Here is an example where the width is too small.

```
printf("..%2s..", "Dots");
Output:  ..Dots..
```

Left Justification (equivalent to C++'s "left" manipulator)

A dash (-) can be added to a conversion specification to indicate the value displayed should be left, instead of right, justified within the width. An example is given below. The dash should be placed where shown.

```
printf("***%-3d***", 7);
Output:  ***7  ***
```

The dash only impacts the printf statement argument corresponding to the conversion specification. Unlike C++, left justification is not "turned on" and then in effect until "turned off."

Precision (equivalent to C++'s "setprecision()" manipulator)

When a floating-point value is displayed, six digits will appear after the decimal point unless formatting indicates a different number of digits should appear. The floating-point number examples above reflect this fact.

A precision value can be added to a floating-point number's conversion specification to define the number of decimal places to display. A precision is a decimal point followed by a number. Two examples are given below. The precision value should be placed where indicated.

```
printf("%.3f is the current sales tax rate", taxRate);
Output:  0.065 is the current sales tax rate

printf("Price = %6.2f", 2.75);
Output:  Price =   2.75
```

The second example also includes a width. The width (6) precedes the precision (.2). The width value is always first.

If the value to display has more digits after the decimal point than the precision allows, the value will be altered, on screen, to limit it to the number of places indicated by the precision. Suppose a precision value is .2, but the number to display has more than two places after the decimal point. If the third digit after the decimal point is less than five, the digits after the second digit will be truncated and not appear on the screen. If the third digit is five or greater, the second digit will be rounded up by one and the subsequent digits not displayed. Here are two examples.

```
printf("%.2f", 1.234);  // 4 < 5          printf("%.2f", 1.237);   // 7 >= 5
Output:  1.23                             Output:  1.24
```

The precision only impacts the argument in the printf statement corresponding to the conversion specification. Unlike C++, a precision does not remain in effect until changed.

**Input**

The scanf function reads data entered by the user. Its general form is:

```
scanf     (    format string,    address     [ , address . . . ]     );
```

Scanf has two types of arguments. The first argument is always the format string. It contains of one or more conversion specifications (see table to right) which provide scanf with the data type of each value to read. The letters in specifications are lowercase. The 'l' for long and double is a lowercase 'L'.

| Data Type | Conversion Specification |
|---|---|
| char | %c |
| int | %d |
| short | %hd |
| long | %ld |
| float | %f |
| double | %lf |
| long double | %Lf |
| string constant / C string | %s |

Following the format string, each additional argument, of which there must be at least one, provides scanf with the address of a variable into which a value read is to be stored. A comma should be placed between arguments. Every conversion specification will have a corresponding address (variable) and their data types should also match. Here are a few examples.

```
int number;              char letter;             char word[6];

scanf("%d", &number);    scanf("%c", &letter);    scanf("%s", word);
```

The address of a variable is indicated by preceding the variable's name with the address operator (&). Input into a C string is the exception as an array's name is already treated as an address.

Width

An integer can be placed in a conversion specification to limit the number of characters read (recall: data comes to a program as a series of characters and is converted to the appropriate data type, as necessary). This option can be used, for example, to limit the size of an integer read. Here is an example.

```
scanf("%3d", &number);
```

The largest number in the example which can be read is 999. Any additional characters (digits) entered by the user will remain in the keyboard buffer. If the number of characters entered is less than width, scanf will simply read the characters present.

White Space

For conversion specifications other than %c, leading white space is read and discarded and reading stops when a trailing white space character is reached. The trailing white space character remains in the keyboard buffer.

Scanf reads a single character when %c is used. If the keyboard buffer is not empty, the first character present will be extracted and stored in the variable. If a newline is in the buffer as a result of prior user input, it will be placed in the variable and the program will not pause for input. If a space is placed prior to %c in the format string (see example below), scanf will read but discard any white space characters in the keyboard buffer and also pause for user input.

```
scanf(" %c", &letter);
```

**Input Error Handling**

If a user enters invalid data, scanf will leave the data in the keyboard buffer and not change the value of the variable. For example, if the user is asked to enter a number and instead enters an ‘x’, the ‘x’ and newline will remain in the keyboard buffer and the variable used in the scanf statement will retain its previous value.

Scanf returns the number of values successfully read. Successful input can be confirmed by checking this value. An example is given below.

```
int number;    // Used to store the number successfully read
int numRead;   // Number of values read

do
{
   printf("\nPlease enter a number: ");
   numRead = scanf("%d", &number);

   if(numRead != 1)
   {
      printf("Invalid data was entered. Please try again.");

      while(getchar() != '\n')
         ;
   }
}
while(numRead != 1);
```

Since scanf, in the example, is asked to read one value, it will return 1 when the read is successful. If the value returned, and stored in numRead, is not 1, the program will inform the user invalid data was entered, clear the keyboard buffer, and loop back up to allow the user to try again.

The while loop is used to clear the keyboard buffer. The getchar() function reads a single character. If the keyboard buffer is not empty (true in the example when invalid data is entered), getchar() will provide the first character in the buffer. The character provided by getchar() is tested to determine whether it is a newline (i.e., the last character entered by the user). If it is, the while loop is exited as the buffer is now empty. If it is not a newline, the while loop repeats. The semicolon on a line by itself is called a null, or empty, statement and is serving in the example as the loop body of the while loop.

Note: The while loop can be used after any scanf statement to remove data remaining in the keyboard buffer. Suppose a user is asked to enter a number but enters 23xyz. Scanf will store 23 in the variable but leave xyz and the newline in the keyboard buffer. The while loop placed after the scanf statement will remove the four characters from the buffer. If the user instead enters a good value, the loop will simply remove the newline from the keyboard buffer.