APPENDIX D

Input and Output Format Specifications

This appendix summarizes all the format specifications you have available for stream input and output. You use these with the standard streams `stdin`, `stdout`, and `stderr`, as well as text file streams.

Output Format Specifications

There are three standard library functions for formatted output: the `printf()` that writes to the standard output stream `stdout` (which by default is the command line), the `sprintf()` function that writes to a string, and the `fprintf()` function that writes to a file. These functions have the following form:

```
int printf(const char* format_string, . . .);
int sprintf(char* source_string, const char* format_string, . . .);
int fprintf(FILE* file_stream, const char* format_string, . . .);
```

The ellipsis at the end of the parameter list indicates that there can be zero or more arguments supplied. These functions return the number of bytes written, or a negative value if an error occurs. The format string can contain ordinary characters (including escape sequences) that are written to the output, together with format specifications for outputting the values of succeeding arguments.

An output format specification always begins with a `%` character and has the following general form:

```
%[flags][width][.precision][size_flag]type
```

The items between square brackets are all optional, so the only mandatory bits are the `%` character at the beginning and the `type` specifier for the type of conversion to be used.

The significance of each of the optional parts is as follows:

`[flags]` are zero or more conversion flags that control how the output is presented. The flags you can use are shown in Table D-1.

Table D-1. *Conversion Flags*

Flag	Description
`+`	Include the sign in the output, + or–. For example, `%+d` will output a decimal integer with the sign always included.
`space`	Use space or – for the sign, that is, a positive value is preceded by a space. This is useful for aligning output when there may be positive and negative values in a column of output. For example, `% d` will output a decimal integer with a space for the sign with positive values.

Table D-1. *Conversion Flags (Continued)*

Flag	Description
-	Left-justify the output in the field width with spaces padding to the right if necessary. For example, `%-10d` will output an integer as a decimal value left-justified in a field width of ten characters. The `%-+10d` specification will output a decimal integer with the sign always appearing, and left-justified in a field width of ten characters.
#	Prefix hexadecimal output values with `0x` or `0X` (corresponding to `x` and `X` conversion type specification respectively), and octal values with 0.
0	Use 0 as the pad character to the left in a right-justified output. For example, `%012d` will output a decimal integer right-justified in a field width of 12 characters, padded to the left with zeros as necessary.

`[width]` specifies the minimum field width for the output value. The width you specify will be exceeded if the value does not fit within the specified minimum width. For example, `%15u` outputs an unsigned integer value right-justified in a field width of 15 characters padded to the left with spaces as necessary.

`[.precision]` specifies the number of places following the decimal point in the output for a floating-point value. For example, `%15.6f` outputs a floating-point value in a minimum field width of 15 characters with four places after the decimal point.

`[size_flag]` is a size specification for the value that modifies the meaning of the type specification. Possible size specifications are `l` (lowercase *L*), `L`, `ll` (two lowercase *L*'s), or `h`. The size specification you can use in any given situation depends on the type specification you are using, as shown in Table D-2.

`type` is a character specifying the type of conversion to be applied to the value to be output, as shown in Table D-2.

Table D-2. *Conversion Type and Size Specifications*

Conversion Type	Description
`d`, `i`	The value is assumed to be of type `int` and the output is as a decimal integer. With the `h` size modifier (`hd` or `hi`) the argument is assumed to be type `short`. With the `l` size modifier (`ld` or `li`) the argument is assumed to be type `long`. With the `ll` modifier (`lld` or `lli`) the argument is assumed to be type `long long`.
`u`	The value is assumed to be of type `unsigned int` and the output is as an unsigned decimal integer. With the `h` size modifier (`hu`) the argument is assumed to be type `unsigned short`. With the `l` size modifier (`lu`) the argument is assumed to be type `unsigned long`. With the `ll` modifier (`llu`) the argument is assumed to be type `unsigned long long`.
`o`	The value is assumed to be of type `unsigned int` and the output is as an unsigned octal value. With the `h` size modifier (`ho`) the argument is assumed to be type `unsigned short`. With the `l` size modifier (`lo`) the argument is assumed to be type `unsigned long`. With the `ll` modifier (`llo`) the argument is assumed to be type `unsigned long long`.

Table D-2. *Conversion Type and Size Specifications*

Conversion Type	Description
x or X	The value is assumed to be of type `unsigned int` and the output is as an unsigned hexadecimal value. The hexadecimal digits `a` to `f` are used if the lowercase type conversion specification is used, and `A` to `F` otherwise. With the `h` size modifier (`ho`) the argument is assumed to be type `unsigned short`. With the `l` size modifier (`lo`) the argument is assumed to be type `unsigned long`. With the `ll` modifier (`llo`) the argument is assumed to be type `unsigned long long`.
c	The value is assumed to be of type `char` and the output is as a character. With the `l` size modifier (`lc`) the argument is assumed to be the wide character type `wchar_t`.
e or E	The value is assumed to be of type `double` and the output is as a floating-point value in scientific notation (with an exponent). The exponent value in the output will be preceded by `e` when you use the lowercase type conversion, `e`, and `E` otherwise. With the `L` modifier (`Le` or `LE`) the argument is assumed to be type `long double`.
f or F	The value is assumed to be of type `double` and the output is as a floating-point value in ordinary notation (without an exponent). With the `L` modifier (`Lf` or `LF`) the argument is assumed to be type `long double`.
g or G	The value is assumed to be of type `double` and the output is as a floating-point value in ordinary notation (without an exponent) unless the exponent value is greater than the precision (default value 6) or is less than –4, in which case the output will be in scientific notation. With the `L` modifier (`Lg` or `LG`) the argument is assumed to be type `long double`.
s	The argument is assumed to be a null-terminated string of characters of type `char` and characters are output until the null character is found or until the precision specification is reached if it is present. The optional precision specification represents the maximum number of characters that may be output.
S	When used with `printf()` the argument is assumed to be a null-terminated string of characters of type `wchar_t` and characters are output until the null character is found or until the precision specification is reached if it is present. The optional precision specification represents the maximum number of characters that may be output.
p	The argument is assumed to be a pointer, and because the output is an address it will be a hexadecimal value.
n	The argument is assumed to be a pointer of type `int*` (pointer to `int`) and the number of characters in the output so far is stored at the address pointed to by the argument. If you use the `h` modifier (`hn`) the argument is assumed to be type `short*` (pointer to `short`). If you use the `l` modifier (`ln`) the argument is assumed to be type `long*` (pointer to `long`). If you use the `ll` modifier (`lln`) the argument is assumed to be type `long long*` (pointer to `long long`).
%	No argument is expected and the output is the `%` character.

Input Format Specifications

For the `scanf()` function that reads data from the standard input stream, `stdin` (which by default is the keyboard), the `sscanf()` function that reads data from a string in memory, and the `fscanf()` function that reads data from a file, data is read from the source controlled by a format string that is passed as an argument to the function. These input functions have the following form:

```
int scanf(const char* format_string, pArg1, ...);
int sscanf(const char* destination_string, const char* format_string, ...);
int fscanf(FILE* file_stream, const char* format_string, ...);
```

Each of these functions returns a count of the number of data items read by the operation. The ellipsis at the end of the parameter list indicates that there can be zero or more arguments here. Don't forget, the arguments that follow the format string must always be pointers. It is a common error to use a variable that is not a pointer as an argument to one of these input functions.

The format string controlling how the input is processed can contain spaces, other characters, and format specifications, beginning with a `%` character, for data items.

A single whitespace character in the format string causes the function to ignore successive whitespace characters in the input. The first nonwhitespace character found will be interpreted as the first character of the next data item. When a newline character in the input follows a value that has been read, for example, when you are reading a single character from the keyboard using the `%c` format specification, any newline, tab, or space character that is entered will be read as the input character. This will be particularly apparent when you are reading a single character repeatedly, where the newline from the Enter key will be left in the buffer. If you want the function to ignore the whitespace in such situations, you can force the function to skip whitespace by including at least one whitespace character preceding the `%c` in the format string.

You can also include nonwhitespace characters in the input format string that are not part of a format specification. Any nonwhitespace character in the format string that is not part of a format specification must be matched by the same character in the input, otherwise the input operation ends.

The format specification for an item of data is of the following form:

```
%[*][width][size_flag]type
```

The items enclosed between square brackets are optional. The mandatory parts of the format specification are the `%` character marking the start of the format specification and the `type` conversion type specification at the end. The meanings of the optional parts are as follows:

`[*]` indicates that that the input data item corresponding to this format specification should be scanned but not stored. For example, `%*d` will scan an integer value and discard it.

`[width]` specifies the maximum number of characters to be scanned for this input value. If a whitespace character is found before width characters have been scanned, it is the end of the input for the current data item. For example, `%2d` reads up to two characters as an integer value. The width specification is useful for reading multiple inputs that are not separated by whitespace characters. You could read 12131415 and interpret it as the values 12, 13, 14, and 15 by using `"%2d%2d%2d%2d"` as the format string.

`[size_flag]` modifies the input type specified by the `type` part of the specification. Possible `size_flag` specifications include `h`, `l`, (lowercase *L*), `ll` (lowercase *L*'s), and `L`. Which of these you can use depends on the type specifier you are using, as described in Table D-3.

Table D-3. *Input Conversion Type Specifications and Modifiers*

Conversion Type	Description
c	Reads a single character as type char. When you use the l modifier (%lc) a single character is read as type wchar_t. You can also precede the c or lc specification with a decimal integer, m, to read m successive characters as a string not terminated by null. For example, %20c will read 20 successive characters. The corresponding argument should be a pointer to a character array with sufficient elements to accommodate the number of characters to be read.
d	Reads successive decimal digits as a value of type int. With the h modifier (%hd), successive digits are read and interpreted as type short. With the l modifier (%ld), successive digits are read and interpreted as type long. With the ll modifier (%lld), successive digits are read and interpreted as type long long.
u	Reads successive decimal digits as a value of type unsigned int. With the h modifier (%hu), successive digits are read and interpreted as type unsigned short. With the l modifier (%lu), successive digits are read and interpreted as type unsigned long. With the ll modifier (%llu), successive digits are read and interpreted as type unsigned long long.
o	Reads successive octal digits as a value of type unsigned int. With the h modifier (%ho), successive octal digits are read and interpreted as type unsigned short. With the l modifier (%lo), successive octal digits are read and interpreted as type unsigned long. With the ll modifier (%llo), successive octal digits are read and interpreted as type unsigned long long.
x or X	Reads successive hexadecimal digits as a value of type unsigned int. With the h modifier (%hx or %hX), successive hexadecimal digits are read and interpreted as type unsigned short. With the l modifier (%lx or %lX), successive hexadecimal digits are read and interpreted as type unsigned long. With the ll modifier (%llx or %llX), successive digits are read and interpreted as type unsigned long long.
s	Reads successive characters until a whitespace is reached and stores the address of the null-terminated string that results in the corresponding argument. If you use the l modifier (%ls) the characters are read and stored as a null-terminated wide character string.
n	Reads no input, but the number of characters that have been read from the input source up to this point is stored as an integer at the address specified by the corresponding argument, which should be of type int*.

Note that if you want to read a string that includes whitespace characters, you have the `%[set_of_characters]` form of specification available. With this specification, successive characters are read from the input source as long as they appear in the set you supply between the square brackets. Thus, the specification `%[ abcdefghijklmnopqrstuvwxyz]` will read any sequence of lowercase letters and spaces as a single string. A more useful variation on this is to precede the set of characters with a caret, `^`, as in `%[^set_of_characters]`, in which case the set of characters represents the characters that will be interpreted as ending the string input. For example, the specification `%[^,!]` will read a sequence of characters until either a comma or an exclamation point is found, which will end the string input.