Chapter 11

Input/Output and the Operating System

In this chapter, we explain how to use some of the input/output functions in the standard library, including the functions printf() and scanf(). Although we have used these functions throughout this text, many details still need to be explained. We present extensive tables showing the effects of various formats. The standard library provides functions related to printf() and scanf() that can be used for dealing with files and strings. The use of these functions is explained.

General file input/output is important in applications where data reside in files on disks and tapes. We will show how to open files for processing and how to use a pointer to a file. Some applications need temporary files. Examples are given to illustrate their use.

The operating system provides utilities for the user, and some of these utilities can be used by the programmer to develop C software. A number of the more important tools for programmers will be discussed, including the compiler, *make*, *touch*, *grep*, beautifiers, and debuggers.

11.1 The Output Function printf()

The printf() function has two nice properties that allow flexible use at a high level. First, a list of arguments of arbitrary length can be printed, and second, the printing is controlled by simple conversion specifications, or formats. The function printf() delivers its character stream to the standard output file stdout, which is normally connected to the screen. The argument list to printf() has two parts:

control_string and *other_arguments*

In the example

```
printf("she sells %d %s for f", 99, "sea shells", 3.77);
```

we have

control_string: "she sells %d %s for f"
other_arguments: 99, "sea shells", 3.77

The expressions in *other_arguments* are evaluated and converted according to the formats in the control string and then placed in the output stream. Characters in the control string that are not part of a format are placed directly in the output stream. The % symbol introduces a conversion specification, or format. A single-conversion specification is a string that begins with % and ends with a conversion character:

printf() conversion characters	
Conversion character	How the corresponding argument is printed
c	as a character
d, i	as a decimal integer
u	as an unsigned decimal integer
o	as an unsigned octal integer
x, X	as an unsigned hexadecimal integer
e	as a floating-point number; example: 7.123000e+00
E	as a floating-point number; example: 7.123000E+00
f	as a floating-point number; example: 7.123000
g	in the e-format or f-format, whichever is shorter
G	in the E-format or f-format, whichever is shorter
s	as a string
p	the corresponding argument is a pointer to void; its value is printed as a hexadecimal number
n	the corresponding argument is a pointer to an integer into which the number of characters written so far is printed; the argument is not converted
%	with the format %% a single % is written to the output stream; there is no corresponding argument to be converted

The function printf() returns as an int the number of characters printed. In the example

```
printf("she sells %d %s for f", 99, "sea shells", 3.77);
```

we can match the formats in the control string with their corresponding arguments in the argument list.

Format	Corresponding argument
%d	99
%s	"sea shells"
%f	3.77

Explicit formatting information may be included in a conversion specification. If it is not included, then defaults are used. For example, the format %f with corresponding argument 3.77 will result in 3.770000 being printed. The number is printed with six digits to the right of the decimal point by default.

Between the % that starts a conversion specification and the conversion character that ends it, there may appear in order

§ zero or more flag characters that modify the meaning of the conversion specification. These flag characters are discussed below.

§ an optional positive integer that specifies the minimum field width of the converted argument. The place where an argument is printed is called its field, and the number of spaces used to print an argument is called its field width. If the converted argument has fewer characters than the specified field width, then it will be padded with spaces on the left or right, depending on whether the converted argument is right- or left-adjusted. If the converted argument has more characters than the specified field width, then the field width will be extended to whatever is required. If the integer defining the field width begins with a zero and the argument being printed is right-adjusted in its field, then zeros rather than spaces will be used for padding.

§ an optional *precision*, which is specified by a period followed by a nonnegative integer. For d, i, o, u, x, or X conversions, it specifies the minimum number of digits to be printed. For e, E, and f conversions, it specifies the number of digits to the right of the decimal point. For g and G conversions, it specifies the maximum number of significant digits. For an s conversion, it specifies the maximum number of characters to be printed from a string.

§ an optional h or l, which is a "short" or "long" modifier, respectively. If an h is followed by a d, i, o, u, x, or X conversion character, the conversion specification applies to a short int or unsigned short int argument. If an h is followed by an n conversion character, the corresponding argument is a pointer to a short int or unsigned short int. If an l is followed by a d, i, o, u, x, or X conversion character, the conversion specification applies to a long int or unsigned long int argument. If an l is followed by an n conversion character, the corresponding argument is a pointer to a long int or unsigned long int.

§ an optional L, which is a "long" modifier. If an L is followed by an e, E, f, g, or G conversion character, the conversion specification applies to a long double argument.

The flag characters are

§ a minus sign, which means that the converted argument is to be left-adjusted in its field. If there is no minus sign, then the converted argument is to be right-adjusted in its field.

§ a plus sign, which means that a nonnegative number that comes from a signed conversion is to have a + prepended. This works with the conversion characters d, i, e, E, f, g, and G. All negative numbers start with a minus sign.

§ a space, which means that a nonnegative number that comes from a signed conversion is to have a space prepended. This works with the conversion characters d, i, e, E, f, g, and G. If both a space and a + flag are present, the space flag is ignored.

§ a #, which means that the result is to be converted to an "alternate form" that depends on the conversion character. With conversion character o, the # causes a zero to be prepended to the octal number being printed. In an x or X conversion, the # causes 0x or 0X to be prepended to the hexadecimal number being printed. In a g or G conversion, it causes trailing zeros to be printed. In an e, E, f, g, and G conversion, it causes a decimal point to be printed, even with precision 0. The behavior is undefined for other conversions.

§ a zero, which means that zeros instead of spaces are used to pad the field. With d, i, o, u, x, X, e, E, f, g, and G conversion characters, this can result in numbers with leading zeros. Any sign and any 0x or 0X that gets printed with a number will precede the leading zeros.

In a format, the field width or precision or both may be specified by a * instead of an integer, which indicates that a value is to be obtained from the argument list. Here is an example of how the facility can be used:

```
int     m, n;
double  x = 333.7777777;
.....                   /* get m and n from somewhere */
printf("x = %*.*f\n", m, n, x);
```

If the argument corresponding to the field width has a negative value, then it is taken as a – flag followed by a positive field width. If the argument corresponding to the precision has a negative value, then it is taken as if it were missing.

The conversion specification %% can be used to print a single percent symbol in the output stream. It is a special case because there is no corresponding argument to be converted. For all the other formats, there should be a corresponding argument. If there are not enough arguments, the behavior is undefined. If there are too many arguments, the extra ones are evaluated but otherwise ignored.

The field width is the number of spaces used to print the argument. The default is whatever is required to properly display the value of the argument. Thus, the integer value 255 (decimal) requires three spaces for decimal conversion d or octal conversion o, but only two spaces for hexadecimal conversion x.

When an argument is printed, characters appropriate to the conversion specification are placed in a field. The characters appear right-adjusted unless a minus sign is present as a flag. If the specified field width is too short to properly display the value of the corresponding argument, the field width will be increased to the default. If the entire field is not needed to display the converted argument, then the remaining part of the field is padded with blanks on the left or right, depending on whether the converted argument is right- or left-adjusted. The padding character on the left can be made a zero by specifying the field width with a leading zero.

The precision is specified by a nonnegative number that occurs to the right of the period. For string conversions, this is the maximum number of characters to be printed from the string. For e, E, and f conversions, it specifies the number of digits to appear to the right of the decimal point.

Examples of character and string formats are given in the next table. We use double-quote characters to visually delimit the field. They do not get printed.

Declarations and initializations			
char c = 'A', s[] = "Blue moon!";			
Format	Corresponding argument	How it is printed in its field	Remarks
%c	c	"A"	field width 1 by default
%2c	c	" A"	field width 2, right adjusted
%–3c	c	"A "	field width 3, left adjusted
%s	s	"Blue moon!"	field width 10 by default
%3s	s	"Blue moon!"	more space needed
%.6s	s	"Blue m"	precision 6
%–11.8s	s	"Blue moo "	precision 8, left adjusted

Examples of formats used to print numbers are given in the next table. Again, we use double-quote characters to visually delimit the field. They do not get printed.

Declarations and initializations			

int i = 123;
double x = 0.123456789;			

Format	Corresponding argument	How it is printed in its field	Remarks
%d	i	"123"	field width 3 by default
%05d	i	"00123"	padded with zeros
%7o	i	" 173"	right adjusted, octal
%–9x	i	"7b "	left adjusted, hexadecimal
%–#9x	i	"0x7b "	left adjusted, hexadecimal
%10.5f	x	" 0.12346"	field width 10, precision 5
%–12.5e	x	"1.23457e–01 "	left adjusted, e-format

11.2 The Input Function scanf()

The function scanf() has two nice properties that allow flexible use at a high level. The first is that a list of arguments of arbitrary length can be scanned, and the second is that the input is controlled by simple conversion specifications, or formats. The function scanf() reads characters from the standard input file stdin. The argument list to scanf() has two parts:

control_string and *other_arguments*

In the example

```
char    a, b, c, s[100];
int     n;
double  x;

scanf("%c%c%c%d%s%lf", &a, &b, &c, &n, s, &x);
```

we have

control_string: "%c%c%c%d%s%l
other_arguments: &a, &b, &c, &n, s, &x