# Sams Teach Yourself C in 24 Hours



# Hour 5 - Reading from and Writing to Standard I/O

I/O, I/O, it's off to work we go…

**–The Seven Dwarfs (sort of)**

In the last lesson you learned how to print out characters, integers, and floating-point numbers to the screen by calling the printf() function. In this lesson you're going to learn more about printf(), as well as about the following functions, which are necessary to receive the input from the user or print the output to the screen:

- The getc() function
- The putc() function
- The getchar() function
- The putchar() function

Before we jump into these new functions, let's first get an idea about the standard input and output in C.

## The Standard Input and Output (I/O)

A file contains related characters and numbers. Because all characters and numbers are represented in bits on computers, the C language treats a file as a series of bytes. (8 bits make up 1 byte.) A series of bytes is also called a stream. In fact, the C language treats all file streams equally, although some of the file streams may come from a disk or tape drive, from a terminal, or even from a printer.

Additionally, in C, there are three file streams that are pre-opened for you:

- stdin–The standard input for reading.
- stdout–The standard output for writing.
- stderr–The standard error for writing error messages.

Usually, the standard input (stdin) file stream links to your keyboard, while the standard output (stdout) and the standard error (stderr) file streams point to your terminal screen. Also, many operating systems allow you to redirect these files' streams.

In fact, you've already used stdout. When you executed the printf() function in the last lesson, you were in fact sending the output to the default file stream, stdout, which points to your screen.

You'll learn more on stdin and stdout in the following sections.

**NOTE**

The C language provides many functions to manipulate file reading and writing (I/O). The header file stdio.h contains the declarations for those functions. Therefore, always include the header file stdio.h in your C program before doing anything with the file I/O.

## Getting the Input from the User

In these days, typing from keyboard is still the de facto standard way to input information into computers. The C language has several functions to direct the computer to read the input from the user (typically through the keyboard.) In this lesson the getc() and getchar() functions are introduced.

### Using the getc() Function

The getc() function reads the next character from a file stream, and returns the character as an integer.

The syntax for the getc() function is

```
#include <stdio.h>

int getc(FILE *stream);
```

Here FILE *stream declares a file stream (that is, a variable). The function returns the numeric value of the character read. If an end-of-file or error occurs, the function returns EOF.

At this moment, don't worry about the FILE structure. More details about it are introduced in Hours 21, "Disk File Input and Output: Part I," and 22, "Disk File Input and Output: Part II." In this section, the standard input stdin is used as the file stream specified by FILE *stream.

**NOTE**

Defined in the header file stdio.h, EOF is a constant. EOF stands for end-of-file. Usually, the value of EOF is -1. But keep using EOF, instead of -1, if you need an end-of-file indicator, in case a compiler uses a different value.

Listing 5.1 shows an example that reads a character typed in by the user from the keyboard and then displays the character on the screen.

**TYPE**
**Listing 5.1. Reading in a character entered by the user.**

```
1:  /* 05L01.c: Reading input by calling getc() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int ch;
7:
8:     printf("Please type in one character:\n");
9:     ch = getc( stdin );
10:    printf("The character you just entered is: %c\n", ch);
11:    return 0;
12: }
```

The following is the output displayed on the screen after I run the executable file, 05L01.exe, enter the character H, and press the Enter key:

```
C:\app> 05L01
Please type in one character:
H

The character you just entered is: H

C:\app>
```

**OUTPUT**
You see in line 2 of Listing 5.1 that the header file stdio.h is included for both the getc() and printf() functions used in the program. Lines 4_12 give the name and body of the main() function.

**ANALYSIS**
In line 6, an integer variable, ch, is declared; it is assigned the return value from the getc() function later in line 9. Line 8 prints out a piece of message that asks the user to enter one character from the keyboard. As I mentioned earlier in this lesson, the printf() function in line 8 uses the default standard output stdout to display messages on the screen.

In line 9, the standard input stdin is passed to the getc() function, which indicates that the file stream is from the keyboard. After the user types in a character, the getc() function returns the numeric value (that is, an integer) of the character. You see that, in line 9, the numeric value is assigned to the integer variable ch.

Then, in line 10, the character entered by the user is displayed on the screen with the help of printf(). Note that the character format specifier (%c) is used within the printf() function in line 10. (Exercise 1 in this lesson asks you to use %d in a program to print out the numeric value of a character entered by the user.)

### Using the getchar() Function

The C language provides another function, getchar(), to perform a similar operation to getc(). More precisely, the getchar() function is equivalent to getc(stdin).

The syntax for the getchar() function is

```
#include <stdio.h>

int getchar(void);
```

Here void indicates that no argument is needed for calling the function. The function
returns the numeric value of the character read. If an end-of-file or error occurs, the function returns EOF.

The program in Listing 5.2 demonstrates how to use the getchar() function to read the input from the user.

**TYPE**
**Listing 5.2. Reading in a character by calling getchar().**

```
1:  /* 05L02.c: Reading input by calling getchar() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int ch1, ch2;
7:
8:     printf("Please type in two characters together:\n");
```

```
9:     ch1 = getc( stdin );

10:    ch2 = getchar( );

11:    printf("The first character you just entered is: %c\n", ch1);

12:    printf("The second character you just entered is: %c\n", ch2);

13:    return 0;

14: }
```

**OUTPUT**

After running the executable file, 05L02.exe, and entering two characters (H and i) together without spaces, I press the Enter key and the following output is displayed on the screen:

```
C:\app> 05L02

Please type in two characters together:

Hi

The first character you just entered is: H

The second character you just entered is: i

C:\app>
```

**ANALYSIS**

The program in Listing 5.2 is quite similar to the one in Listing 5.1, except that the former reads in two characters.

The statement in line 6 declares two integers, ch1 and ch2. Line 8 displays a message asking the user to enter two characters together.

Then, the getc() and getchar() functions are called in lines 9 and 10, respectively, to read in two characters entered by the user. Note that in line 10, nothing is passed to the getchar() function. This is because, as mentioned earlier, getchar() has its default file stream–stdin. You can replace the getchar() function in line 10 with getc(stdin), because getc(stdin) is equivalent to getchar().

Lines 11 and 12 send two characters (kept by ch1 and ch2, respectively) to the screen.

## Printing the Output on the Screen

Besides getc() and getchar() for reading, the C language also provides two functions, putc() and putchar(), for writing. The following two sections introduce these functions.

### Using the putc() Function

The putc() function writes a character to the specified file stream, which, in our case, is the standard output pointing to your screen.

The syntax for the putc() function is

```
#include <stdio.h>

int putc(int c, FILE *stream);
```

Here the first argument, int c, indicates that the output is a character saved in an integer variable c; the second argument, FILE *stream, specifies a file stream. If successful, putc() returns the character written; otherwise, it returns EOF.

In this lesson the standard output stdout is used to be the specified file stream in putc().

The putc() function is used in Listing 5.3 to put the character A on the screen.

**TYPE**

**Listing 5.3. Putting a character on the screen.**

```
1:  /* 05L03.c: Outputting a character with putc() */

2:  #include <stdio.h>

3:

4:  main()

5:  {

6:     int ch;

7:

8:     ch = 65;   /* the numeric value of A */

9:     printf("The character that has numeric value of 65 is:\n");

10:    putc(ch, stdout);

11:    return 0;
```

```
12: }
```

**OUTPUT**
The following is what I get from my machine:

```
C:\app> 05L03

The character that has numeric value of 65 is:

A

C:\app>
```

**ANALYSIS**
As mentioned, the header file stdio.h, containing the declaration of putc(), is included in line 2.

The integer variable, ch, declared in line 6, is assigned the numeric value of 65 in line 8. You may remember that 65 is the numeric value of character A.

Line 9 displays a message to remind the user of the numeric value of the character that is going to be put on the screen. Then, the putc() function in line 10 puts character A on the screen. Note that the first argument to the putc() function is the integer variable (ch) that contains 65, and the second argument is the standard output file stream, stdout.

### Another Function for Writing: putchar()

Like putc(), putchar() can also be used to put a character on the screen. The only difference between the two functions is that putchar() needs only one argument to contain the character. You don't need to specify the file stream, because the standard output (stdout) is the default file stream to putchar().

The syntax for the putchar() function is

```
#include <stdio.h>

int putchar(int c);
```

Here int c is the argument that contains the numeric value of a character. The function returns EOF if an error occurs; otherwise, it returns the character that has been written.

An example of using putchar() is demonstrated in Listing 5.4.

**TYPE**
**Listing 5.4. Outputting characters with putchar().**

```
1:  /* 05L04.c: Outputting characters with putchar() */

2:  #include <stdio.h>

3:

4:  main()

5:  {

6:     putchar(65);

7:        putchar(10);

8:           putchar(66);

9:              putchar(10);

10:          putchar(67);

11:       putchar(10);

12:    return 0;

13: }
```

**OUTPUT**
After running the executable file, 05L04.exe, I get the following output:

```
C:\app> 05L04

A

B

C

C:\app>
```

**ANALYSIS**
The way to write the program in Listing 5.4 is a little bit different. There is no variable declared in the program. Rather, integers

are passed to putchar() directly, as shown in lines 6_11.

As you might have figured out, 65, 66, and 67 are, respectively, the numeric values of characters A, B, and C. From exercise 5 of Hour 4, "Data Types and Names in C," or from Appendix C, "ASCII Character Set," you can find out that 10 is the numeric value of the newline character (\n).

Therefore, respectively, lines 6 and 7 put character A on the screen and cause the computer to start at the beginning of the next line. Likewise, line 8 puts B on the screen, and line 9 starts a new line. Then, line 10 puts C on the screen, and line 11 starts another new line. Accordingly, A, B, and C, are put at the beginnings of three consecutive lines, as shown in the output section.

## Revisiting the printf() Function

The printf() function is the first C library function you used in this book to print out messages on the screen. printf() is a very important function in C, so it's worth it to spend more time on it.

The syntax for the printf() function is

```
#include <stdio.h>

int printf(const char *format-string, ...);
```

Here const char *format-string is the first argument that contains the format specifier(s); ... indicates the expression section that contains the expression(s) to be formatted according to the format specifiers. The number of expressions is determined by the number of the format specifiers inside the first argument. The function returns the numbers of expressions formatted if it succeeds. It returns a negative value if an error occurs.

const char * is explained later in this book. For the time being, consider the first argument to the printf() function as a series of characters surrounded with double quotes with some format specifiers inside. For instance, you can pass "The sum of two integers %d + %d is: %d.\n" to the function as the first argument, if needed.

Figure 5.1 shows the relationship between the format string and expressions. Note that the format specifiers and the expressions are matched in order from left to right.

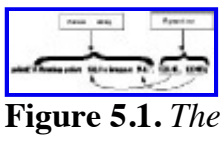

**Figure 5.1.** *The relation between the format string and the expressions in printf().*

Please remember that you should use exactly the same number of expressions as the number of format specifiers within the format string.

The following are all the format specifiers that can be used in printf():

%c The character format specifier.
%d The integer format specifier.
%i The integer format specifier (same as %d).
%f The floating-point format specifier.
%e The scientific notation format specifier (note the lowercase e).
%E The scientific notation format specifier (note the uppercase E).
%g Uses %f or %e, whichever result is shorter.
%G Uses %f or %E, whichever result is shorter.
%o The unsigned octal format specifier.
%s The string format specifier.
%u The unsigned integer format specifier.
%x The unsigned hexadecimal format specifier (note the lowercase x).
%X The unsigned hexadecimal format specifier (note the uppercase X).
%p Displays the corresponding argument that is a pointer.
%n Records the number of characters written so far.
%% Outputs a percent sign (%).

Among the format specifiers in this list, %c, %d, %f, %e, and %E have been introduced so far. Several others are explained later in this book. The next section shows you how to convert decimal numbers to hexadecimal numbers by using %x or %X.

### Converting to Hex Numbers

The difference between a decimal number and a hexadecimal number is that the hexadecimal is a base-16 numbering system. A hexadecimal number can be represented by four bits. ($2^4$ is equal to 16, which means four bits can produce 16 unique numbers.) Hexadecimal is often written as hex for short.

The hexadecimal numbers 0 through 9 use the same numeric symbols founded in the decimal numbers 0 through 9. uppercase A, B, C, D, E, and F are used to represent, respectively, the hexadecimal numbers 10 through 15. (Similarly, in lowercase, a, b, c, d, e, and f are used to represent these hex numbers.)

Listing 5.5 provides an example of converting decimal numbers to hex numbers by using %x or %X in the printf() function.

**TYPE**
**Listing 5.5. Converting to hex numbers.**

```
1:  /* 05L05.c: Converting to hex numbers */
```

```
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     printf("Hex(uppercase)    Hex(lowercase)    Decimal\n");
7:     printf("%X                %x                %d\n", 0, 0, 0);
8:     printf("%X                %x                %d\n", 1, 1, 1);
9:     printf("%X                %x                %d\n", 2, 2, 2);
10:    printf("%X                %x                %d\n", 3, 3, 3);
11:    printf("%X                %x                %d\n", 4, 4, 4);
12:    printf("%X                %x                %d\n", 5, 5, 5);
13:    printf("%X                %x                %d\n", 6, 6, 6);
14:    printf("%X                %x                %d\n", 7, 7, 7);
15:    printf("%X                %x                %d\n", 8, 8, 8);
16:    printf("%X                %x                %d\n", 9, 9, 9);
17:    printf("%X                %x                %d\n", 10, 10, 10);
18:    printf("%X                %x                %d\n", 11, 11, 11);
19:    printf("%X                %x                %d\n", 12, 12, 12);
20:    printf("%X                %x                %d\n", 13, 13, 13);
21:    printf("%X                %x                %d\n", 14, 14, 14);
22:    printf("%X                %x                %d\n", 15, 15, 15);
23:    return 0;
24: }
```

**OUTPUT**

The following output is obtained by running the executable file, 05L05.exe, on my machine:

```
C:\app> 05L05
Hex(uppercase)   Hex(lowercase)   Decimal
0                0                0
1                1                1
2                2                2
3                3                3
4                4                4
5                5                5
6                6                6
7                7                7
8                8                8
9                9                9
A                a                10
B                b                11
C                c                12
D                d                13
E                e                14
F                f                15
C:\app>
```

**ANALYSIS**

Don't panic when you see so many printf() functions being used in Listing 5.5. In fact, the program in Listing 5.5 is very simple. The program has just one function body from lines 5_23.

The printf() function in line 6 prints out a headline that contains three fields: Hex(uppercase), Hex(lowercase), and Decimal.

Then, lines 7_22 print out the hex and decimal numbers 0 through 15. Sixteen printf() functions are called to accomplish the job. Each of the printf() functions has a format string as the first argument followed by three integers as three expressions. Note that the hex format specifiers %X and %x are used within the format string in each of the printf() functions to convert the corresponding expressions to the hex format (both uppercase and lowercase).

In reality, nobody would write a program like the one in Listing 5.5. Instead, a loop can be used to call the printf() function repeatedly. Looping (or iteration) is introduced in Hour 7, "Doing the Same Thing Over and Over."

### Adding the Minimum Field Width

The C language allows you to add an integer between the percent sign (%) and the letter in a format specifier. The integer is called the minimum field width specifier because it specifies the minimum field width and ensures that the output reaches the minimum width. For example, in %10f, 10 is a minimum field width specifier that ensures that the output is at least 10 character spaces wide.

The example in Listing 5.6 shows how to use the minimum field width specifier.

**TYPE**
**Listing 5.6. Specifying the minimum field width.**

```
1:  /* 05L06.c: Specifying minimum field width */

2:  #include <stdio.h>

3:

4:  main()

5:  {

6:     int num1, num2;

7:

8:     num1 = 12;

9:     num2 = 12345;

10:    printf("%d\n", num1);

11:    printf("%d\n", num2);

12:    printf("%5d\n", num1);

13:    printf("%05d\n", num1);

14:    printf("%2d\n", num2);

15:    return 0;

16: }
```

**OUTPUT**
The following is the result I obtain by running the executable file 05L06.exe:

```
C:\app> 05L06

12

12345

   12

00012

12345

C:\app>
```

**ANALYSIS**
In Listing 5.6, two integer variables, num1 and num2, are declared in line 6, and assigned 12 and 12345, respectively, in lines 8 and 9.

Without using any minimum field width specifiers, lines 10 and 11 print out the two integers by calling the printf() function. You can see in the output section that the output made by the statements in line 10 is 12, which takes two character spaces, while the output, 12345, from line 11 takes five character spaces.

In line 12, a minimum field width, 5, is specified by %5d. The output from line 12 therefore takes five character spaces, with three blank spaces plus two character spaces of 12. (See the third output line in the output section.)

The %05d in printf(), shown in line 13, indicates that the minimum field width is 5, and zeros are used to pad the spaces. Therefore, you see the output made by the execution of the statement in line 13 is

```
00012
```

The %2d in line 14 sets the minimum field width to 2, but you still see the full-size output of 12345 from line 14. This means that when the minimum field width is shorter than the width of the output, the latter is taken, and the output is still printed in full.

### Aligning Output

As you might have noticed in the previous section, all output is right-justified. In other words, by default, all output is placed on the right edge of the field, as long as the field width is longer than the width of the output.

You can change this and force output to be left-justified. To do so, you need to prefix the minimum field specifier with the minus sign (-). For example, %-12d specifies the minimum field width as 12, and justifies the output from the left edge of the field.

Listing 5.7 gives an example of aligning output by left- or right-justification.

**TYPE**
**Listing 5.7. Left- or right-justified output.**

```
1:  /* 05L07.c: Aligning output */

2:  #include <stdio.h>

3:

4:  main()

5:  {

6:     int num1, num2, num3, num4, num5;

7:

8:     num1 = 1;

9:     num2 = 12;

10:    num3 = 123;

11:    num4 = 1234;

12:    num5 = 12345;

13:    printf("%8d  %-8d\n", num1, num1);

14:    printf("%8d  %-8d\n", num2, num2);

15:    printf("%8d  %-8d\n", num3, num3);

16:    printf("%8d  %-8d\n", num4, num4);

17:    printf("%8d  %-8d\n", num5, num5);

18:    return 0;

19: }
```

**OUTPUT**
I get the following output displayed on the screen after I run the executable 05L07.exe from a DOS prompt on my machine:

```
C:\app> 05L07

       1  1

      12  12

     123  123

    1234  1234

   12345  12345

C:\app>
```

**ANALYSIS**
In Listing 5.7, there are five integer variables, num1, num2, num3, num4, and num5, that are declared in line 6 and are assigned values in lines 8_12.

These values represented by the five integer variables are then printed out by the printf() functions in lines 13_17. Note that all the printf() functions have the same first argument: "%8d %-8d\n". Here the first format specifier, %8d, aligns the output at the right edge of the field, and the second specifier, %-8d, does the alignment by justifying the output from the left edge of the field.

After the execution of the statements in lines 13_17, the alignment is accomplished and the output is put on the screen like this:

```
    1  1

   12  12

  123  123

 1234  1234

12345  12345
```

### Using the Precision Specifier

You can put a period (.) and an integer right after the minimum field width specifier. The combination of the period (.) and the integer makes up a precision specifier. The precision specifier is another important specifier you can use to determine the number of decimal places for floating-point numbers, or to specify the maximum field width (or length) for integers or strings. (Strings in C are introduced in Hour 13, "Manipulating Strings.")

For instance, with %10.3f, the minimum field width length is specified as 10 characters long, and the number of decimal places is set to 3. (Remember, the default number of decimal places is 6.) For integers, %3.8d indicates that the minimum field width is 3, and the maximum field width is 8.

Listing 5.8 gives an example of left- or right-justifying output by using precision specifiers.

**TYPE**
**Listing 5.8. Using precision specifiers.**

```
1:  /* 05L08.c: Using precision specifiers */

2:  #include <stdio.h>

3:

4:  main()

5:  {

6:     int int_num;

7:     double flt_num;

8:

9:     int_num = 123;

10:    flt_num = 123.456789;

11:    printf("Default integer format:    %d\n", int_num);

12:    printf("With precision specifier:  %2.8d\n", int_num);

13:    printf("Default float format:      %f\n", flt_num);

14:    printf("With precision specifier:  %-10.2f\n", flt_num);

15:    return 0;

16: }
```

**OUTPUT**
After running the executable file 05L08.exe on my machine, I get the following output on the screen:

```
C:\app> 05L08

Default integer format:    123

With precision specifier:  00000123

Default float format:      123.456789

With precision specifier:  123.46

C:\app>
```

**ANALYSIS**
The program in Listing 5.8 declares one integer variable, int_num, in line 6, and one floating-point number, flt_num, in line 7. Lines 9 and 10 assign 123 and 123.456789 to int_num and flt_num, respectively.

In line 11, the default integer format is specified for the integer variable, int_num, while the statement in line 12 specifies the integer format with a precision specifier that indicates that the maximum field width is 8 characters long. Therefore, you see that five zeros are padded prior to the integer 123 in the second line of the output.

For the floating-point variable, flt_num, line 13 prints out the floating-point value in the default format, and line 14 reduces the decimal places to two by putting the precision specifier .2 within the format specifier %-10.2f. Note here that the left-justification is also specified by the minus sign (-) in the floating-point format specifier.

The floating-point number 123.46 in the fourth line of the output is produced by the statement in line 14 with the precision specifier for two decimal places. Therefore, 123.456789 rounded to two decimal places becomes 123.46.

## Summary

In this lesson you've learned the following:

- The C language treats a file as a series of bytes.
- stdin, stdout, and stderr are three file streams that are pre-opened for you to use.
- The C library functions getc() and getchar() can be used to read in one character from the standard input.
- The C library functions putc() and putchar() can be used to write one character to the standard output.

- %x or %X can be used to convert decimal numbers to hex numbers.
- A minimum field width can be specified and ensured by adding an integer into a format specifier.
- An output can be aligned at either the left or right edge of the output field.
- A precision specifier can be used to specify the decimal place number for floating-point numbers, or the maximum field width for integers or strings.

In the next lesson you'll learn about some important operators in C.

## Q&A

**Q** What are stdin, stdout, and stderr?

**A** In C, a file is treated as a series of bytes that is called file stream. stdin, stdout, and stderr are all pre-opened file streams. stdin is the standard input for reading; stdout is the standard output for writing; stderr is the standard error for outputting error messages.

**Q** How much is the hex number 32?

**A** Hexadecimal, or hex for short, is a base-16 numerical system. Therefore, 32 (hex) is equal to $3*16^1+2*16^0$, or 50 in decimal.

**Q** Are getc(stdin) and getchar() equivalent?

**A** Because the getchar() function reads from the file stream stdin by default, getc(stdin) and getchar() are equivalent.

**Q** In the function printf("The integer %d is the same as the hex %x", 12, 12), what is the relationship between the format specifiers and the expressions?

**A** The two format specifiers, %d and %x, specify the formats of numeric values contained in the expression section. Here the first numeric value of 12 is going to be printed out in integer format, while the second 12 (in the expression section) will be displayed in the hex format. Generally speaking, the number of format specifiers in the format section should match the number of expressions in the expression section.

## Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

### Quiz

1. Can you align your output at the left edge, rather than the right edge, of the output field?
2. What is the difference between putc() and putchar()?
3. What does getchar() return?
4. Within %10.3f, which part is the minimum field width specifier, and which one is the precision specifier?

### Exercises

1. Write a program to put the characters B, y, and e together on the screen.
2. Display the two numbers 123 and 123.456 and align them at the left edge of the field.
3. Given three integers–15, 150, and 1500–write a program that prints the integers on the screen in the hex format.
4. Write a program that uses getchar() and putchar() to read in a character entered by the user and write the character to the screen.
5. If you compile the following C program, what warning or error messages will you get?

```
main()
{
   int ch;
   ch = getchar();
   putchar(ch);
   return 0;
}
```