# Sams Teach Yourself C in 24 Hours

# Hour 15 - Functions in C

Form follows function.

**—L. H. Sullivan**

In Hour 14, "Scope and Storage Classes in C," you might have noticed that a function definition is always given first, before the function is called from a main() function. In fact, you can put a function definition anywhere you want, as long as you keep the function declaration at the first place before the function is called. You'll learn about many function features from the following topics covered in this lesson:

- Function declarations
- Prototyping
- Values returned from functions
- Arguments to functions
- Structured programming

In addition, several C library functions and macros, such as time(), localtime(), asctime(), va_start(), va_arg(), and va_end() are introduced in this hour.

## Declaring Functions

As you know, you have to declare or define a variable before you can use it. This is also true for functions. In C, you have to declare or define a function before you can call it.

### Declaration Versus Definition

According to the ANSI standard, the declaration of a variable or function specifies the interpretation and attributes of a set of identifiers. The definition, on the other hand, requires the C compiler to reserve storage for a variable or function named by an identifier.

A variable declaration is a definition, but a function declaration is not. A function declaration alludes to a function that is defined elsewhere and specifies what kind of value is returned by the function. A function definition defines what the function does, as well as gives the number and type of arguments passed to the function.

A function declaration is not a function definition. If a function definition is placed in your source file before the function is first called, you don't need to make the function declaration. Otherwise, the declaration of a function must be made before the function is invoked.

For example, I've used the printf() function in almost every sample program in this book. Each time, I had to include a header file, stdio.h, because the header file contains the declaration of printf(), which indicates to the compiler the return type and prototype of the function. The definition of the printf() function is placed somewhere else. In C, the definition of this function is saved in a library file that is invoked during the linking states.

### Specifying Return Types

A function can be declared to return any data type, except an array or function. The return statement used in a function definition returns a single value whose type should match the one declared in the function declaration.

By default, the return type of a function is int, if no explicit data type is specified for the function. A data type specifier is placed prior to the name of a function like this:

```
data_type_specifier  function_name();
```

Here data_type_specifier specifies the data type that the function should return. function_name is the function name that should follow the rule of naming in C.

In fact, this declaration form represents the traditional function declaration form before the ANSI standard was created. After setting up the ANSI standard, the function prototype is added to the function declaration.

### Using Prototypes

Before the ANSI standard was created, a function declaration only included the return type of the function. With the ANSI standard, the number and types of arguments passed to a function are allowed to be added into the function declaration. The number and types of an argument are called the function prototype.

The general form of a function declaration, including its prototype, is as follows:

```
data_type_specifier  function_name(
      data_type_specifier argument_name1,
      data_type_specifier argument_name2,
      data_type_specifier argument_name3,
      .
      .
      .
      data_type_specifier argument_nameN,
);
```

The purpose of using a function prototype is to help the compiler check whether the data types of arguments passed to a function match what the function expects. The compiler issues an error message if the data types do not match.

Although argument names, such as argument_name1, argument_name2, and so on, are optional, it is recommended that you include them so that the compiler can identify any mismatches of argument names.

### Making Function Calls

As shown in Figure 15.1, when a function call is made, the program execution jumps to the function and finishes the task assigned to the function. Then the program execution resumes after the called function returns.

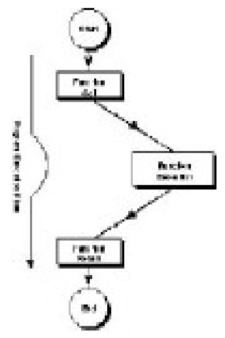
**Figure 15.1.** *Program execution jumps to an invoked function when a function call is made.*

A function call is an expression that can be used as a single statement or within other statements.

Listing 15.1 gives an example of declaring and defining functions, as well as making function calls.

**TYPE**

**Listing 15.1. Calling functions after they are declared and defined.**

```
1:  /* 15L01.c: Making function calls */
2:  #include <stdio.h>
3:
4:  int function_1(int x, int y);
5:  double function_2(double x, double y)
6:  {
7:     printf("Within function_2.\n");
8:     return (x - y);
9:  }
10:
11: main()
12: {
13:    int x1 = 80;
14:    int y1 = 10;
15:    double x2 = 100.123456;
16:    double y2 = 10.123456;
17:
18:    printf("Pass function_1  %d and %d.\n", x1, y1);
19:    printf("function_1 returns %d.\n", function_1(x1, y1));
20:    printf("Pass function_2  %f and %f.\n", x2, y2);
21:    printf("function_2 returns %f.\n", function_2(x2, y2));

22:    return 0;
23: }
24: /* function_1() definition */
25: int function_1(int x, int y)
26: {
27:    printf("Within function_1.\n");
28:    return (x + y);
29: }
```

**OUTPUT**

The following output is displayed on the screen, after the executable (15L01.exe) of the program in Listing 15.1 is created and run from a DOS prompt:

**ANALYSIS**

```
C:\app>15L01
Pass function_1  80 and 10.
Within function_1.
function_1 returns 90.
Pass function_2  100.123456. and 10.123456.
Within function_2.
function_2 returns 90.000000.
C:\app>
```

The purpose of the program in Listing 15.1 is to show you how to declare and define functions. The statement in line 4 is a function declaration with a prototype.
The declaration alludes to the function_1 defined later in Listing 15.1. The return type of function_1 is int, and the function prototype includes two int variables, called x and y.

In lines 5_9, the second function, function_2, is defined before it is called. As you can see, the return type of function_2 is double, and two double variables are passed to the function. Note that the names of the two variables are also x and y. Don't worry because function_1 and function_2 share the same argument names. There is no conflict because these arguments are in different function blocks.

Then, in the main() function defined in lines 11_23, two int variables, x1 and y1, and two double variables, x2 and y2, are declared and initialized in lines 13_16, respectively. The statement in line 18 shows the values of x1 and y1 that are passed to the function_1 function. Line 19 calls function_1 and displays the return value from function_1.

Likewise, lines 20 and 21 print out the values of x2 and y2 that are passed to function_2, as well as the value returned by function_2 after the function is called and executed.

Lines 25_29 contain the definition of the function_1 function, specifying that the function can perform an addition of two integer variables (see line 28) and print out the string of Within function_1. in line 27.

## Prototyping Functions

In the following subsections, we're going to study three cases regarding arguments passed to functions. The first case is a function that takes no argument; the second one is a function that takes a fixed number of arguments; the third case is a function that takes a variable number of arguments.

### Functions with No Arguments

The first case is a function that takes no argument. For instance, the C library function getchar() does not need any arguments. It can be used in a program like this:

```
int c;
c = getchar();
```

As you can see, the second statement is left blank between the parentheses (( and )) when the function is called.

In C, the declaration of the getchar() function can be something like this:

```
int getchar(void);
```

Note that the keyword void is used in the declaration to indicate to the compiler that no argument is needed by this function. The compiler will issue an error message if somehow there is an argument passed to getchar() later in a program when this function is called.

Therefore, for a function with no argument, the void data type is used as the prototype in the function declaration.

The program in Listing 5.2 gives another example of using void in function declarations.

**TYPE**

**Listing 15.2. Using void in function declarations.**

```
1:  /* 15L02.c: Functions with no arguments */
2:  #include <stdio.h>
3:  #include <time.h>
4:
5:  void GetDateTime(void);
6:
7:  main()
8:  {
9:     printf("Before the GetDateTime() function is called.\n");
10:    GetDateTime();
11:    printf("After the GetDateTime() function is called.\n");
12:    return 0;
13: }
14: /* GetDateTime() definition */
15: void GetDateTime(void)
16: {
17:    time_t now;
18:
19:    printf("Within GetDateTime().\n");
20:    time(&now);
21:    printf("Current date and time is: %s\n",
22:       asctime(localtime(&now)));
23: }
```

**OUTPUT**

I obtain the following output after I run the executable, 15L02.exe, of the program in Listing 15.2 from a DOS prompt:

**ANALISYS**

```
C:\app>15L02
Before the GetDateTime() function is called.
Within GetDateTime().
Current date and time is: Sat Apr 05 11:50:10 1997
After the GetDateTime() function is called.
C:\app>
```

The purpose of the program in Listing 15.2 is to give you the current date and time on your computer by calling the function GetDateTime(), declared in line 5. Because no argument needs to be passed to the function, the void data type is used as the prototype in the declaration of GetDateTime().

Additionally, another void keyword is used in front of the name of the GetDateTime() function to indicate that this function doesn't return any value either. (See line 5.)

The statements in lines 9 and 11 print out messages respectively before and after the GetDateTime() function is called from within the main() function.

In line 10, the function is called by the statement GetDateTime();. Note that no argument should be passed to this function, because the function prototype is void.

The definition of GetDateTime() is in lines 15_23; it obtains the calendar time and converts it into a character string by calling several C library functions, such as time(), localtime(), and asctime(). Then, the character string containing the information of current date and time is printed out on the screen by the printf() function with the format specifier %s. As you can see, the output on my screen shows that at the moment the executable 15L02.exe is being executed, the date and time are

```
Sat Apr 05 11:50:10 1997
```

time(), localtime(), and asctime() are date and time functions provided by the C language. These functions are discussed in the following subsection. You might notice that the header file time.h is included at the beginning of the program in Listing 15.2 before these time functions can be used.

**Using time(), localtime(), and asctime()**

Several C functions are called date and time functions. The declarations of all date and time functions are included in the header file time.h. These functions can give three types of date and time:

- Calendar time
- Local time
- Daylight savings time

Here calendar time gives the current date and time based on the Gregorian calendar. Local time represents the calendar time in a specific time zone. Daylight savings time is the local time under the daylight savings rule.

In this section, three date and time functions—time(), localtime(), and asctime()—are briefly introduced.

In C, the time() function returns the calendar time.

The syntax for the time() function is

```
#include <time.h>
time_t time(time_t *timer);
```

Here time_t is the arithmetic type that is used to represent time. timer is a pointer variable pointing to a memory storage that can hold the calendar time returned by this function. The time() function returns -1 if the calendar time is not available on the computer.

The localtime()function returns the local time converted from the calendar time.

The syntax for the localtime() function is

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Here tm is a structure that contains the components of the calendar time. struct is the keyword for structure, which is another data type in C. (The concept of structures is introduced in Hour 19, "Collecting Data Items of Different Types.") timer is a pointer variable pointing to a memory storage that holds the calendar time returned by the time() function.

To convert the date and time represented by the structure tm, you can call the asctime() function.

The syntax for the asctime() function is

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

Here timeptr is a pointer referencing the structure tm returned by date and time functions like localtime(). The asctime() function converts the date and time represented by tm into a character string.

In Listing 15.2, the statement in line 17 declares a time_t variable called now. Line 20 stores the calendar time into the memory location referenced by the now variable. Note that the argument passed to the time() function should be the left value of a variable; therefore, the address-of operator (&) is used prior to now. Then, the expression in line 22, asctime(localtime(&now)), obtains the local time expression of the calendar time by calling localtime(), and converts the local time into a character string with help from asctime(). The character string of the date and time is then printed out by the printf() function in lines 21 and 22, which has the following format:

```
Sat Apr 05 11:50:10 1997\n\0
```

Note that there is a newline character appended right before the null character in the character string that is converted and returned by the asctime() function.

**Functions with a Fixed Number of Arguments**

You have actually seen several examples that declare and call functions with a fixed number of arguments. For instance, in Listing 15.1, the declaration of the function_1() function in line 4

```
int function_1(int x, int y);
```

contains the prototype of two arguments, x and y.

To declare a function with a fixed number of arguments, you need to specify the data type of each argument. Also, it's recommended to indicate the argument names so that the compiler can have a check to make sure that the argument types and names declared in a function declaration match the implementation in the function definition.

**Prototyping a Variable Number of Arguments**

As you may still remember, the syntax of the printf() function is

```
int printf(const char *format[, argument, ...]);
```

Here the ellipsis token ... (that is, three dots) represents a variable number of arguments. In other words, besides the first argument that is a character string, the printf() function can take an unspecified number of additional arguments, as many as the compiler allows. The brackets ([ and ]) indicate that the unspecified arguments are optional.

The following is a general form to declare a function with a variable number of arguments:

```
data_type_specifier  function_name(
      data_type_specifier argument_name1, ...
);
```

Note that the first argument name is followed by the ellipsis (...) that represents the rest of unspecified arguments.

For instance, to declare the printf() function, you can have something like this:

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

**Processing Variable Arguments**

There are three routines, declared in the header file stdarg.h, that enable you to write functions that take a variable number of arguments. They are va_start(), va_arg(), and va_end().

Also included in stdarg.h is a data type, va_list, that defines an array type suitable for containing data items needed by va_start(), va_arg(), and va_end().

To initialize a given array that is needed by va_arg() and va_end(), you have to use the va_start() macro routine before any arguments are processed.

The syntax for the va_start() macro is

```
#include <stdarg.h>
void va_start(va_list ap, lastfix);
```

Here ap is the name of the array that is about to be initialized by the va_start() macro routine. lastfix should be the argument before the ellipsis (...) in the function declaration.

By using the va_arg()macro, you're able to deal with an expression that has the type and value of the next argument. In other words, the va_arg() macro can be used to get the next argument passed to the function.

The syntax for the va_arg() macro is

```
#include <stdarg.h>
type va_arg(va_list ap, data_type);
```

Here ap is the name of the array that is initialized by the va_arg() macro routine. data_type is the data type of the argument passed to function.

To facilitate a normal return from your function, you have to use the va_end() function in
your program after all arguments have been processed.

The syntax for the va_end() function is

```
#include <stdarg.h>
void va_end(va_list ap);
```

Here ap is the name of the array that is initialized by the va_end() macro routine.

Remember to include the header file, stdarg.h, in your program before you call va_start(), va_arg(), or va_end().

Listing 5.3 demonstrates how to use va_start(), va_arg(), and va_end() in a function that takes a variable number of arguments.

**TYPE**

**Listing 15.3. Processing variable arguments.**

```
/* 15L03.c: Processing variable arguments */
#include <stdio.h>
#include <stdarg.h>

double AddDouble(int x, ...);

main ()
{
   double d1 = 1.5;
   double d2 = 2.5;
   double d3 = 3.5;
   double d4 = 4.5;

   printf("Given an argument: %2.1f\n", d1);
   printf("The result returned by AddDouble() is: %2.1f\n\n",
      AddDouble(1, d1));
   printf("Given arguments: %2.1f and %2.1f\n", d1, d2);
   printf("The result returned by AddDouble() is: %2.1f\n\n",
      AddDouble(2, d1, d2));
   printf("Given arguments: %2.1f, %2.1f and %2.1f\n", d1, d2, d3);
   printf("The result returned by AddDouble() is: %2.1f\n\n",
      AddDouble(3, d1, d2, d3));
23:    printf("Given arguments: %2.1f, %2.1f, %2.1f, and %2.1f\n", d1, d2, d3, Âd4);
24:    printf("The result returned by AddDouble() is: %2.1f\n",
25:       AddDouble(4, d1, d2, d3, d4));
26:    return 0;
27: }
28: /* definition of AddDouble() */
29: double AddDouble(int x, ...)
30: {
31:    va_list   arglist;
32:    int i;
33:    double result = 0.0;
34:
35:    printf("The number of arguments is: %d\n", x);
36:    va_start (arglist, x);
37:    for (i=0; i<x; i++)
38:       result += va_arg(arglist, double);
39:    va_end (arglist);
40:    return result;
41: }
```

**OUTPUT**

The following output is displayed on the screen after the executable, 15L03.exe, is run from a DOS prompt:

```
C:\app>15L03
Given an argument: 1.5
The number of arguments is: 1
The result returned by AddDouble() is: 1.5

Given arguments: 1.5 and 2.5
The number of arguments is: 2
The result returned by AddDouble() is: 4.0

Given arguments: 1.5, 2.5, and 3.5
The number of arguments is: 3
The result returned by AddDouble() is: 7.5

Given arguments: 1.5, 2.5, 3.5, and 4.5
The number of arguments is: 4
The result returned by AddDouble() is: 12.0
C:\app>
```

**ANALYSIS**

The program in Listing 15.3 contains a function that can take a variable number of double arguments, perform the operation of addition on these arguments, and then return the result to the main() function.

The declaration in line 5 indicates to the compiler that the AddDouble() function takes a variable number of arguments. The first argument to AddDouble() is an integer variable that holds the number of the rest of the arguments passed to the function each time AddDouble() is called. In other words, the first argument indicates the number of remaining arguments to be processed.

The definition of AddDouble() is given in lines 29_41, in which a va_list array, arglist, is declared in line 31. As mentioned, the va_start() macro has to be called before the arguments are processed. Thus, line 36 invokes va_start() to initialize the array arglist. The for loop in lines

37 and 38 fetches the next double argument saved in the array arglist by calling va_arg(). Then, each argument is added into a local double variable called result.

The va_end() function is called in line 39 after all arguments saved in arglist have been fetched

and processed. Then, the value of result is returned back to the caller of the AddDouble() function, which is the main() function in this case.

The va_end() function has to be called in a C program to end variable argument processing. Otherwise, the behavior of the program is undefined.

As you can see, within the main() function, AddDouble() is called four times, with a different number of arguments each time. These arguments passed to AddDouble() are displayed by the printf() functions in lines 14, 17, 20, and 23. Also, the four different results returned by AddDouble() are printed out on the screen.

## Learning Structured Programming

Now you've learned the basics of function declaration and definition. Before we go to the next hour, let's talk a little bit about structured programming in program design.

Structured programming is one of the best programming methodologies. Basically, there are two types of structured programming: top-down programming and bottom-up programming.

When you start to write a program to solve a problem, one way to do it is to work on the smallest pieces of the problem. First, you define and write functions for each of the pieces. After each function is written and tested, you begin to put them together to build a program that can solve the problem. This approach is normally called bottom-up programming.

On the other hand, to solve a problem, you can first work out an outline and start your programming at a higher level. For instance, you can work on the main() function at the beginning, and then move to the next lower level until the lowest-level functions are written. This type of approach is called top-down programming.

You'll find that it's useful to combine these two types of structured programming and use them alternately in order to solve a real problem.

## Summary

In this lesson you've learned the following:

- A function declaration alludes to a function that is defined elsewhere, and specifies what type of arguments and values are passed to and returned from the function as well.

- A function definition reserves the memory space and defines what the function does, as well as the number and type of arguments passed to the function.
- A function can be declared to return any data type, except an array or a function.
- The return statement used in a function definition returns a single value whose type must be matched with the one declared in the function declaration.
- A function call is an expression that can be used as a single statement or within other expressions or statements.
- The void data type is needed in the declaration of a function that takes no argument.
- To declare a function that takes a variable number of arguments, you have to specify at least the first argument, and use an ellipsis (...) to represent the rest of the arguments passed to the function.
- va_start(), va_arg(), and va_end(), all included in stdarg.h, are needed in processing a variable number of arguments passed to a function.
- time(), localtime(), and asctime() are three time functions provided by C. They can be used together to obtain a character string that contains information of local date and time based on the calendar time.

In the next lesson you'll learn more about pointers and their applications in C.

## Q&A

**Q** What is the main difference between a function declaration and a function definition?

**A** The main difference between a function declaration and a function definition is that the former does not reserve any memory space, nor does it specify what a function does. A function declaration only alludes to a function definition that is placed elsewhere. It also specifies what type of arguments and values are passed to and returned from the function. A function definition, on the other hand, reserves the memory space and specifies tasks the function can complete.

**Q** Why do we need function prototypes?

**A** By declaring a function with prototypes, you specify not only the data type returned by the function, but also the types and names of arguments passed to the function. With the help of a function prototype, the compiler can automatically perform type checking on the definition of the function, which saves you time to debug the program.

**Q** Can a function return a pointer?

**A** Yes. In fact, a function can return a single value that can be any data type except an array or a function. A pointer value—that is, the address—returned by a function can refer to a character array, or a memory location that stores another type of data. For instance, the C library function asctime() returns a character pointer that points to a character string converted from a date-time structure.

**Q** Can you use top-down programming and bottom-up programming together to solve a problem?

**A** Yes. In practice, you can find that it's actually a good idea to combine the top-down and bottom-up programming approaches to solve problems. Using the two types of structured programming can make your program easy to write and understand.

## 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. Given the following function declarations, which ones are functions with a fixed number of arguments, which ones are functions with no arguments, and which ones are functions with a variable number of arguments?
   - int function_1(int x, float y);
   - void function_2(char *str);
   - char *asctime(const struct tm *timeptr);
   - int function_3(void);
   - char function_4(char c, …);
   - void function_5(void);
2. Which one of the following two expressions is a function definition?

   ```
   int function_1(int x, int y);
   int function_2(int x, int y){return x+y;}
   ```

3. What is the data type returned by a function when a type specifier is omitted?

4. Of the following function declarations, which ones are illegal?
   - double function_1(int x, ...);
   - void function_2(int x, int y, ...);
   - char function_3(...);
   - int function_4(int, int, int, int);

### Exercises

1. Rewrite the program in Listing 15.2. This time use the format specifier %c, instead of %s, to print out the character string of the local time on your computer.

2. Declare and define a function, called MultiTwo(), that can perform multiplication on two integer variables. Call the MultiTwo() function from the main() function and pass two integers to MultiTwo(). Then print out the result returned by the MultiTwo() function on the screen.
3. Rewrite the program in Listing 15.3. This time, make a function that takes a variable number of int arguments and performs the operation of multiplication on these arguments.
4. Rewrite the program in Listing 15.3 again. This time, print out all arguments passed to the AddDouble() function. Does va_arg() fetch each argument in the same order (that is, from left to right) of the argument list passed to AddDouble()?

Previous | Table of Contents | Next