Programming Fundamentals  
Tutorial 07 - Functions

## Introduction

So far you have only been dealing with relatively simple problems that can be fully coded inside the statement block that represents main(). However, the programs you write in future may become too long and complex to realistically fit within main(). In this tutorial you will look at how to use **functions** to help structure your programs and make them more readable, as well as how to use them to process data in an efficient and reusable way. Finally, you will be shown how to use the problem solving processes you've previously learned to design and implement programs using functions.

## Functions

A **function** (or **procedure**) is a sequence of statements that does a specific set of operations in order to solve a problem or perform a task. As you already know, the main() function is where all programs begin running and you have also used functions such as getchar(), rand(), and fgets() amongst others. In fact, there are many pre-defined functions available in C/C++ that perform all sorts of different operations. One of the characteristics of a good programmer is an inquisitive and exploratory approach to solving problems, which includes finding out and understanding new concepts. Investigating and experimenting with these pre-defined functions is part of being a good programmer and is a great way to improve programming skills.

Your programs are not just limited to using pre-defined functions, however. C/C++ allows you to **write your own** too. By creating your own functions, you can **break up** large sections of a program into smaller, more easily coded and understood pieces. Functions can also be beneficial for operations that are repeated several times in your program - instead of repeating the code you can simply write a function to carry out that operation, then call the function **in place of the repeated code**. Most importantly, you can create functions that **accept values**, do some processing that is affected by those values, and **return a result**. Functions are an excellent way of structuring and organising programs, as well as an enormously powerful tool for processing data within a program.

### Declaring and Defining Functions

Before a function can be used it must first be **declared** and **defined**. Here's a simple example:

#include <stdio.h>

// DECLARE function

void DisplayHello(void); // function **PROTOTYPE (NOTE: semicolon needed)**

int main()

{

DisplayHello(); // CALL the function (jumps to and executes the code in the function)

printf("World\n");

getchar();

}

// DEFINE function

void DisplayHello(void)// function **HEADER (NOTE: no semicolon here)**

{

// function **BODY**

printf("Hello\n");

return; // program JUMPS BACK to where it was called from

}

The function DisplayHello() simply outputs "**Hello**" to the console window. The name of any function created should match what the function does as closely as possible, this makes programs much easier to read and understand. The two void keywords specify that the function neither returns nor accepts any values, more on this later in the tutorial. The main() function **calls** the DisplayHello()function, which means the program **jumps to the function** and starts executing the code inside the function. Once the end of the function is reached the program **jumps back to where it was called from** using the return keyword. The code continues from there by displaying "**World**" on the next line in main().

The **function** **declaration** is also called the **function prototype** and it **must** come **before** the function definition. It must also come **before it is called**, in this case before its use in main(), otherwise the compiler won't know what it is and will generate an error. Note that the declaration ends with a semicolon, if you miss this out there will also be a compiler error.

The **function definition** is where the code that performs the operations is located. The definition begins with the **function header**, which needs to be **exactly the same** as the function declaration but **without** the semicolon at the end. The header is followed by the **function body**, which is a statement block containing all of the function’s code. The function definition needs to be in the source file **after** the declaration and it cannot be within another function's statement block.

You normally declare a function **outside** of any other function's statement block, otherwise it will only be visible within that statement block.

You are not limited to calling a function from a specific place, nor is there any limit on the number of times that you can call a function. The below code shows how the DisplayHello()function can be called many different times from different places in the code:

int main()

{

DisplayHello(); // CALL function

printf("World\n");

DisplayHello(); // CALL function

printf("Mum\n");

for(int i=0; i<10; i++)

{

DisplayHello(); // CALL function

}

getchar();

}

## Using Functions to Process Data

The DisplayHello()function demonstrates how functions might be used to divide up a larger program into smaller, reusable sections. However, functions can be much more useful if you can **pass data** to them for processing and/or return the results of that processing back to wherever the function was called from.

### Passing Data to Functions

In order to send data to a function you need to provide a **parameter list**. This is a list of one or more values that you want to pass to the function. The parameter list is specified in a function prototype and header between the round brackets. Each parameter should provide a **type** and a **name** for the data that should be passed in, declared in the same way you would normal variables. The types of the parameter list must match with and be in the same order in both the declaration and the definition of a function. You can have as many parameters as you like, with any mixture of data types, with each parameter separated by a comma. It's also a good idea to name the parameters in a way that reflects how they are to be used.

Here are two examples:

#include <stdio.h>

**//** **DECLARE functions**

void DisplayFloatTimesTwo(float number);

void DisplayAddition(int number1, int number2);

int main()

{

// CALL functions

DisplayFloatTimesTwo(5.3f); // outputs 10.6

DisplayAddition(11, 23); // outputs 34

getchar();

}

// **DEFINE** functions

// Multiplies the passed float value by two and displays the result

void DisplayFloatTimesTwo(float number)

{

printf("%f\n", number \* 2.0f);

return;

}

// Displays addition of two integers

void DisplayAddition(int number1, int number2)

{

printf("%d\n",number1 + number2 );

return;

}

The DisplayFloatTimesTwo() function has a single float **parameter** called number and the DisplayAddition() function has two int **parameters** called number1 and number2. Parameters declared in this way are treated the same way, and have the same scope, as local variables, meaning they are only visible **within** the function. However, their initial values are set by the **arguments** which are **passed** to the function whenever it’s called. In the above case, ***5.3*** is passed to the DisplayFloatTimesTwo() function. In other words, the **argument** ***5.3*** is copied into the number **parameter**. Similarly the **arguments *11*** and ***23*** are **passed** to the DisplayAddition() function, meaning the number1 and number2 **parameters** have the values of those **arguments** assigned to them, respectively. These values can then be used by the functions as required.

Basically, **parameters** are **declared** variables waiting to be **defined** by the **arguments** which are passed to them.

You are not limited to literal values, indeed, functions are generally most useful when being passed variables. Here's a simple loop that passes variables to DisplayAddition():

int x=3, i;

for(i=0; i<5; i++)

{

DisplayAddition(x, i);

}

If **no** data is being passed to a function, you can simply put the keyword void in the parameter list just like the DisplayHello()function earlier. Remember that you must still include the brackets when calling a function with no parameters:

DisplayHello(); // Correct

DisplayHello; // Incorrect

### Returning Data from Functions

Once a function has finished, a calculated value might need to be **returned** back to the code that called it. If you need a function to return data, you must specify a **return type** for the function. A return type can be any data type, and is declared simply by placing it **before** the function name. The functions you have seen so far in this tutorial have used void as the return type, which means that no value is returned. By replacing void with a data type you declare that the function returns a value. The following code demonstrates a function that takes two integers, multiplies them together, and returns the result as another integer:

#include <stdio.h>

**//** **DECLARE functions**

int MultiplyNumbers(int number1, int number2);

int main()

{

int result;

printf("%d\n", MultiplyNumbers(10, 4)); // Outputs '40'

result = MultiplyNumbers(3, 5); // return value '15' copied into result

printf("%d\n", result);

getchar();

}

// DEFINE functions

// **RETURNS** multiplication of the two integers passed in as **ARGUMENTS**

int MultiplyNumbers(int number1, int number2)

{

int mult\_result;

mult\_result = number1 \* number2;

return mult\_result;

}

The MultiplyNumbers() function prototype and header are **preceded** by int, which means the function returns an integer value. Because the function is declared as having a return value you **must** return something of the correct datatype, in this case the result of the multiplication. This return value can be used directly, e.g.

printf(“%d”, MultiplyNumbers(10, 4));

or can be **copied** into a variable using an assignment operator, e.g.

result = MultiplyNumbers(3, 5); // return value '15' copied into result

Note that the following syntax will compile and run just fine:

MultiplyNumbers(666, 666); // DON'T DO THIS. Doesn't do anything useful

but since the result is not being used directly or stored in a variable this code effectively discards the result. The call to the function in this case just uses up time needlessly and is probably the result of an oversight when coding. Make sure you check that you are correctly using or storing any return values.

### Passing Arguments to Functions by Value and by Reference

In the functions that you have seen so far the arguments that are used in the parameter list when calling a function are **copied** into local variables. This is called **passing arguments by value**. If you change the value of the copy local to the function it **makes no difference** to the value of the calling argument. To demonstrate this you could rewrite the function DisplayFloatTimesTwo()as follows:

void DisplayFloatTimesTwo(float number)

{

number = number \* 2.0f;

printf("%f",number);

return;

}

As you can see, the function now actually changes number before displaying it. This doesn't affect the calling argument though, so if you called the function like so:

float my\_float = 6.3f

DisplayFloatTimesTwo(my\_float); // displays 12.6

Printf(“%f”, my\_float); // my\_float is unchanged, displays 6.3

So, my\_float isn't modified, only the copy local to the function has been changed.

Sometimes, it can be useful to modify an argument variable so that the changes are visible outside of the function. A likely situation where you might need this is if a function needs to return more than one value - the normal single return value would not be enough. In order to achieve this, you need to **pass the argument by reference** rather than by value.

Passing by reference means that instead of copying the data from the argument variable into a local variable, the function is instead given a **reference variable**, or **alias**, to the argument variable. This reference can be used to **directly access the external argument variable** from within the function.

In order to pass by reference, you need to declare a parameter using the dereference operator '\*'. The following example shows how this is done:

// **DECLARE** function

void SwapIntegers(int \*integer1, int \*integer2);

// DEFINE function

// Swaps 2 integer variables

void SwapIntegers(int \*integer1, int \*integer2)

{

int temp = \*integer1;

\*integer1 = \*integer2; // changes external variable directly

\*integer2 = temp; // changes external variable directly

}

The SwapIntegers() function clearly needs to return more than one value as it needs to return both swapped values, so it is appropriate to use two references. Both parameters have been declared and defined using 'int \**variable*', meaning '**integer dereference**', so instead of copying the arguments to the function they are **accessed directly**. If the function is called with reference operators (&) in front of the variables to be sent:

int x=10, y=5;

printf("%d,%d", x ,y ); // outputs 10,5

SwapIntegers(&x, &y);

printf("%d,%d",x , y ); // outputs 5,10

then you can see the argument variables outside of the function have been changed. If you declared this function **without** using integer references, then only the function’s local copies would be changed and the two variables used as arguments wouldn't be swapped as required.

Use caution when using references as it is easy to miss that a function is using pass by reference instead of by value. **The code to call the function looks exactly the same except for the reference operator ‘&’ in front of each variable**. Take the time to examine the function prototypes when using functions to make sure you know if it might alter the argument variables.

### Passing Arrays to Functions

The above examples show how to pass data via fundamental data types to functions. Sometimes data is stored in an array, so there needs to be a way to pass arrays to functions. The only requirement is for the array index operators '[]' to be included in the function declaration and definition:

Include <stdio.h>

const int MYINTEGERARRAY1\_SIZE = 4;

const int MYINTEGERARRAY2\_SIZE = 3;

//DECLARE function

void DisplayArrayElements(int an\_array[], int size);

int main()

{

int MyIntegerArray1[MYINTEGERARRAY1\_SIZE] = { 10, 23, 45, 78 };

int MyIntegerArray2[MYINTEGERARRAY2\_SIZE] = { 98, 54, 32 };

DisplayArrayElements(MyIntegerArray1, MYINTEGERARRAY1\_SIZE);

printf("\n");

DisplayArrayElements(MyIntegerArray2, MYINTEGERARRAY2\_SIZE);

getchar();

return 0;

}

// DEFINE function

// Displays all elements of array

void DisplayArrayElements(int an\_array[], int size)

{

int i;

for(i=0; i < size; i++)

{

printf("%d\n", an\_array[i]);

}

return;

}

The DisplayArrayElements() function in the above code has two parameters, the first is the array and the second is an integer size which will be given the number of elements in the array. The size parameter is needed in this case as **an\_array[]** does not have any information about its own size, and therefore you need to give the function this information. This is a fairly standard procedure when passing arrays to functions. When the function is called using an array **you do not use the '**[]' **syntax**. You **just** use the array name as the argument, in the above case MyIntegerArray1 and MyIntegerArray2.

One thing to be aware of when passing arrays to functions is that they are **passed by reference**, not by value, i.e. a copy is **not** made. If you change any of the array element values inside the function you will be **changing the external array**.

You can also pass multi-dimensional arrays to functions:

const int MY2D\_ROWS = 2;

const int MY2D\_COLS = 3;

void Display2DArrayElements(int an\_array[][MY2D\_COLS], int rows, int cols);

int main()

{

int My2DIntegerArray[MY2D\_ROWS][MY2D\_COLS] = { { 65, 23, 93 }, { 142, 527, 943 } };

Display2DArrayElements(My2DIntegerArray, MY2D\_ROWS, MY2D\_COLS);

getchar();

return 0;

}

// Displays all elements of array

void Display2DArrayElements(int an\_array[][MY2D\_COLS], int rows, int cols)

{

int i, j;

for(i=0; i < rows; i++)

{

for(j=0 ; j < cols; j++)

{

printf("%d", an\_array[i][j]);

}

printf("\n");

}

}

In this case, as there is a 2D array being passed to the function, you need to add another set of square brackets in the declaration parameter list to tell the compiler what to expect. As you can see the second set of square brackets contains a constant, in this case 'MY2D\_COLS'. This is so the compiler knows how many columns the array has and therefore which elements are being accessed. One consequence of this is that you will only be able to pass arrays with three columns to this particular function. The same requirement for constant index values applies to arrays with any number of dimensions - all the indices except for the first must be specified for the compiler to calculate the correct location in the array using the index operators.

### Functions Calling Other Functions

You are not restricted to calling functions from main(), in fact you can call functions from within any function. To demonstrate this, here is a function MultiplyPositiveNumbers()that uses another function called AddNumbers() to calculate the result:

int AddNumbers(unsigned int number1,unsigned int number2);

int MultiplyPositiveNumbers(unsigned int number1,unsigned int number2);

int AddNumbers(unsigned int number1,unsigned int number2)

{

return number1 + number2;

}

int MultiplyPositiveNumbers(unsigned int number1, unsigned int number2)

{

int i, mult\_result = 0;

for(i = 0; i < number1; i++)

{

mult\_result = AddNumbers(mult\_result, number2);

}

return mult\_result;

}

Although not the most efficient or general way of calculating a multiplication, this function does demonstrate the ability to call functions from within other functions. It also shows a powerful feature of functions - that they can often be **re-used** in many places in your code, or even potentially in other programs.

### Using Functions for Abstraction

Functions are often used to **hide the detailed implementation** of a program, allowing you to focus on the general higher level tasks that need to be performed without having to worry about those details. This idea is called **abstraction** and it is used extensively in computer programming. You have already come across the idea of abstraction when designing your programs using pseudocode and flow diagrams. You usually start with the overall task that needs to be performed, such as moving a game character:

1. While Character is Alive
   1. Get User Input
   2. Move Character Position
   3. Display Character
2. Display Character Death

As you can see, there are no details of **how** these operations are going to be done, just that they are and in what order. The overall process can therefore be designed first using this kind of abstraction, then the details can be filled in where necessary using stepwise refinement, e.g.

1. Get User Input
   1. Get Keypress
   2. Get Mouse Button
   3. Calculate Direction
   4. Return Direction (0 = none, 1=up, 2=down, 3=left, 4=right)

This idea of using abstract concepts to help in the creation of a design can be directly translated to the use of functions in your code. The above partial design might look something like this in code using functions:

void MoveCharacter(int direction);

void DisplayCharacter();

int CalculateDirection(int k, int b);

int GetUserInput();

int GetKeypress();

int GetMouseButton();

int main()

{

...// rest of code

while(character\_alive == true)

{

int move\_direction = GetUserInput();

MoveCharacter(move\_direction);

DisplayCharacter();

}

...// rest of code

}

int GetUserInput()

{

int keypress, button, direction;

keypress = GetKeypress();

button = GetMouseButton();

direction = CalculateDirection(keypress, button);

return direction;

}

As you can see the process of implementing the code can closely follow the abstract design steps, from both pseudocode or flow diagrams, by implementing them as higher level functions. Then the stepwise refinements of the design can be implemented inside those functions, with potentially even more detailed functions being called from inside the higher level ones.

### Using the Debugger with Functions

You can use the debugger in your functions in exactly the same way as within main(), i.e. you can set breakpoints in the function using **F5**, you can step through the code using **F7**, and you can use the **Watches** window to examine variables. However, there are a few more features of the debugger that are useful to know when using functions. Firstly, if the next line of execution is a function then you can press **F7** to execute the **entire function** and the pointer will move to the next line. If instead you want to **step-into** the function to see what it does, rather than finding the function to set a breakpoint, you can just press **Shift + F7** and the code pointer will **jump into** the function and continue debugging from there. If you want to **jump back** to the where the function was called from then you can use **Ctrl + F7** which completes the execution of the function and goes back to the line where it was called.

Note that sometimes when you press **F7** the code jumps to a file that you haven't written, this is usually because there is some other function on the same line such as printf(). In this case you can simply press **Ctrl + F7** to exit the function and go back to the calling line.

If you don't already use it you can also look at the **Watches** window when inside a function. This can be accessed via the **Debug** menu -> **Debugging Windows** -> **Watches**. This provides a complete list of all of the **local variables** in scope for that particular function. You can examine the local variables for main() using this window, as well.

Use the debugger to experience step by step exactly how functions work, it will give you a much better understanding of how your code works. One other simple idea that you can use is to set a breakpoint within a function to see if it is ever called. This quick idea can save you a lot of time tracking down bugs that stem from a function not being run when you expect it to.

## Exercises

#### Exercise 01

Write a program that has a function that displays your name. Use this function within main() **several** times, along with other text output of your choosing (perhaps a song or short story). Include at least one loop that calls the function multiple times.

#### Exercise 02

Create a program that has a function called Launch() that takes a single integer parameter. The function should use this value to count down from *n* to 1, then display "LIFTOFF!" Call the function from inside the main() function.

#### Exercise 03

Write a function that has a single integer parameter, and displays that many star characters in a row. Use this function in main() to first display five stars, then ask the user for a number and display that many stars.

#### Exercise 04

Create a function that has two integer parameters. The function should display a rectangle of stars, the first parameter determines how many stars across, the second how many rows. Test this function in main() by asking the user to enter two numbers.

#### Exercise 05

Start a new project and copy and paste the code from Exercise 04. Modify the function to take a third parameter that is a char variable, and display this instead of a star. Modify the calling code to ask the user for the character to display.

#### Exercise 06

Create a function that asks the user for a number, inputs a number then returns that number. The function should have no parameters and return an integer. Use this function in main() to ask the user for 2 numbers, then add them together and display them.

#### Exercise 07

Write a function that asks the user to enter 'y' or 'n'. The function should have no parameters and return a boolean, and should return true if the user enters 'y' and false if the user enters 'n', if neither it asks again. Create a short quiz in main() that uses this function to get responses to several yes/no quiz questions. Keep a score and display it at the end.

#### Exercise 08

Write a function that takes a single integer and returns a floating point value that is the area of a circle, with the passed integer as its radius. Test as usual.

#### Exercise 09

Create a function that has two integer parameters and returns a Boolean. Return true if the first parameter is a multiple of the second, i.e. it can be divided with no remainder, and false otherwise. Test as usual.

#### Exercise 10

Create a function called increase() that takes a single integer parameter and has a return type of void. The function should increment the input parameter by a fixed amount. Declare the function in such a way that the argument it receives is updated by the function, directly, e.g.

int x=5;

increase(x); // x should be greater than 5 after this statement is executed

#### Exercise 11

Write a function called halfdouble() that has 3 float parameters. The argument passed in the first parameter should be used to directly set the second parameter to half the first, and the third to double, e.g.

float a=2.4;

float b, c;

halfdouble(a, b, c); // a should remain as 2.4, b should be set to 1.2, c should be set to 4.8

#### Exercise 12

Design and implement a program that calls a function called menu() from main(), until menu() returns **false**. Menu should take no parameters and return a boolean. The menu() function should display a list of options:

1. New Game
2. Load Game
3. Options
4. Quit

and the user should be prompted to choose an option. The first two options should call **relevant functions**, though the load and new game functions simply need to display something and then return to the original menu. The Options choice should call a function that displays another menu:

1. Video
2. Sound
3. Gameplay
4. Back

This asks for user input then calls other appropriate functions that display what has been chosen. The final Quit option should be used to return a value to main in order for the program to exit.

#### Exercise 13

Implement a program that has a function that accepts an array of chars and outputs the array to the screen using printf(). Explain in your notebook why you think the size does not need to be passed to the function.

#### Exercise 14

Implement a program that has a function called GenerateLotteryNumbers() with an array of integers as a parameter. This function should generate 6 random numbers from 1 to 49, placing the random numbers in the array that was passed in. Create a second function called DisplayLotteryNumbers()which displays the first 6 numbers of the integer array passed in as a parameter. Test both of these functions by creating several integer arrays with 6 elements each. Test, and in your notebook explain what happens if the size of the array passed in is either bigger or smaller than 6.

#### Exercise 15

Implement a program that has several functions called Reverse\*(), where the \* means int, float, bool or char. These functions should have the appropriate type, array, and size parameters, and should all reverse the order of the elements in the array. Test as usual.

#### Exercise 16

Implement a program with a function that will accept a 2D array of floating point numbers. The function should add together the elements of the array and return the total. Consider how the function might cope with arrays with differing numbers of rows, and implement this extra if you can.