Introduction To Functions

Contents

- What is a function
- Returning from functions
- Function Arguments
- C++ Function Syntax
- Function Scoping
- Splitting up code

What is a Function?

- A function is a self contained section of code.
- Their purpose is to bundle a chunk of code together so that it can be used elsewhere.
- While loops let you run the same code multiple time, functions let you run the same code from multiple places.

Calling Functions

- Using a function is referred to as calling it.
- When you call a function, all the code inside the function executes.
- You can call a function at any point in your code.

Code Example – Pseudocode

```
FUNCTION PrintHelloWorld()
    Print(“hello world!”)
ENDFUNCTION

FUNCTION main()

    PrintHelloWorld()
    Print(“More Text”)
    PrintHelloWorld()
    Print(“Even More Text”)
    PrintHelloWorld()

ENDFUNCTION
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Code Example – C++

```cpp
#include <iostream>

void PrintHelloWorld() //<-- Defining a function
{
    std::cout << "hello world!" << std::endl;
}

int main()
{
    PrintHelloWorld(); //<-- Calling a function
    std::cout << "some specific text" << std::endl;
    PrintHelloWorld();
    std::cout << "some other text" << std::endl;
    PrintHelloWorld();
    std::cout << "even more text" << std::endl;
    PrintHelloWorld();

    system("pause");
    return 0;
}
```

Passing values into functions

- Functions have an optional parameter list.
- Parameters are variables that the function gets as input.
- This makes functions far more powerful as they can behave differently depending on what information comes into them.

Returning Values

- You can also return a value from a function.
- This means that you can give some data back to the code that called the function.
- Return values combined with function parameters means functions can both take input and return output.
- This makes functions

Returning values

- The return type for a function can be any primitive or compound data type.
  - Note that the return type cannot be an array.

- Not all functions have to return a value. You can use a return type of void to denote that the function will not return anything.

Defining a Function

- The syntax for a function is:

```
return type FunctionName(parameters)
{
      //Code goes here
}
```

- The function needs a return type, a unique function name and an optional list of arguments.

- The statements executed by the function are within a block of code known as the function body.

Example function

- Here is a very simple example of a C++ function.

```
int SumTwoNumbers(int a, int b)
{
    int result = a + b;
    return result;
}
```

Example function

- The first part of a function declaration is its return type.
- This indicates what kind of variable it will output.
- Can be void for no return value.

```
int SumTwoNumbers(int a, int b)
{
    int result = a + b;
    return result;
}
```

Example function

- Next is the function name.

- Function names follow the same rules as variables.

```
int SumTwoNumbers(int a, int b)
{
    int result = a + b;
    return result;
}
```

Example function

- Next is the parameter list.
- It looks like a list of variable declarations with commas in between.
- A function can have as many arguments as you would like.
- The parameters become variables inside the body of the function.

```
int SumTwoNumbers(int a, int b)
{
    int result = a + b;
    return result;
}
```

Example function

- The body of the function is the actual code that runs when you call the function.
- Any code statements can go inside a function, including calling another function.

```
int SumTwoNumbers(int a, int b)
{
    int result = a + b;
    return result;
}
```

Example function

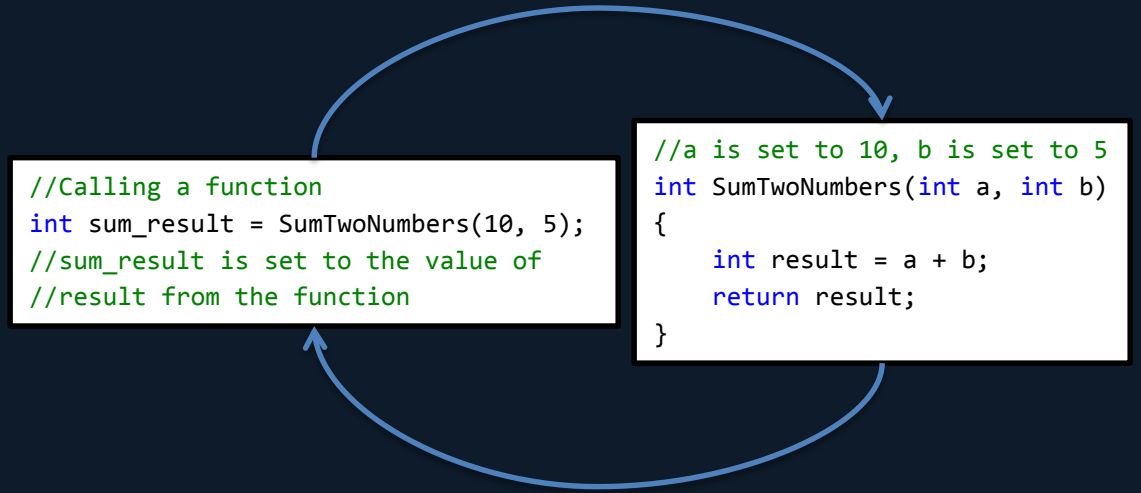
- When your function reaches a return statement, it exits immediately.

```
int SumTwoNumbers(int a, int b)
{
	int result = a + b;
	return result;
}
```

Calling a function

- A function is executed (called) – just type the function with the correct arguments.
  - Arguments must match the list of parameters (if any).

- This is how we would call the previous example functions in code:



```
//Calling a function
int sum_result = SumTwoNumbers(10, 5);
//sum_result is set to the value of
//result from the function
```

```
//a is set to 10, b is set to 5
int SumTwoNumbers(int a, int b)
{
    int result = a + b;
    return result;
}
```

Calling functions inside a function

- A function can call other functions

- The compiler reads through your code files top to bottom.
  - This means you can only call functions that have been defined earlier in the file

Calling functions inside a function

- Note that function1 cannot call function2, but function2 can call function 1.

```
void Function1()
{
    // error C3861: 'Function2': identifier not found
    Function2();
}

void Function2()
{
    // works fine
    Function1();
}
```

Declaring Functions (Prototyping)

- We can solve this problem by declaring our functions first.
- This is called prototyping our function.
- A function prototype does not have a function body.
- A prototype needs the return type, the name and the parameter list.

```
//Function prototypes
void Function1();
void Function2();
```

Function Overloads

- You can have multiple functions with the same name.
- This is called Function Overloading.
- Each function has to have a different, unique set of parameters.
- The return types can be different, but it cannot be the only thing that is different.

Function Example

```
#include <iostream>

//Function Prototype
int SumTwoNumbers(int a, int b);

void main()
{
    //Call our function and store the result
    int sum = SumTwoNumbers(10, 5);

    std::cout << sum << std::endl;

    system("pause");
}

//The function definition
int SumTwoNumbers(int a, int b)
{
    int result = a + b;
    return result;
}
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Local Storage

- What happens when a function is called?
- Local storage is made available for the function
- Any variables created in the function are lost after the function exits!
- In the previous example, you wouldn't be able to access "result" anymore, as the function has exited.

Scope Recap

- All variables in C++ have a lifetime, or scope.
- Any variable within brackets is only visible inside those brackets – including functions.
- Note that we can place braces anywhere in our code, as long as there is a matching opening and closing brace.

```
{
    int i = 5;
    std::cout << i << std::endl;
}

//Doesn't compile because i is out of scope
std::cout << i << std::endl;
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Passing Arrays

- Arrays work differently than most variables when passed into functions in C++
- When passing regular variables, the arguments are copied into the parameters of the function.
- Inside the function, you are working with copies of the values that were passed into it.

Passing Arrays

- When passing arrays, however the array is passed into the function without being copied.
- This means if you modify the contents of the array inside the function, whatever array was passed in will be modified.
- This isn’t true for other variables.

Passing Arrays

- When passing arrays into functions, you should always pass both the array and the length.

```
void AddToArrayValues(int array_var[], int array_length)
{
    for ( int i = 0 ; i < array_length ; ++i)
    {
        array_var[i] = array_var[i] + 1;
    }
}
int main()
{
    int int_array[] = {5, 6, 7, 8, 9, 10};
    AddToArrayValues(int_array, 6);
    //int_array is now {6, 7, 8, 9, 10, 11}
}
```

Splitting up Code

- Functions allow you to split up your code into self contained chunks.
- If you ever find yourself writing the same code multiple times, or copy/pasting chunks around, this is a big hint that you should pull that code out into a function.
- Later, as your programs grow, you will want to further split up groups of functions and data into their own modules.

Splitting up Code

- There are a few core reasons to want to split up your code.
  - Reuse
  - Division of Labour
  - Mental Clarity

Reuse

- Splitting code out into its own function means that you can call that function from multiple places in your code instead of having to retype it.

- With a bit more work, you can make functions that can be copied over to other projects so that you don’t have to duplicate the work, even then.

Division of Labour

- Most large projects involve many people working on the one code base.

- Splitting up your code into separate chunks means that different people can work on separate parts of the code without stepping on each others toes.

Mental Clarity

- Programming is hard
- When programming complex systems, its easy to get confused and make mistakes
- Splitting code into chunks means you only have to think about the small section of code you are working on.

Summary

- Functions let you group code into self contained chunks.
- They are used to either perform tasks or calculate values.
- Functions can take parameters as input and return values as output.
- In C++ we need to need to say what data types the input and output are.
- Splitting code into functions lets us reuse them, work with others and helps us think clearly about our code.

References


Canberra Institute
of Technology


aie
SPECIALIST EDUCATORS IN
GAMES, ANIMATION & FILM VFX