Functions

Topics:

+ Parameters & Arguments
+ Function Prototypes
+ Function Definitions
+ Function Calls
+ Function Overloading

Resources:

- `main.cpp`

Introduction

A function is used to modulate a program so that it is more managable and readable. Typically, they perform tasks that can be reused in multiple files and/or multiple times within a file. They structured similar to mathematics functions in that they take inputs and produce an output; although, you can write C++ functions that neither take inputs nor produce an output.

Parameters & Arguments

A function, as stated, can take inputs called *parameters* and can produce an output called a *return-value*. Parameters are local variables of the function that are assigned values during the function call. However, a parameter can come in multiple forms. The way the parameter is formatted determines the data type and the types of arguments it can accepts. There are four formats we will discuss.

The first format accepts arguments passed by value. Its syntax is

$$[\texttt{const}]^{?}\ \textit{data-type identifier}$$

This format is similar to a variable declaration. It accepts arguments that match the data type of any form excluding arrays and pointers (although, you can pass an element of an array or a deferenced pointer to it). The r-value of the argument becomes the r-value of the parameter [literals and expressions are not variable; but, they are provided temporary storage during the transfer to the parameter]; however, the l-values of the argument and parameter are different. This means changes made to the parameter within the body of the function are not made to the argument in the calling function if it were a variable. Last, if the parameter is constant, the parameter cannot be assigned a new value within the body of the function. Trying to an assignment with a constant parameter will cause an error.

The second format accepts arguments passed by reference. Its syntax is

$$[\texttt{const}]^{?}\ \textit{data-type}\&\ \textit{identifier}$$

This format accepts arguments that are variables of the same data type only when it is not constant. The l-value of the argument becomes the l-value of the parameter which means the argument and parameter are aliases. Thus, any changes made to the parameter within the body of the function will be made to the argument in the calling function. However, if the parameter is a constant reference, it can accept any form of an argument excluding arrays and pointers that match the data type of the parameter. Likewise, a constant reference parameter cannot be assigned a new value as expected.

The third format accepts arguments as arrays. Its syntax is

$$\left[\texttt{const}\right]^{?}\ \textit{data-type identifier}\texttt{[]}$$

When you pass an array as an argument, you just use its name. Also, arrays are always passed by reference. We will discuss arrays in more details in the arrays lecture.

Finally, the fourth format accepts arguments by address. Its syntax is

$$\left[\texttt{const}\right]^{?}\ \textit{data-type} * \left[\texttt{const}\right]^{?}\ \textit{identifier}$$

This format only accepts addresses; hence, it does not accept arguments that are literals and expressions. The details on this format will be discussed in the pointers lecture.

Function Prototypes

A function prototype is a function declaration. It tells the compiler that the function will be defined later in the program. Thus, it allows you to call the function before its definition. For instances, the code below will represent function prototypes and definitions as comments with the exception of the main function. The function C and main can call the function B since the function prototype of B is written before their definitions; although, the function definition of B is written after them. Likewise, the function D can call function B because it is defined after the function definition of B; whereas, the function A cannot call function B because it is defined before the function prototype of B.

```
//Function Definition of A
//Function Prototype of B
int main()
{
 return 0;
}
//Function Definition of C
//Function Definition of B
//Function Definition of D
```

To reiterate, a function cannot be called by any function that is definied before its definition unless its function prototype is provided before their function definitions. Knowing the above information, for each function A, B, C and D determine which functions can call them.

Now, the syntax of a function prototype is

$$\textit{return-type identifier}\left(\left[\textit{parameter}\left[,\textit{parameter}\right]^{*}\right]^{?}\right)\texttt{;}$$

where *return-type* is normally a data type or the keyword `void`. The function prototype is simply the *function header* followed by a semicolon where the function header is the first line of the function definition. It provides the essential properties of a function that are necessary to call it properly. An important note about function prototypes to consider is that the identifiers of their parameters can be omitted, and their parameter identifiers do not have to be the same as the parameter identifiers of the function definition. Furthermore, function prototypes can be written in both local scopes and global scope.

Function Definition

A function definition gives the details of the function. Its syntax is

$$\textit{return-type identifier}\Big(\big[\textit{parameter}\big[, \textit{parameter}\big]^{*}\big]^{?}\Big)$$
$$\{\textit{body}\}$$

where *body* is a collection of statements. In fact, you have been writing a function definition in every program thus far; you have been defining the main function. To help us breakdown the essential aspects of a function definition, let us examine the following function

```
int main()
{
 string name;
 getline(cin,name);
 cout << "Hello, "<< name << "\n";
 return 0;
}
```

The above function is requesting a name from the user, and then, it displays the input of the user preceded by the string `"Hello,"` on its own line. The description is of the function body. It is the content enclosed in curly braces. A function body can consist of practically any type of statement. It cannot, however, contain a function definition. Functions can only be defined in global scope. The rest of the function, the first line, as stated, is called the functon header. To reiterate, it provides all the essential components necessary to call the function. But its components also tells you what is required and prohibited in the body of the function.

First, the function header begins with the return type. If the return type is not void, the function must have at least one reachable return statement within its body. The syntax of a return statement is

$$\texttt{return}\ \big[\textit{argument}\big]^{?}\texttt{;}$$

where the type of the argument must either match the return type or be (implicitly or explicitly) typecast to the return type. Notice, that the argument is said to be optional in the syntax definition; however, it is mandatory for non-void functions and prohibited for void functions, if you choose to include return statements in those functions. As implied a function body can consist of several return statements; however, a function terminates as soon as it executes any return statement. It does not matter if there is additional code following the return statement in the function body. For instances, the following function has several return statements but only the code before the first return statement and the first return statement are executed.

```
int main()
{
 cout << "Line A is executed\n";
 return 0;
 cout << "Line B is executed\n";
 return 0;
 cout << "Line C is executed\n";
 return 0;
}
```

Since only the code before the first return statement and the return statement will be executed, only the string `"Line A is executed"` will display if this program runs (remember you would have to include the iostream library to run it). Next, notice in both examples the arguments of the return statements are all int literals. This is because the return types of the functions are int. Hence, regardless of how many return statements are in the body of the function, the type of arguments of those return statements must match the return type. This does not mean that the arguments must be the same value.

After the return type, the function header has an identifier and a parameter list that is enclosed in parentheses. The identifier and the parameter list are used to identify the function when it is called. Now that we know its components, let us define a few functions.

Example 1. A double function named `PI()` that takes no parameters. It returns π with five decimal places.

```
double PI()
{
 return 3.14159;
}
```

Example 2. A int function named `DTS()` that takes two int parameters. It returns the difference of the second parameter squared from the first parameter squared.

```
int DTS(int x,int y)
{
 int r = x * x - y * y;
 return r;
}
```

Example 3. A void function named `Triple()` that takes an int reference parameter. It assigns the parameter three times its value.

```
void Triple(int& n)
{
 n *= 3;
}
```

Example 4. A void function named `Print()` that takes two string parameters. It displays the parameters on their own line with a space between them.

```
void Print(string a,string b)
{
 cout << a << " "<< b << "\n";
 return;
}
```

Function Call

A function call is an evaluation of a function. It is similar to substituting actual numbers for the variables in mathematical functions. Its syntax is

$$\textit{function-name}\Big(\big[\textit{argument}\big[, \textit{argument}\big]^*\big]^?\Big)$$

where

- *function-name* must match the identifier of the function in the function definition.
- The amount of arguments must match the amount of parameters of the function definition.
- Each argument corresponds to the parameter in the same position as it in their respective list; that is, the first argument corresponds to the first parameter, the second argument corresponds to the second argument and so on.
- The type of the argument must match the type of its corresponding parameter.

Furthermore, if the return type of the function is not void, the function call can be an argument. Additionally, function calls are performed in local scopes only.