CMSC 202 Spring 2021

Lab 02 – C++ Primer and Functions

Assignment: Lab 02 – C++ Primer and Functions

Due Date: On Discord from February 8th to February 11th

Value: 10 points

1. Overview

In this lab you will:

- **Take a prelab quiz on Blackboard before Monday at 10am**
- Practice working with control structures
- Write a program that calls a function
- Practice using console input and output

2. Introduction to Functions

At the top level, a computer ***program*** is a collection of instructions that performs a specific task. The more complex the task, the more important it is for us to break the program down into more manageable parts. These subparts may include obtaining the input data, calculating the output data, or displaying the output data. We call each of these ***functions***. From a vocabulary standpoint, there are several synonyms for function such as ***modules***, ***procedure***, ***subprogram***, or ***method***. The syntax for a normal function looks like this:

```
returnType FunctionName (Parameter List) {
}
```

The function name is a normal identifier and follows the C++ rules of identifiers. Normally, we use camel case to create meaningful names for our functions for example, **`printName`** or **`addOne`**. As a reminder, those rules include:

1. Keywords cannot be used as an identifier.
2. Identifiers are case-sensitive and therefore **`addOne`** is different than **`addone`**.

3. Only alphanumeric and the underscore are allowed in identifiers in C++. However, an identifier cannot begin with a number.

The return type is the data type that the function returns when it runs. There are two options regarding function return types:

1. A ***value-returning function*** can return a normal data type such as an integer, a string, a specific object, or any other user defined data type.
2. A ***void function*** does not return anything. These are normally used to output something that does not require something to be returned from a function.

Parameters are used to pass information to a function. Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: **`int x`**) and acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from. For your information, **`main()`** is a function and is required by all C++ programs.

3. Predefined Functions vs. User-defined Functions

C++ comes with a wide variety of libraries of predefined functions that are available for use with minimal effort. For example, math functions would be in the math library and I/O functions would be in the **`iostream`** library. In order to access those predefined functions, you just need to **`#include`** the library.

For example,

```
#include <cmath>
using namespace std;
int main () {
    double d = 4.0;
    sqrt(d); //Predefined function call from cmath
    exp(d); //Predefined function call from cmath

    return 0;
}
```

As with all functions, a predefined function can either return nothing (a void function) or it can return a value. Here are some of the more commonly used predefined functions in C++:

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0,3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

If there is no predefined function that performs a desired task, then the programmer can design their own function. These are more commonly called user-defined functions. As with the predefined functions, user-defined functions can either return nothing (i.e. void function) or some value. Importantly, a function can only ever return **one** thing.

Value-returning functions are most commonly used in three places:

1. Assignment statement (The right side of an assignment operation)
2. Output statement (ex. a `cout` statement)

3. Argument (actual parameter) in another function call (For example, **`addPos(abs(x),abs(y))`**)

There are three steps involved with creating a user-defined function:

1. Prototype
2. Definition
3. Call

4. Function Prototypes

As mentioned above, a user-defined function is described in two parts. The first part is called the ***function declaration*** or ***function prototype***. The function prototype is the function heading without body of function. The normal prototype is exactly one statement (ending in a semicolon). The prototype indicates the name of the function, the type of data that it receives, and the number, data type, and order of the parameters.

A function declaration or prototype might look like this:

```
double addTwo(double val1, double val2);
or
double addTwo(double, double);
```

As with all functions, the first word is the return type. In this case the function returns a double. The second word, **`addTwo`**, is the identifier for this function. The third part, **`double val1`** and **`double val2`** are the parameters for this function (the identifiers **`val1`** and **`val2`** are optional in the prototype but the data types are mandatory). This function requires that two numbers are passed to it. Regarding the parameters, you do NOT need to include their name, just their data type.

Function prototypes are not always required if the function is defined above main but including them anyway is considered best practice. It can help reduce the number of program errors because the compiler will check to make sure that you used the correct function, and the correct number and type of parameters. It also makes sure the function returns the correct data type.

The easiest way to create a function prototype is to code the entire function and then to copy and paste the function header above main and add a semicolon at the end.

The prototype also indicates the ***function signature***. The function signature is defined as the identifier of the function and the parameter list of the function. A function signature does **NOT** include the return type. All functions in the same scope must have unique signatures.

5. Function Definitions

A function definition is where the actual body of the code is included. The definition is made up of two parts:

1. ***Function Header*** – this is the information that appears in the function prototype including the return type, name, and parameter list of the function.
2. ***Function Body*** – the body of the function includes all of the code required to complete the task. A function body is enclosed in curly braces {} and any variable declared in the body of the function is local to the function (unless returned).

For example,

```
double addTwo(double val1, double val2){ //Function Header
    return (val1+val2); //Function Body
}
```

A ***return*** statement consists of the keyword return followed by an expression. In the example above, the return statement is the entire body of the function.

Parameters are defined in two ways:

1. Formal parameter (or parameter): variable declared in heading
2. Actual parameter (or argument): expressions, variables, or constant values listed in function call

6. Function Calls

A function call is when the function is used in our code. When a function is called, the parameter list of the call must match the function prototype in the type, order, and number. As previously mentioned, a function call occurs in one of three places:

1. Assignment Statement

```
double newDouble = addTwo(3.6, 5.2);
```

2. Output Statement

```
cout << addTwo(3.6, 5.2) << endl;
```

3. Argument in another function call

```
addTwo(addTwo(3.1,5.1), 5.2);
```

Return statements will force a function to stop and control will return to where the function call was.

7. A Simple Example

Here is a full example of a function prototype, definition, and call.

```
#include <iostream> //Library import
using namespace std; //Defining namespace for std

double addTwo(double, double); //Function Prototype

int main () {
  double d = 4.0; //Declare and Initialize double
  double e = 2.5; //Declare and Initialize double
  cout << addTwo(d, e) << endl; //Function Call
  return 0;
}

double addTwo(double val1, double val2){ //Header
  return (val1+val2); //Function Body
}
```

8. Function Preconditions and Postconditions

In the coding standard for CMSC 202, we discuss the importance of documenting our functions. The coding standards require that you write a function declaration comment that includes two pieces: the precondition and the postcondition.

The precondition states what is assumed to be true when the function is called. The function cannot be expected to perform correctly, or sometimes at all, unless the precondition holds. For example, if a function is going to divide by a parameter `x`, part of the precondition would be that `x` cannot equal zero.

The postcondition describes the effect of the function call; that is, the postcondition tells what will be true after the function is executed in a situation in which the precondition holds. For a function that returns a value, the postcondition will describe the value returned by the function. For a function that changes the value of some argument variables, the postcondition will describe all the changes made to the values of the arguments.

9. Prelab Quiz

When you have finished reading over this prelab, you need to complete your prelab quiz on Blackboard under “Assignments” then “Labs”. You have until Monday at 10am to complete your prelab quiz.