

<b>Due</b>
3:00pm Friday, 9/28
<b>Turn-in:</b> <i>This completed lab description and print-outs of your 5 programs</i>

---

---

## 1 Introduction

The field of Computer Science has contributed the notion of **top-down** design. This problem solving methodology consists of breaking a big problem into a series of simpler to solve sub-problems. The sub-problems are then combined into a solution for the original problem. This technique is still of fundamental importance in an **object-oriented** paradigm.

An implication of top-down design manifests itself as programs that call functions, which in turn call other functions. It is therefore important to understand how functions are implemented in **C++**. This lab focuses on:

- Calling standard library functions
- Writing our own functions
- Parameter passing
- Return values
- Reference parameters
- Recursive functions

Create a directory called **Lab2** off of your **CSC112** directory to contain all your lab 2 program files (5 total).

## 2 Calling Functions

The **C++** language consists of certain well defined constructs (i.e. loops, variables, syntax, ...) as well as a set of **standard** libraries. These libraries contain prewritten functions that are common to many programs. Some example functions are:

`abs(i)`    `cos(x)`    `pow(x, y)`    `sqrt(x)`

Using library functions in your programs is encouraged because they are correct, efficient, and standard across all official **C++** implementations. Additionally, standard functions are simple to use. There are three steps in using standard library functions:

1. including the appropriate header file
2. making the correct **function call**
3. linking to the appropriate library

The appropriate header files most likely include:

`iostream`    `stdio`    `stdlib`    `math`

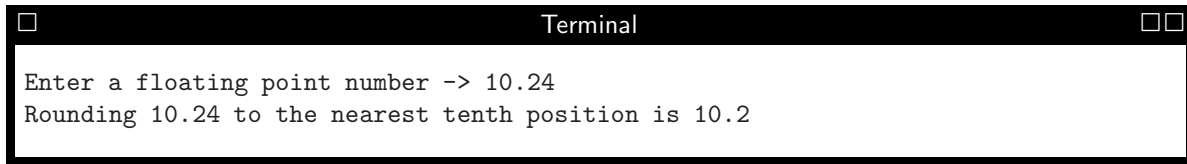
The corresponding libraries are automatically linked (combined) when using **g++** for linking.

### 2.1 Program 1

You are to use the Unix command **man** to learn about the standard library function **floor**. Pay attention to the calling sequence. What is:

1. The return type \_\_\_\_\_
2. The calling parameter(s) \_\_\_\_\_
3. Case(s) for function failure? \_\_\_\_\_

Write a program that prompts the user to input a floating point number, reads this input, then uses `floor` to round the number to the tenth position (*a decimal of 5 in the hundredth position shall be rounded up*). For example, if the user enters 10.24 the output is



```
Terminal
Enter a floating point number -> 10.24
Rounding 10.24 to the nearest tenth position is 10.2
```

Avoid using a selection structure for rounding since comparing floating point numbers can be unreliable (more about this later in the semester). A simple math function using `floor` can be used to quickly round a floating point number. The source code for this program should reside in a file labeled, `lab2_1.cpp`

### 3 Writing Functions

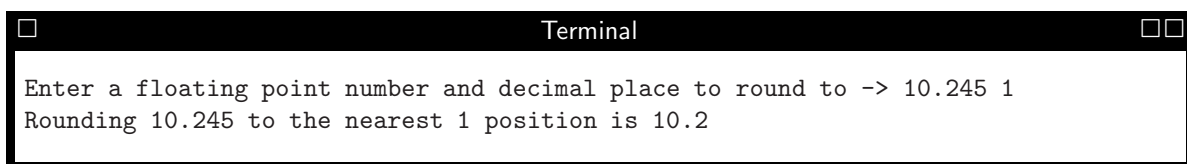
When using the **top-down** design methodology you will need to implement and use functions of your own design. In fact, we should strive to design and implement well-defined, general purpose functions that *can* be used as frequently as needed. It is better to write and test a function once and reuse than constantly rewriting the same function over and over. When writing your own function you need to decide three things,

- The name of the function
- Function return type: the value returned by the function, or the function output
- Function parameters: information supplied to the function, or function input

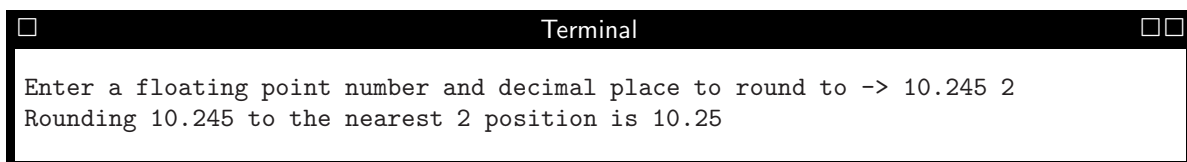
The combination of these three items constitute the **function signature**. The name of the function should reflect the purpose of the function. For example, `sqrt`, `cos`, `abs`, ... The value returned by the function should be the **result** or the value computed by the function. The function parameters are the value(s) you want to use during function execution. You should limit the number of parameters you pass to a function so the function signature fits on one line.

#### 3.1 Program 2

Create a program that rounds a floating point number to the  $n^{th}$  decimal position. The program should prompt the user for a `double` value (number to be rounded) and an `integer` (the decimal position to round to). The program should then display the original number, the decimal rounding position, followed by the rounded value. For example,



```
Terminal
Enter a floating point number and decimal place to round to -> 10.245 1
Rounding 10.245 to the nearest 1 position is 10.2
```



```
Terminal
Enter a floating point number and decimal place to round to -> 10.245 2
Rounding 10.245 to the nearest 2 position is 10.25
```

Rounding **must** be done in a separate function. **Write the function signature in the space below.**

---

The source code for this program should reside in a file labeled, `lab2_2.cpp`

## 4 Scope

Scope in C++ determines where objects are visible within your program code. C++ scope rules provide the programmer with many useful programming features.

### 4.1 Local Scope

The C++ language specifies that objects declared in a function are only defined within that function. Function parameters can only be used within the function since C++ uses *call by value*. That is, parameter values are *copied* into new memory locations when the function is executed. Parameters and objects are said to be **local** to the function.

Scope rules in C++ state that a local object can be used only in the code block and nested code blocks of the block in which it has been defined. One of the benefits of the local scoping rule is **object name reuse**. That is, objects with the same name can appear in multiple functions, i.e., `int i`, `cnt`, ... Name reuse is permitted in different code blocks, even if one of the blocks is nested within the other block. In this case, the variable in the *closest* enclosing code block is used. Therefore the following is legal

```
int main()
{
    int i;
    {
        char i;
    }
}
```

Name reuse is *not* permitted within the same code block.

### 4.2 Global Scope

Global scope in C++ is defined as those parts of your program that are not contained within any code block. Function prototypes must occur in the global scope. Object definitions and declarations can also occur within the global scope and are referred to as **global objects**.

Scoping rules for global objects are similar to the scope rules for local objects. Global objects may be referenced by using the object's name in any desired block that occurs after its definition. Name reuse is permitted, but not within global scope. That is, two global objects with the same identifier cannot occur at the global scope level. If an object at local scope reuses the name of an object at global scope, the global scope object is available within the local scope by using the unary **scope operator**, `::`

Programmers can lose track of what a global object is supposed to represent. For example, a function `f()` does not use or change the values of any global variables. However, the function `f()` calls another function which calls another function, which calls another function which *does* change the value of a global object. Programming methodologies try to avoid this type of problem by **limiting** the use of global objects.

## 5 Reference Parameters

The *call-by-value* methodology for function parameters means that the calling parameters will have the same values after the function call as they did before the function call. For example,

```
float area(float length, float width);
int main ()
{
    float l = 4.0, w = 12.4;
    float value = f(l, w);
}
```

In this example the function `area()` will not change the values of `l` or `w` and only changes a value in the calling function using the functions return value. There are no **side effects** caused by calling function `area()`.

Sometimes side effects are needed, like when a function needs to return multiple values. The C++ mechanism is referred to as **reference parameters**. A reference parameter means that the **address** of the calling parameter is used instead of the value of the parameter. This gives the called function access to the calling parameter, which if modified in a function remains changed following the function call.

## 5.1 Program 3

Write a program that reads three integers from standard in and writes the values in **ascending sorted** order to standard out. Include a function whose prototype is:

```
void sort3 (int& a, int& b, int& c);
```

The function `sort3()` should take three references to integer parameters, swap parameter values so that when the function terminates,

$$a \leq b \leq c$$

The source code for this program should reside in a file labeled, `lab2_3.cpp`

## 5.2 Constant Reference Parameters

Another important use of reference parameters is efficiency. Parameters are **always copied** when a function is called. This is true even for reference parameters, but in this case the address of the calling parameter is copied instead of its value as it is with *call-by-value*. If a calling parameter occupies a significant amount of memory it is significantly more efficient to copy the beginning address of the block of memory rather than the block itself.

If a function is called with a reference parameter for efficiency and not for modification the potential exists that the calling object may be accidentally modified by the function. When this happens programs frequently develop **undocumented program features** or **bugs** that are difficult to track down. It is an easy matter to determine if a reference parameter is modified by a called function. In fact, even a compiler can do it. There are two ways a reference parameter can be modified by a function.

- The reference parameter appears on the left hand side (lhs) of an assignment statement
- The reference parameter is used to call a function intended to modify the parameters value

Standard C++ compilers insure that neither of these things happen when the **const keyword** is prepended to the reference parameter that should not be modified by the function. For example,

```
void function(const LargeObject& large);

int main ()
{
    LargeObject big;
    function(big);

    return 0;
}
```

### 5.2.1 Program 4

You are to write an efficient function that computes and returns the Euclidean distance between two, two-dimensional points  $(x_1, y_1)$  and  $(x_2, y_2)$ , where the distance is  $\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$ . Assume a significantly large block of memory is required to represent  $x_1, y_1, x_2,$  and  $y_2$ . Write the function signature below

---

The source code for this program should reside in a file labeled, `lab2_4.cpp`

## 6 Recursion

Recursion is the ability of a language for a function to call itself. Recursive implementations frequently results in elegant functions consisting of only a few lines of code. There are two components of recursive functions, the **termination part** (base case) and the **recursive part** (induction step). The termination part determines if the recursion should stop and the recursive part makes the recursive call with parameter values that are closer to termination. For example in a recursive implementation of a factorial function, we may have:

```
unsigned int factorial (unsigned int n)
{
    if (n == 0)
        return 1;
    return n * factorial (n - 1);
}
```

Rewrite the above function using a single *conditional statement*. Place your answer below.

-----

-----

-----

-----

-----

-----

### 6.1 Program 5

Write a program that prompts the user for  $n$  the number of terms used in approximating Euler's number:  $e = 2.71828182845904523536028747135266$ . Have your program call a recursive function to approximate  $e$  using the geometric series:

$$e = \sum_{i=0}^n \left( \frac{1}{i!} \right)$$

The factorial **and** the  $e$  approximation **must** be recursive; therefore, this program will have two recursive functions and **no loops**. In addition, the output **must** display more than the default number of decimal places. The source code for this program should reside in a file labeled, `lab2_5.cpp`

## 7 Programming Points

You **must** adhere to all of the following points to receive credit for this lab assignment.

1. Create a directory `Lab2` off of your `CSC112` directory to store all your files for this assignment (`lab2_1.cpp`, `lab2_2.cpp`, `lab2_3.cpp`, `lab2_4.cpp`, and `lab2_5.cpp`)
2. All programs must adhere to documentation style and standards. Don't forget function headers and pre/post-conditions.
3. **Turn-in** this completed lab description (be certain all questions are answered) and print-outs of your 5 programs. In addition, submit your 5 programs using `blackboard.wfu.edu`.