Programming 11 Pre-Lab - C++ Functions

Already Covered

In the past labs, we have already covered the following:

- The parts of a C++ Program
- Data Types
- Input/Output Formatting
- Mathematical Operators
- Loops
- Making Decisions
  - If statements
  - If – else statements
  - Case statements
    - Break
- File Streams
  - Inputting data from files
  - Outputting data to files
- Arrays

Look to the those resources for more information and guidance on these individual concepts.

Lesson's Objectives

This will go over the following:

- Functions
  - Placement
  - Declaration
  - Definition
  - Invocation (Call)

Functions

Functions in C++ work similarly to other programming languages. They are segments of code that can take in inputs and provide an output. We will be sticking with the basic raw structure of functions in C++ meaning that they are to be defined in the program, instead of creating a custom library of them, and their output is limited to a single parameter.

Placement

There are 2 structural placements for functions in C++. You can define them **before** `main()` or **after**. This is based mainly on preference, but also on design needs. Defining the functions **before** `main()` is becoming more prevalent, but **WE WILL PUT THEM AFTER**. This is because it requires an extra step that you must be familiar with.

The basic structures look like this:

Before `Main()`	After `Main()`
1. `Libraries & Directives` 2. `Global Variables` 3. `Functions` 4. `Main()`	1. `Libraries & Directives` 2. `Global Variables` 3. `Function Declarations` 4. `Main()` 5. `Functions`

There is no inherent issue with using either. It typically depends on what's most valuable, what's happening in the functions or in main(). The more important one would be at the top and therefore most quickly accesible.

It may not look like it would matter, but if you consider a case with multiple functions, the important of placement in ease of access/editing becomes a little more apparent.

Before `Main()`	After `Main()`
1. `Libraries & Directives` 2. `Global Variables` 3. `Function 1` 4. `Function 2`	1. `Libraries & Directives` 2. `Global Variables` 3. `Function Declarations` 4. `Main()`

5. `Function 3`	5. `Function 1`
6. `Function 4`	6. `Function 2`
7. `Function 5`	7. `Function 3`
8. `...`	8. `Function 4`
9. `Function n`	9. `Function 5`
10. `Main()`	10. `...`
	11. `Function`

You are required to use the *after* main() structure though because it requires declarations which you need to recognize and be able to use. If you use the before main, then you are specifically avoiding showing an understanding of a concept and will lose credit for that.

Understanding a Course Requirement Show Correct Answer Show Responses

For this course, you will be required to place functions ________ `main()` because doing so shows that you know about ________ as well as creating the function definition and invocation, which can't be avoided.

Structure of Functions

For that placement, there are 3 parts to that structure:

- **Declaration**
  - This is where it is initially declared, letting the program know a user defined function exists.
- **Definition**
  - This is where the function is defined, providing the code for what it's supposed to do.
- **Invocation**

   - This is where it is used in the program, either in main() or another function.

We'll be using the following example to describe these:

```
1.  #include <iostream>
2.  using namespace std;
3.  float find_min(float, float, float);
4.  int main ()
5.  {
6.    float x, y, z, M;
7.    cout << "Enter value for x: "; cin >> x;
8.    cout << "Enter value for y: "; cin >> y;
9.    cout << "Enter value for z: "; cin >> z;
10.   M = find_min(x, y, z);
11.   cout << "The minimum is: " << M << endl<< endl;
12.   return 0;
13. }
14. float find_min(float x1, float x2, float x3)
15. {
16.   float min;
17.   if (x1<x2)
18.     min=x1;
19.   else
20.     min=x2;
21.   if (x3<min)
22.     min=x3;
23.   return min;
24. }
```

Which is which Show Correct Answer Show Responses

Maybe you will need to come back to this question, which is fine, but at least try it first based on the very general descriptions given above.
Match the part of a function to it's line (or first line) in the above code.

Premise			Response	
1	Declaration	→	A	14

2	Invocation	→	B	6
3	Definition	→	C	23
			D	3
			E	10

Declaration

A C++ project is compiled from top to bottom. Having the declaration of the functions **above** `main()` simply informs the compiler that there are function invocations that it will see in main() or other places that it wouldn't otherwise recognize. If you didn't declare it, the compiler might interpret it as a typo or other error and stop.

This is again one of those things where modern compilers are getting better about pre-reading the whole code then officially compiling. You may be able to get away without it but you can't guarantee anyone else will so make sure you, ***as required by this course***, use the declaration and put your functions after main().

From above, the declaration is:

```
float find_min(float, float, float);
```

A declaration has 3 parts:

- **Data Type** - `float`
  - This corresponds to the data type of the output from the function
  - If there is no output, the type is `void`
- **Name** - `find_min`
  - This is the name by which it will be called in the program
- **Input Parameters** - `(float, float, float)`
  - These are the data types of the input parameters

        - They don't all have to be the same type, they just happen to be for this case. You can have different data type inputs.
    - Names for these input parameters are **not** necessary in the declaration. The compiler is simply setting aside data for them.

Definition

The definition is the essentially the whole function. It has, like so many things apparently, 3 Parts of concern: the Header, the Body, and the Return.

Header

The header is one of the most important, as it will be where most errors occur. It essentially is a repeat of the declaration, defining the output data type, function name, and required input parameters. Examine the similarities:

- Declaration: `float find_min(float, float, float);`
- Definition's Header: `float find_min(float x1, float x2, float x3) {...`

The **definition** does not need the names of the parameters and is a single statement, so ends with a semi-colon, but otherwise needs to match the header information. The **Definition's Header *must*** include the parameter names and is followed immediately by the **body { }**, so should **NOT** be followed by a semi-colon. You can include the names of the input parameters in the declaration as well but it's not necessary. Personally, I typically do because it becomes mostly a copy paste between them but I want you to understand that they're not required.

Body

The body is the code that will run when the function is invoked. It should make use of the input parameters in that code.

```
float find_min(float x1, float x2, float x3)
{
  float min;
  if (x1<x2)
    min=x1;
  else
    min=x2;
  if (x3<min)
    min=x3;
    return min;
}
```

Return

The return is part of the body that is returned as the output of the function. The information returned must match the data type established in the header.

```
float find_min(float x1, float x2, float x3)
{
    float min;
...
    return min;
}
```

Understanding differences   Show Correct Answer   Show Responses

Use 'must' or 'may' to fill in the blanks:

The function declaration ________ exactly match the function definition header and the return value's data type ________ match the function's data type.

Invocation

Invoking a function is essentially just calling it like you would any built-in functions. A function can be declared in main or any other function. Calling a function within itself is called recursion and we will not cover or use that in this course.

For locally defined functions, you must provide as many inputs as are required by the function definition and match, in order, the data types of those input parameters. There are ways to set up a function to not need every input, but we won't get into that.

```
...
```

```
int main ()
{ float x, y, z, M;
   ...
   M = find_min (x, y, z);
   ...
}
float find_min(float x1, float x2, float x3)
{ ...
   return min;
}
```

If there is an output of the function, the variable used to capture that must match the output of the function. In this case, `find_min` is established as having a data type of `float` so the variable `M` had to be a `float`.

The example above uses all floats for simplicity. There is no limitation to the number of input parameters you can use or their types. You can have a mix of input types or different output than any of the inputs. There is no requirement to have any inputs. For example:

Declarations:	**Invocations:**
`int find_max(int, int, int);`	`int M; M=find_max(x, y, z);`
`char convert_case(char);`	`char C; C=convert_case('Z');`
`double slope(int, float);`	`double S1; S1=slope(7, -3.75);`
`void print_stars(int);`	`print_stars(10);`
`int read_first_data( );`	`int X; X=read_first_data( );`
`void print_prompt( );`	`print_prompt( );`
`bool flip_coin();`	`bool W; W=flip( );`

Invoked variables Show Correct Answer Show Responses

Which pairs are, in the example code, essentially the same variables?

A	x & x1
B	y & y2

C	x & x2
D	z & x3
E	y & z
F	None of the above

Calling Functions

Functions can be called in 2 ways, **Call by Value** or **Call by Reference**. This determines whether the function will be dealing with the value of whatever's passed to it or the actual location.

Call by Value

Call by Value is the method we typically use in functions. This means that the input parameters used in the function invocation don't themselves transfer to the function when it runs but rather pass their values onto the corresponding arguments in the function header. So using the example from Lesson 1:

- `...`
- `int main ()`
- `{ ...`
- `  M = find_min(x, y, z);`
- `  ...`
- `}`
- `float find_min(float x1, float x2, float x3)`
- `{ ...`
- `}`

When `find_min` is called, `x1` takes on the value of `x`, `x2` the value of `y`, and `x3` the value of `z`. This is the same as how you've previously used functions, both the built-in ones and those you created in MATLAB. The invoked variable passes along it's value to the function defined input variable. Anything done to `x1`, `x2`, or `x3` in the function can change those assigned values but that doesn't any effect on `x`, `y`, or `z`.

Call by Reference

In call by reference, you are passing the address of the variable being used as an input parameter. Let's start with this example:

```
#include<iostream>
using namespacestd;
voidtest(int &x);
int main()
{
   int y;
   test(y);
   cout << ”y = " << y << endl;
   return(0);
}
void  test(int &x)
{
   cout << “Enter value for x: “;
   cin >> x;
}
```

If you were to run this, your would get an output like:

```
Enter value for x: 42
y = 42
```

Note that the value of `y` is never defined in `main()`. The function, `test()`, is called with `y` as an input. In `test()`, the argument `x` is assigned a value and has no output. When back in `main()`, `y` is printed and is shown to have the same value you assigned to `x` in `test()`. This is because the value of `y` wasn't assigned to `x` but rather its location or address on the computer.

In the example, the function parameters are being declared and defined using an **&**. That is the **address-of** operator and, when used to define a function, means that pass by reference is to be used. When called by reference, the input argument and function parameter share the same location so anything done to that function parameter, like changing the value stored there, is reflected by the input argument.

Call by Reference Example:

```
#include <iostream>
using namespace std;

void test1(int a);
void test2(int &b);

int main() {
  int x = 5.0;
  cout << "Main:  x - Value: " << x <<" & address: " << &x << endl;
  test1(x);
  test2(x);
  return(0);
}

void test1(int a) {
  cout << "Test1: a - Value: " << a <<" & address: " << &a << endl;
}

void test2(int &b) {
  cout << "Test2: b - Value: " << b <<" & address: " << &b << endl;
}
```

The console output would look like:

```
Main:  x - Value: 5 & address: 0x7ffeeb35da48
Test1: a - Value: 5 & address: 0x7ffeeb35da1c
Test2: b - Value: 5 & address: 0x7ffeeb35da48
```

Note that the address for Main and Test1 differ. Test1 uses call by value. The address for Main and Test2 are the same, it's call by reference.

The basic structures look like this:

There is no inherent issue with using either. It typically depends on what's most valuable, what's happening in the functions or in main(). The more important one would be at the top and therefore most quickly accesible.

It may not look like it would matter, but if you consider a case with multiple functions, the important of placement in ease of access/editing becomes a little more apparent.

Call by Value or Reference Show Correct Answer Show Responses

What are the values of x, y, and z?

CALL BY VALUE: x = ______ , y = ______ , and z = ______

CALL BY REFERENCE: x = ______ , y = ______ , and z = ______

Arrays as Parameters

Sometimes you'll want to pass an array into a function. To do so, like having to declare a variable as being an array, you have to declare that the input parameters are arrays. Look at the following example:

```
#include <iostream>
using namespace std;
void ScaleGrades(int m[]);
int main()
{
  int marks[5] = {88, 76, 90, 61, 69};
  cout << "Current Grades " << endl;
  for (int i = 0; i < 5; ++i) {
```

```
9.       cout << "Student "<< i + 1 <<": ”<< marks[i] << endl;
10.    }
11.    ScaleGrades(marks);
12.    cout << "Scaled Grades " << endl;
13.    for (int i = 0; i < 5; ++i) {
14.      cout << "Student "<< i + 1 <<": ”<< marks[i] << endl;
15.    }
16.    return 0;
17.  }
18.  void ScaleGrades(int m[])
19.  {
20.  cout << "Displaying marks: "<< endl;
21.    for (int i = 0; i < 5; ++i) {
22.      m[i]=100-((100-m[i])/2);
23.    }
24.  }
```

When declaring the input parameters for the function, you need to include the type and, for arrays, the name with `[]` as well. You do not need to specify the size of the array. When using an array as an input argument, you just need the name of the array since you're passing the whole array.

Note that if you ran this, your output would not match up with the initial values of `marks[]`. Arrays, by default, get passed by reference. So the actions taken in the function `ScaleGrades()` on `m[]` get applied to the same location as `marks[]`, the array used in the function call. Run the above example and see how even though lines 8-10 and 13-15 are the same, they produce different results after `ScaleGrades()` has been called.

Multi-dimensional Arrays

To use a multi-dimensional array, you just need to set up the parameter as you would any multidimensional arrays. The only requirement is that the depth of those additional dimensions must be defined. For example using function declarations:

```
void matrixprint(int arr[][3], string);
```

For the non-array parameter, you don't need the name but it will still run if you include it. For the parameter that is a multidimensional array, you do need to define the name even in the declaration because the `[ ]`, which lets the compiler know that the function input is expected to be an array, is attached to the name.

Function Declaration with Arrays Show Correct Answer Show Responses

When making a function declaration, do you need to include the variable name for arrays?

A	True
B	False
C	Depends on circumstance

Part 1: Function-al Sort

For this activity, we'll be taking the program from P10L1 and converting elements in functions.

Part 1 Activity

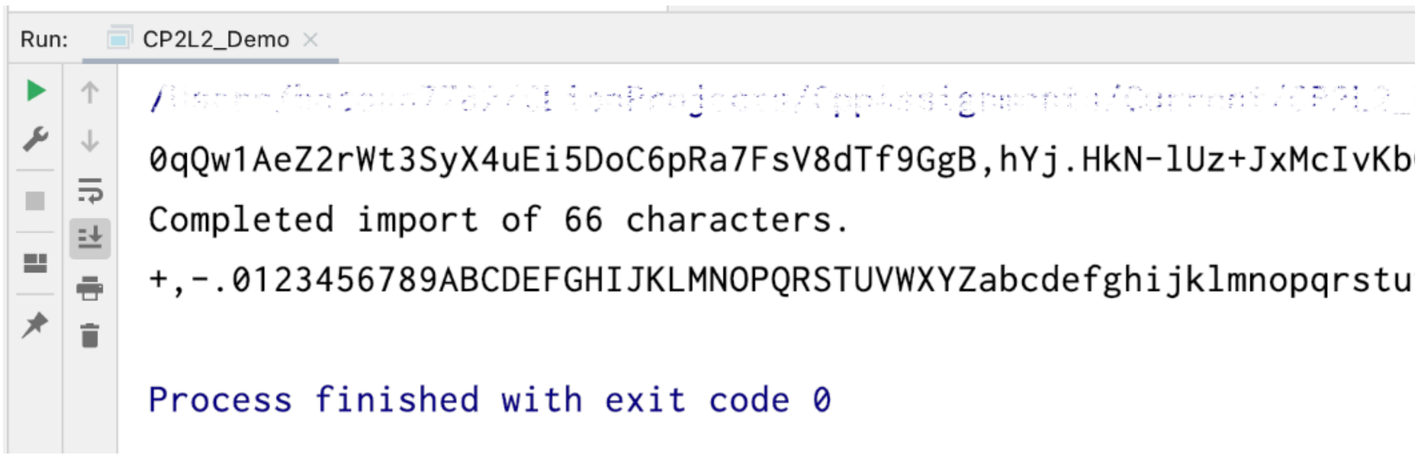
- Start with your code from P10L1 - Keystroke Sorter
- Move the Bubblesort to a function
- Move the swapping value to a function

Requirements:

- For your swap function
  - Use Call by Reference
- For your bubblesort function
  - Take the array and effective array length as an input.
- Use at least 50 characters in your test file
  - At least half are unique, meaning you can have some repeat characters but at least try and keep them mostly unique
- Do not copy the example from below

Example Output:

Using a test file made with 1 character from the majority of keys on an U.S. QWERTY keyboard, including capitalization for the letters.



```
/[redacted]
0qQw1AeZ2rWt3SyX4uEi5DoC6pRa7FsV8dTf9GgB,hYj.HkN-lUz+JxMcIvKb
Completed import of 66 characters.
+,-.0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstu

Process finished with exit code 0
```

Here is a functional version of P10L1:

https://raw.githubusercontent.com/boconn7782/CourseCode/refs/heads/main/CPP/P11L1_Base

You will need to create your own text file for importing though.

Pseudocode:

This is the expanded pseudocode from last week :

```
• PROGRAM KeystrokeSorter:
• Introduce Program to user
• Open test file as an input stream
• // Check the file stream
• IF file stream not valid
•   Provide user warning and end program with error
• ENDIF
• // Stream file content into an array
• FOR each element in array
•   load character from file stream into array element
•   Print original array data
•   Print element to console output
•   Keep count of number of characters
• ENDFOR
• // Sort array data using Bubblesort
```

```
SET n = Length of characters used in Array A
FOR j = 0 to n-1
  FOR k = 0 to (n-j-1)
    IF element is greater than the next
    // Swap their positions
      temp = element;
      element = next element;
      next element = temp;
    ENDIF
  ENDFOR
ENDFOR
// Print sorted array
FOR each element in sorted array
  Print element to console output
ENDFOR
ENDPROGRAM
```

Here it is broken into the required functions:

```
PROGRAM KeystrokeSorter:
Introduce Program to user;
Open test file as an input stream;
// Check the file stream
IF file stream not valid
  Provide user warning and end program with error;
ENDIF
// Stream file content into an array
FOR each element in array A
  load character from file stream into array A element;
  Print element to console output;
  Keep count of number of characters;
ENDFOR
// Sort array data using Bubblesort
CALL bubblesort WITH array A and count of number of characters;
// Print sorted array
FOR each element in sorted array
  Print element to console output;
ENDFOR
ENDPROGRAM

```

```
FUNCTION  Swap:
  PASS IN: E1, E2
  temp = E1;
  E1 = E2;
  E2 = temp;
ENDFN

FUNCTION bubblesort:
  PASS IN: list[], Count
  FOR j = 0 to Count-1
    FOR k = 0 to (Count-j-1)
      IF element of list is greater than the next element
        CALL swap WITH element of list and next element;
      ENDIF
    ENDFOR
  ENDFOR
ENDFN
```

Build Your Program

In this case, you can start with a copy of the previous program. Some of this will be repetitive because it's already established by the previous program but I'm just trying to establish good practices in developing your programs.

Set up the parts of your program

1. Libraries & Directives

We have requirements that will need certain libraries:

- Communicating with the console window
- Loading information from a file

Already setup

2. Declare Variables

We already know of some information we will need to store and use throughout:

- We'll need a large array
- A counter for counting the characters in it

Already setup

3. Main Program

We know what we want to do in our main program based on the pseudocode

4. Functions

We know we'll have 2 functions and we know what we want for inputs.

We'll have to make sure we declare each and define them as well, but we know what their inputs will be and since they're both pass by reference, there doesn't need to be an output.

Integrate into your code:

- `/* Title Block */`

Title Block and the introduction & instructions can typically wait till the end, when cleaning up for submission, but you'll have to update it from the previous assignment.

- `// Libraries & Directives`
- `#include... // Load the necessary libraries`
- `using namespace std; // To simplify code`

Based on what has been established as required, load the necessary libraries.

No changes have been made that would require changes from the original program.

Declare your functions

We need to include declarations for the functions. Remember the structure is:

`Data Type Function Name (Parameter 1's data type, Parameter 2's data type, ...)`

We will also be using an array as a parameter for one and pass by reference for the other.

- `// Function Declarations`
- `// Declare function swap using pass by reference for 2 characters`
- `// Declare function bubblesort for an array and an integer`

So we have 1 function with 2 character inputs. In it the values between the variables will be swapped. If that's just done in the function, it will have no effect anywhere else. We want to make sure that the initial inputs to the function have the values swapped so this will have to be pass by ...

Question

... so this will have to be pass by ...

A	Value
B	Reference

HINT

Show Responses Show Answer

We have another function with 2 inputs, 1 being a character array and the other being an integer. For the integer, we just need to allocate space for an integer. But for the array, we need to make sure enough space is set aside to handle an array of characters, not just a single character.

You should check the pseudocode for the function definitions to see what they return so you know the function type.

- `...`
-

- `FUNCTION  Swap:`
- `  PASS IN: E1, E2`
- `  ...`
- `ENDFN`
-
- `FUNCTION bubblesort:`
- `  PASS IN: list[], Count`
- `  ...`
- `ENDFN`

Extract the function code

You should already have some thing that matches the above pseudocode and we can assume it works. So the following is already handled in our code and doesn't have any changes to include functions so we can leave it be:

- `int main() {`
- `// Introduce Program to user`
- `// Open test file as an input stream`
- `// Check the file stream`
- `// IF file stream not valid`
- `//   Provide user warning and end program with error`
- `// ENDIF`
- `// Stream file content into an array`
- `// FOR each element in array`
- `//   load character from file stream into array element`
- `//   Print original array data`
- `//   Print element to console output`
- `//   Keep count of number of characters`
- `// ENDFOR`
- `...`

We want to skip to the following.

- `...`
- `// Sort array data using Bubblesort`
- `// SET n = Length of Array A`
- `// FOR j = 0 to n-1`
- `//  FOR k = 0 to (n-j-1)`

```
//    IF element is greater than the next
//     // Switch their positions
//      temp = element;
//      element = next element;
//      next element = temp;
//    ENDIF
//  ENDFOR
// ENDFOR
...
```

We want to take the **sorting task** and the **value swap** out of the code.

Create function Swap()

Let's start with **swap**. I'm just using pseudocode here but you can match that to the content of the base program provided.

```
...
// Sort array data using Bubblesort
// SET n = Length of Array A
// FOR j = 0 to n-1
//  FOR k = 0 to (n-j-1)
//    IF element is greater than the next
//     // Switch their positions
//      temp = element;
//      element = next element;
//      next element = temp;
//    ENDIF
//  ENDFOR
// ENDFOR
...
```

Based on the updates to the pseudocode, you want:

```
...
//    IF element is greater than the next
//     // Switch their positions
//      temp = element;
//      element = next element;
//      next element = temp;
//    ENDIF
```

```
...
```

To become:

```
...
//    IF element is greater than the next
//      CALL swap WITH element of list and next element;
//    ENDIF
...
```

Don't just delete that original code though and replace it with the invocation. You'll be cutting and pasting it in its new location soon.

We've already created the function declarations for this. Now you'll have to skip to the end and start defining the code below main():

```
int main() {
... }
```

Your function header should match you declaration, just with the variable names defined. It'll have no output as well. Building off the provided pseudocode, add to your code:

```
void swap(...) {
  //FUNCTION  Swap:
  //  PASS IN: E1, E2
  //  temp = E1;
  //  E1 = E2;
  //  E2 = temp;
}
```

You can take the old code from your main program and use it here. You just need to exchange your for the local arguments in the function, shown as `E1` and `E2` here.

So back up in `main()`, you can replace the old code, the bolded pseudocode above, for just the single function invocation `swap(element, next element)`, **exchanging element and next element for the appropriate array calls.**

```
// Sort array data using Bubblesort
// SET n = Length of Array A
// FOR j = 0 to n-1
//   FOR k = 0 to (n-j-1)
```

- `//     IF element is greater than the next`
- **`         swap(element, next element);`**
- `//     ENDIF`
- `//   ENDFOR`
- `// ENDFOR`

Create function bubblesort()

Now let's work on the bubblesorting function. After main(), you'll need to have created a function header. It should match the declaration but with both variable names defined, not just that for the array. It will also have no output. Arrays are passed by reference by default so there's no need for an address-of operator here and it will still provide new values for use in main() after it's call.

Building off the provided pseudocode, we'll add to your code:

- `void bubblesort(...) {`
- `//PASS IN: list[], Count`
- `//FOR j = 0 to Count-1`
- `//  FOR k = 0 to (Count-j-1)`
- `//    IF element of list is greater than the next element`
- `//      CALL swap WITH element of list and next element;`
- `//    ENDIF`
- `//  ENDFOR`
- `//ENDFOR`
- `}`

We'll point out the code that can be take from main() to replace the pseudocode here in a moment. You will have to update it to use the local arguments for the function, no longer what you used for the array and character count in `Main()`.

Back up in `main()`, you can grab the old code that handled the bubblesorting and replace it with a single function invocation `bubblesort(array, character count)`, exchanging array and character count for the variables you used for that information. Remember, you're using the whole array as an input, not just a single element. So in you want that original code:

- `...`
- `// Sort array data using Bubblesort`
- `// SET n = Length of Array A`
- `// FOR j = 0 to n-1`