

Introduction to Software Development – CS 6010

Lecture 5 – Functions

Master of Software Development (MSD) Program

Varun Shankar

Fall 2022

Misc

- Strings
 - `string myString; // What is the length of myString at this point?`
 - `myString[10] = 'H'; // Indexing the 11th location of an empty string is "undefined"`
 - `cout << myString; // Displays nothing...`
 - Nothing is displayed because `myString` did not have any space for putting in the new letters. The
 - `string myString(10, ' '); // Creates a string of length 10 filled with ' ' (blanks)`
 - `string myString(otherString.length(), ' ')`
- Loops
 - What is a loop within another loop called?
 - A *nested* loop.

Lecture 5 – Functions

- Topics
 - Functions
 - Lab – Functions
 - Homework – String Analyzer

You are the Computer. Can You Execute Code By Hand?

Ladder Diagram

- ➔ 1. `int total = 0;`
- 2. `for(int index = 1; index < 3; index++) {`
- 3. `total = total + index;`
- 4. `}`

You are the Computer

Ladder Diagram

total : 0

1. int total = 0;
- ➔ 2. for(int index = 1; index < 3; index++) {
3. total = total + index;
4. }

You are the Computer

Ladder Diagram

total : 0

index : 1

1. int total = 0;
2. for(int index = 1; index < 3; index++) {
- ➔ 3. total = total + index;
4. }

You are the Computer

Ladder Diagram

total : 0 1

index : 1

1. int total = 0;
2. for(int index = 1; index < 3; index++) {
3. total = total + index;
- ➔ 4. }

You are the Computer

What happens when this line executes?

Ladder Diagram

total : 0 1

index : 1

1. int total = 0;
- ➔ 2. for(int index = 1; index < 3; index++) {
3. total = total + index;
4. }

You are the Computer

What happens when this line executes?

1. Verify that index is < 3
2. Increment index

Ladder Diagram

total : 0 1

index : 1 2

```
1. int total = 0;  
2. for( int index = 1; index < 3; index++ ) {  
➔ 3.     total = total + index;  
4. }
```

You are the Computer

You should at this point be able to follow this code, and update the variable values correctly.

Ladder Diagram

total : 0 1 3

index : 1 2

1. int total = 0;
2. for(int index = 1; index < 3; index++) {
3. total = total + index;
- ➔ 4. }

What is a Function?

- Functions are:
 - Self-contained
 - Modules of Code
 - Take in Information (data values)
 - Perform a Computation (or computations)
 - Return the computed value (or values)
 - Have their own variables (scope)
- Also called 'procedures', 'methods' (usually with respect to an Object), etc.

To “Call” a function

- ‘Calling’ a function is the terminology we use to say that we are invoking (using) a function.

- Thus to call the `cos ()` function we write:

```
x = 5;
```

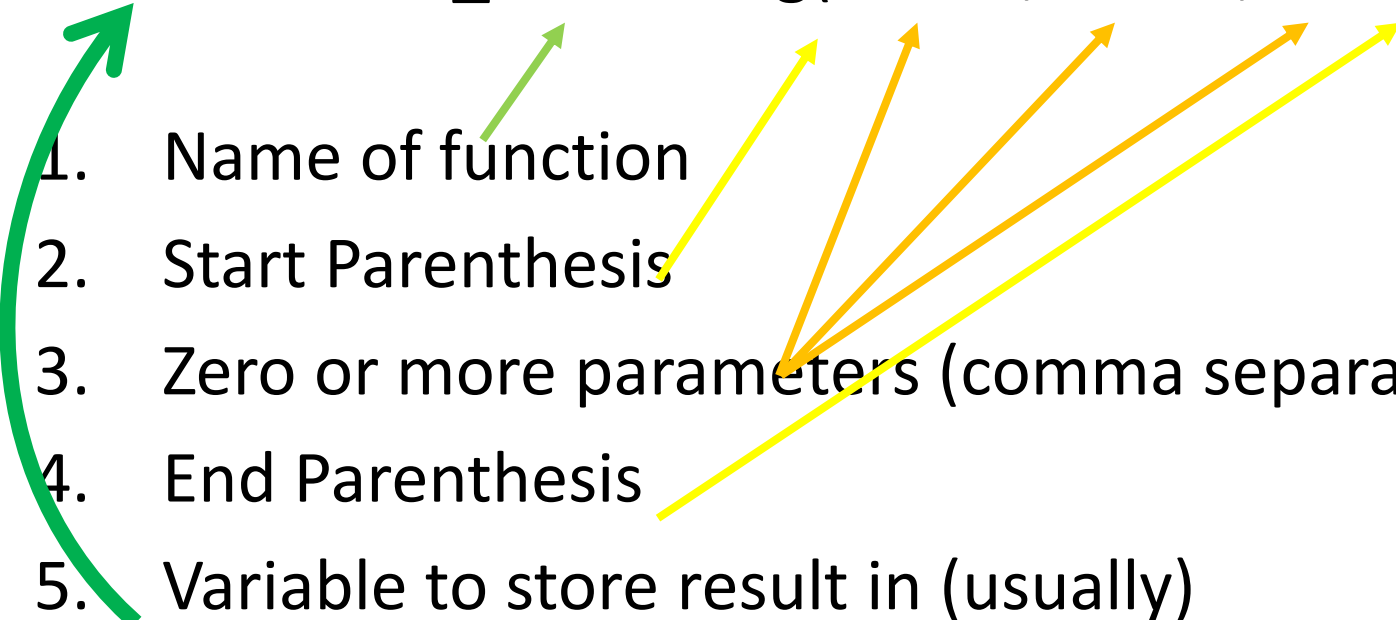
```
answer = cos ( x );
```

- Above, `x` is passed into the function as the parameter. The variable, *answer*, stores the output from the function, and `cos` is the name of the function being called.

Calling (invoking/using) a Function

To utilize a function, write code like this:

```
result = do_something( count, 'hello', false );
```

1. Name of function
 2. Start Parenthesis
 3. Zero or more parameters (comma separated variables or values)
 4. End Parenthesis
 5. Variable to store result in (usually)
- 

Examples of Calling Functions

- `test_value = pi;`
 - Not calling a function. This is assigning a variable!
- `sin(pi);`
 - Calculating the sine of `pi` but doing nothing with it!
 - Most of the time it does not do any good to call a function without assigning the result to a variable.
- `x = sin(test_value);`
 - Storing the value of the sine of `test_value` (which currently holds the value `pi`) into the variable `x`.
- `a = sin(cos(pi));`
 - First calling the `cos` function with the parameter (passed in value) of `pi`. Then passing the result of this into the `sin` function.

1st

2nd

Function Examples

- Functions ‘just do something’ for you:

```
result = sin( 3.1415 );  
shuffle_cards();  
result = is_odd( x );  
date = get_date();  
setPlotTitle( "Temperature" );
```

- Most take in inputs (parameters/variables), some don't.
- Most return a value (or values) and some don't.
- Functions are a basic building block of programs.

Functions You've Already Seen

- List some functions that we have already used:
- Some functions that you might use:
 - Math
 - cos, sin, tan, acos, ceil, floor, sqrt, pow

Functions You've Already Seen

- List some functions that we have already used:

- `main()`
 - `cout`
 - `printf()`
 - `myString.length()`
 - `.substring()`
 - `.find()`

- Some functions that you might use:

- Math

- `cos`, `sin`, `tan`, `acos`, `ceil`, `floor`, `sqrt`, `pow`

Why Functions

- Encapsulation and Abstraction
 - Encapsulation - Bundling required information and actions together
 - Abstraction
 - Hiding the details from the user of the object
 - Note: The original programmer does have to deal with ALL the complexity ONCE! After that the “application programmer” ignores it and just uses the function.
- Testing
 - Test one piece (function) of a program at a time.
- Re-use of code.
- “Self-Documenting Code”

Function Contracts, Preconditions, Postconditions

- Functions have **contracts** that define:
 - Inputs (Parameters)
 - Output(s)
 - Definition of purpose
- Unfortunately these contracts are not enforced by the programming language or computer, but must be done by the programmer.
- Note, the contract just says what the function will do – NOT how it will be done.

Functions Should Be Generic

- Bad function (too specific):

```
result = add_5_to_7();
```

- Ok, but:

```
add_two_numbers( 3, -7 ); // Where did the result (the returned value) go?
```

- Better:

```
result = add_two_numbers( 12, 44 );
```

Creating a Function in C++

```
<return-type> functionName( <param type> parameter1, <param type>  
parameter2, ... ) {
```

```
    // Lines of code that process data (the input parameters)
```

```
    return result; // Strive to have only one return in your function
```

```
}
```

- If a function does not return any value, its type written as *void*.
 - There's generally no reason to call *return* in a void function...
 - But if you do, it'll quit the function (without returning anything).

Variables and Functions

- Variables have three additional properties
 - Scope
 - Lifetime
 - Memory (Where they are physically stored in the computer's hardware.)
- Scope
 - Where the variable can be used. In the case of functions, the scope is **only in the function!**
- Lifetime (Related to Scope, but...)
 - When the variable can be used. In the case of a function, the data in the variable **goes away** when the function returns.

Scope (of a variable) – Functions and { }

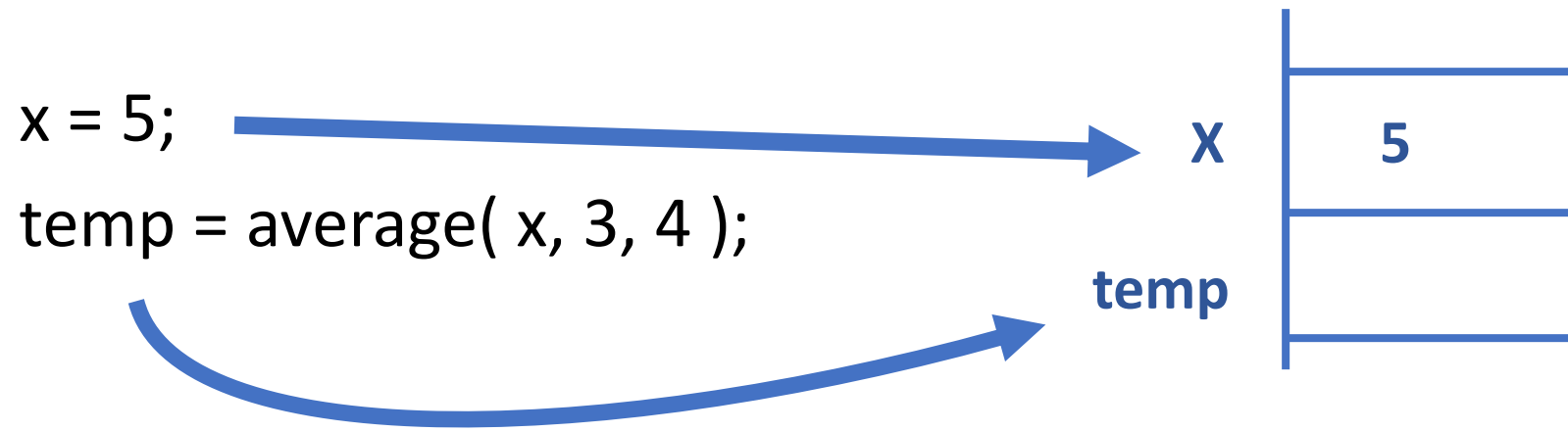
```
char firstLetter( string sentence ) {  
    char c = sentence.front();  
    // c is said to be local to this function  
    return c;  
}
```

```
int main() {  
    string name = "Varun";  
    char theLetter = firstLetter( name );  
    cout << c; // Does this work?  
    while() {  
        int x;  
    }  
    // x does not exist at this point!
```

```
        for( int x = 0; x < 10; x++ ) {  
        }  
        // x does NOT exist here.  
    } // end main()
```

Functions and the Ladder Diagram

(Ladder diagram represents the current call stack.)



Functions and the Ladder Diagram

(Ladder diagram represents the current call stack.)

```
float average( 5float n1, 3float n2, 4float n3 ) {
```

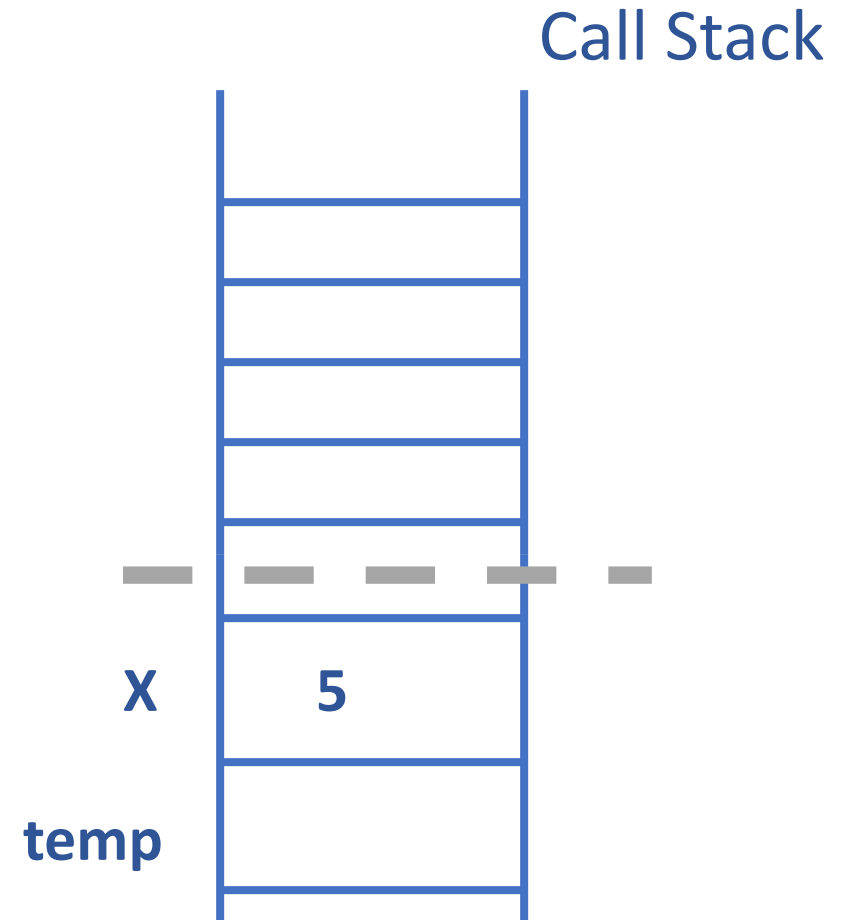
```
float avg = n1 + n2 + n3;  
avg = avg / 3;
```

```
return avg;
```

```
} // end function
```

```
x = 5;
```

```
temp = average( x, 3, 4 );
```



Functions and the Ladder Diagram

(Ladder diagram represents the current call stack.)

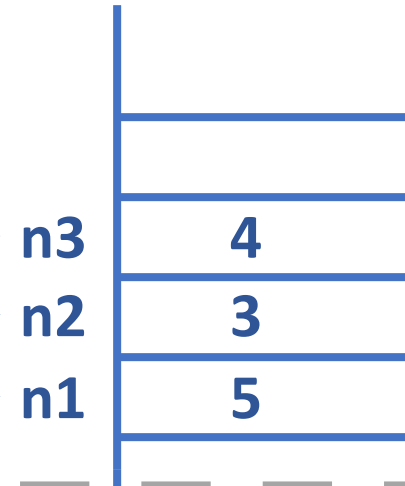
```
float average( float 5n1, float 3n2, float 4n3 ) {
```

```
float avg = n1 + n2 + n3;  
avg = avg / 3;
```

```
return avg;
```

```
} // end function
```

Call Stack



For all intents and purposes, the memory (*frame*) of the main function ceases to exist. (The memory is actually still there, but we cannot use or see it.)

Functions and the Ladder Diagram

(Ladder diagram represents the current call stack.)

```
float average( float n1, float n2, float n3 ) {
```

```
    float avg = n1 + n2 + n3;
```

```
    avg = avg / 3;
```

```
    return avg;
```

```
} // end function
```

Call Stack



For all intents and purposes, the memory (*frame*) of the main function ceases to exist. (The memory is actually still there, but we cannot use or see it.)

Functions and the Ladder Diagram

(Ladder diagram represents the current call stack.)

```
float average( float n1, float n2, float n3 ) {
```

```
    float avg = n1 + n2 + n3;
```

```
    avg = avg / 3;
```

```
    return avg;
```

```
} // end function
```

Call Stack



For all intents and purposes, the memory (*frame*) of the main function ceases to exist. (The memory is actually still there, but we cannot use or see it.)

Functions and the Ladder Diagram

(Ladder diagram represents the current call stack.)

```
float average( float n1, float n2, float n3 ) {
```

```
    float avg = n1 + n2 + n3;
```

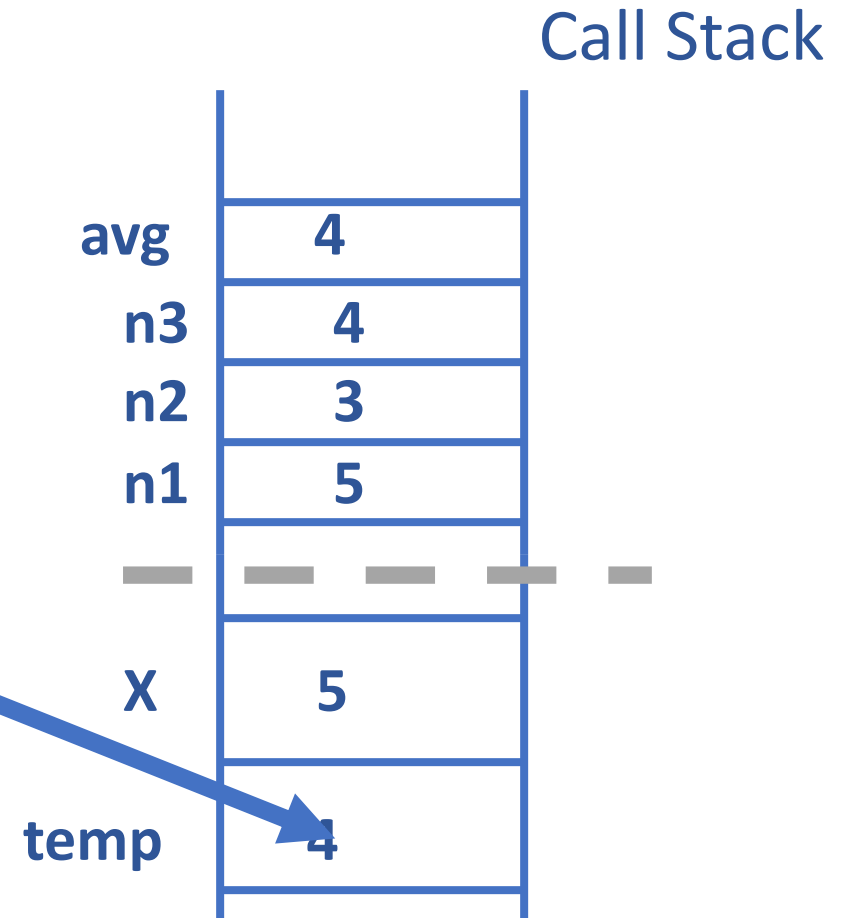
```
    avg = avg / 3;
```

```
    return avg;
```

```
} // end function
```

```
    x = 5;
```

```
    temp = average( x, 3, 4 );
```



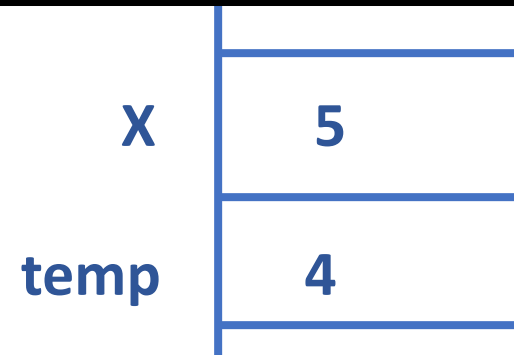
Functions and the Ladder Diagram

(Ladder diagram represents the current call stack.)

At this point, the function has *returned*, and all knowledge of it (variables) cease to exist.

`x = 5;`

`temp = average(x, 3, 4);`



Call Stack

Return Statement in Functions

- Sometimes it is necessary to end the execution of a function early.
 - Error conditions
 - Simple base case answers
- The **return** statement causes the execution of the function to end immediately, and to *return* from the current function to the caller of the function.
- Any code in the function that is after the *return* will not be executed.

Let's Write Some Functions

- Calculate the area of a rectangle
- Repeatedly display the given character some number of times
- Roll a dice X times and return the largest roll
- A function that determines if the input* number is odd
 - What does *input* mean when talking about a function?
 - Now a function to determine if the parameter is odd.

Today's Assignment(s)

- Lab – Functions
- Homework – String Analyzer