Introductory Robot Programming

ENPM809Y

Lecture 04 – Functions

Zeid Kootbally
zeidk@umd.edu

Fall 2020

Overview I

Overview II

Functions

- Function: A group of statements that together perform a particular job.
- Every C++ program must have a function named `main` (which is where the program starts execution when it is run). However, as programs start to get longer and longer, putting all the code inside the `main` function becomes increasingly hard to manage.
- Functions provide a way for us to split our programs into small, modular chunks that are easier to organize, test, and use.

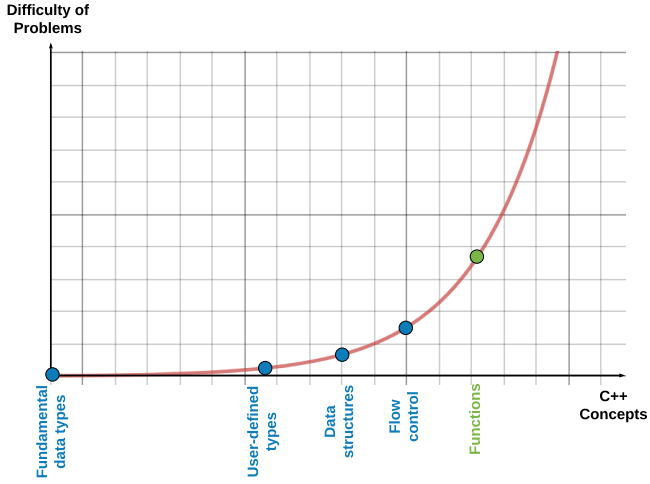
```
int main(){
    //--read input
    statement_1;
    statement_2;
    //--process input
    statement_3;
    statement_4;
    //--provide result
    statement_5;
    statement_6;
}
```

```
int main(){
    //--read input
    ReadInput();
    //--process input
    ProcessInput();
    //--provide result
    ProvideResult();
}
```

Functions | Why Use Functions?

- Organization – As programs grow in complexity, having all the code live inside the `main` function becomes increasingly complicated.
- Reusability – Once a function is written, it can be called multiple times from within the program. This allows you to avoid duplicated code (Don't Repeat Yourself). You can also share functions with other programs.
- Testing – Less code to test (because of less code redundancy). Working functions do not need to be tested again unless they are changed.
- Extensibility – When we need to extend our program to handle a case it didnt handle before, functions allow us to make the change in one place and have that change take effect every time the function is called.
- Abstraction – No need to know how functions work. This lowers the amount of knowledge required to use other peoples code (including everything in the Standard Library).

Functions


Difficulty of Problems
Fundamental data types
User-defined types
Data structures
Flow control
Functions
C++ Concepts

Functions | Standard Library

- You will mainly work with functions from the Standard Library.

```cpp
#include <cmath>

int main(){
    std::cout << sqrt(400) << std::endl;
    double result = pow(3.0,4.0);
    std::cout << result << std::endl;
}
```

Functions | Third-party Libraries

- You may also use functions from third-party libraries (e.g., Eigen).

```
#include <iostream>
#include "Eigen/Core"

int main() {
    int rows=5, cols=5;
    Eigen::MatrixXf m(rows,cols);
    m << (Eigen::Matrix3f() << 1, 2, 3, 4, 5, 6, 7, 8, 9).finished(),
            Eigen::MatrixXf::Zero(3,cols-3),
            Eigen::MatrixXf::Zero(rows-3,3),
            Eigen::MatrixXf::Identity(rows-3,cols-3);
    std::cout << m;
}
```

Functions | User-defined Functions

- You will very likely write your own (user-defined) functions.

```
/**
* @brief Adds two integer numbers and returns the result.
* @param a an integer argument.
* @param b an integer argument.
* @return The result of \p a + \p b.
*/
int AddNumbers(int a, int b){
    std::cout << a << " + " << b << " = "<< a+b << std::endl;
    return a + b;
}

int main(){
    int x = AddNumbers(2,3);
    int y = AddNumbers(4,6);
    AddNumbers(x,y);
}
```

Functions | Syntax

```
ReturnType FunctionName([Type par_name_1, ..., Type par_name_n]){
    Body
}
```

- `ReturnType`: Type of the value that the function returns. This can be a fundamental data type or a user-defined type. Some functions do not return any value, in this case, the `ReturnType` is the keyword `void`.
- `FunctionName`: Name of the function (see Google C++ style guide).
- `[Type par_name_1, ..., Type par_name_n]`: Optional list of formal parameters. `Type` can be a fundamental data type or a user-defined type.
- `Body`: Instructions to be executed when the function is invoked, enclosed in curly braces `{}`
- Header: `ReturnType FunctionName([Type par_name_1, ..., Type par_name_n])`
- Signature: `FunctionName([Type par_name_1, ..., Type par_name_n])`

Functions | Examples

```
//--No parameters
int FunctionName(){
    int i{1};
    return i;
}
```

```
//--One parameter
int FunctionName(int i){
    return i+1;
}
```

```
//--No return type
void FunctionName(){
}
```

```
//--Multiple parameters and no return type
void FunctionName(int i, int j){
    std::cout << i + j << std::endl;
}
```

```
//--Multiple parameters
int FunctionName(int i, int j){
    return i+j;
}
```

```
//--No return type and no parameters
void FunctionName(){
    std::cout << "Hello" << std::endl;
}
```

Functions | Calling Functions

```
1  void SayHello(){
2      std::cout << "Hello" << std::endl;
3  }
4
5  int main(){
6      SayHello();
7      std::cout << "End of main()" << std::endl;
8  }
```

1. `main()` is called automatically (entry point) and the code in any other function is only executed if the function is called from `main()` (directly or indirectly).
2. Control is passed to `SayHello()`.
3. Body of `SayHello()` is executed.
4. Control is returned to the caller (`main()`) when the closing curly bracket in `SayHello()` is reached (line 3).
5. Remaining code of `main()` is executed.
6. Program exits when the closing curly bracket in `main()` is reached (line 8).

Functions | Calling Functions

```
void SayWorld(){
    std::cout << " World" << std::endl;
}

void SayHello(){
    std::cout << "Hello";
    SayWorld();
}

int main(){
    SayHello();
    std::cout << "End of main()" << std::endl;
}
```

1. `main()` is called automatically (entry point) and the code in any other function is only executed if the function is called from `main()` (directly or indirectly).
2. `SayHello()` is called directly by `main()`.
3. `SayWorld()` is called directly by `SayHello()` and indirectly by `main()`.

Functions | Calling Functions

- Question: What does the following program output?

```
void SayWorld(){
    std::cout << " World" << std::endl;
    std::cout << "End of SayWorld()" << std::endl;
}

void SayHello(){
    std::cout << "Hello";
    SayWorld();
    std::cout << "End of SayHello()" << std::endl;
}

int main(){
    SayHello();
    std::cout << "End of main()" << std::endl;
}
```

Functions | Calling Functions

- Functions can call other functions, however the compiler must know the function details before they are called.
- The compiler parses your code (like a book) first to make sure everything is OK. Then it will call the `main()` function.

```
int main(){
    SayHello();
    std::cout << "End of main()" << std::endl;
}

void SayHello(){
    std::cout << "Hello";
}
```

```
error:'SayHello' was not declared in this scope
```

- Question: How can you fix this?

Functions | Calling Functions | Cyclic References

```
int GetInput(){
    int input{};
    std::cout << "Enter an int: ";
    std::cin >> input;
    if(input > 5)
        return input;
    else
        return CallGetInput();//--function call
}
int CallGetInput(){
    return GetInput();//--function call
}
int main(){
    std::cout << CallGetInput() << std::endl;//--function call
}
```

- Question: Where is the issue?
- Question: How do I fix this?

Functions | Forward Declaration and Function Prototype

- Forward declaration:
  - Allows the compiler to do a better job at validating your code.
  - The compiler wants to ensure you did not make spelling mistakes or passed the wrong number of arguments to the function calls. So, it insists that it first sees a declaration of a function (or any other types) before it is used.
  - Break cyclic references where two definitions both use each other.
- Function prototype:
  - A declaration of a function without its body to give compiler information about user-defined function.
  - Placed at the beginning of the program or in your own header files (more on this later in this course).
  - You need to provide both the function prototype and function definition to reduce risks of errors.

Functions | Forward Declaration and Function Prototype

```
ReturnType FunctionName([Type [par_name_1], ..., Type [par_name_n]]);
```

- A function prototype requires at least `ReturnType`, `FunctionName`, and `Type` of parameters if parameters are required.
- The parameter identifiers are optional:

```
double MultiplyNumbers(int, double);
```

or

```
double MultiplyNumbers(int x, double y);
```

- Using an identifier for each parameter in the prototype is useful if you document (Doxygen) the function prototype instead of the function definition.

Functions | Forward Declaration and Function Prototype

```
/**
* @brief Multiplies two numbers and returns the result.
* @param a First number: int
* @param b Second number: double
* @return \p a \f\times\f \p b.
*/
double MultiplyNumbers(int a, double b);

double MultiplyNumbers(int a, double b){
    return a * b;
}
```

- Choose between documenting function prototypes or function definitions but be consistent with your choice.
  - Do not document function prototypes sometimes and function definitions other times.

Functions | Forward Declaration and Function Prototype

```
//--Function prototype
double MultiplyNumbers(int a, double b);//--Using a and b

//--Function definition
double MultiplyNumbers(int x, double y){//--Using x and y
    return x * y;
}
```

- Question: Will the program above throw an error since we are using different parameter identifiers?
- Answer: No, only `ReturnType` (`double`), `FunctionName` (`MultiplyNumbers`), and number/order/`Type` of the parameters (`int` and `double`) must match between the function prototype and the function definition.
- *Best practice*: Use the same parameter identifier(s) in both the function prototype and function definition if you plan to use parameter identifiers in the function prototypes.

Functions | Disclaimer

- In some of the slides that follow, function prototypes are omitted (on purpose) in the code, mainly to fit the code in the slides.
- However, you are required to include function prototypes whenever you define functions in your assignments and projects.

Functions | `return` Statement

- If a function returns a value then you have to explicitly use the `return` statement.
- Omitting the `return` statement in a non-void function (except `main()`) and using the returned value in your code invokes Undefined Behaviour.
- A `return` statement with an expression can be used only in functions returning a value; the value of the expression is returned to the caller of the function.

Wrong function definition:

```
//--function definition
int AddNumbers(int num1, int num2){
    int result {num1 + num2};
    //--should return an int
}

int main(){
    AddNumbers(1,2);
}
```

Correct function definition:

```
//--function definition
int AddNumbers(int num1, int num2){
    int result {num1 + num2};
    return result;
}

int main(){
    AddNumbers(1,2);
}
```

Functions | `return` Statement

- If a function does not return a value (`void`), then the `return` statement is optional.
- When a `return` statement is executed, the function is terminated immediately at that point, regardless its location in the program, and control is given back to the caller.
- `return` statement can occur anywhere in the body of the function.
- Many `return` statements can be in the body of a function.

```
void PrintLargerNumber(int num1, int num2){
    if (num1 > num2)
        std::cout << num1 << std::endl;
    else if (num1 < num2)
        std::cout << num2 << std::endl;
    else std::cout << "Equal" << std::endl;
    return;//--Optional
}
```

Functions | Formal Parameters

- A function parameter (sometimes called a formal parameter) is always a local variable to the function definition.
- Even though parameters are not declared inside the function block, function parameters have local scope.
- This means that they are created when the function is invoked, and are destroyed when the function block terminates.

```
//--function prototype
int AddNumbers(int num1, int num2);//--num1 and num2 are parameters
//--function definition
int AddNumbers(int num1, int num2){//--num1 and num2 are parameters
    int result {num1 + num2};
    return result;
}
```

Functions | Actual Parameters

- An argument (sometimes called an actual parameter) is the value that is passed to the function by the caller.

```
int main(){
    //--3: Argument passed to the parameter num1
    //--9: Argument passed to the parameter num2
    AddNumbers(3,9);
}
```

- When `AddNumbers(3,9)` is called, `AddNumbers`'s parameter `num1` is created and assigned the value of 3, and `AddNumbers`'s parameter `num2` is created and assigned the value of 9.

Functions | The main Function

- From the C++ standard, the `main` function has two signatures `main()` and `main(int argc, char** argv)`.
- `argv` and `argc` are how command line arguments are passed to `main()` in C and C++
- `argc` (argument count) is the number of arguments being passed into your program from the command line and `argv` (argument vector) is the array of arguments.
- Note: `argc` and `argv` are identifiers used by convention but they can be given any valid identifier: `int main(int num_args, char** arg_strings)` is equally valid.

```
int main(int argc, char** argv) {
    std::cout << "Have " << argc << " arguments:" << std::endl;
    for (int i = 0; i < argc; ++i) {
        std::cout << argv[i] << std::endl;
    }
}
```

Functions | Passing Arguments

- There are 3 primary methods of passing arguments to functions:
  - Pass by value.
    - By default, data passed into a function is passed by value.
    - A copy (different memory location) of the data is passed to the function.
    - Changes made to data passed into the function do not affect the argument(s) (since we are working with a copy).
  - Pass by reference.
    - A reference variable is a nickname, or alias, for some other variable. To declare a reference variable, we use & (unary operator).
    - Changing the value of the reference parameter changes the value of the referred parameter.
  - Pass by pointer will be covered in next lecture.

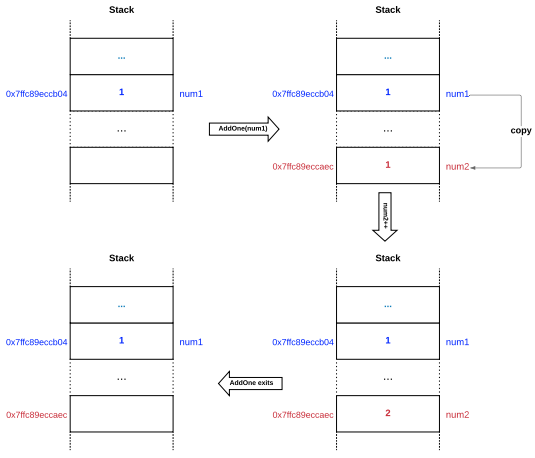
Functions | Passing Arguments | Pass by Value

```
void AddOne(int number);

int main(){
    int num1{1};
    std::cout << num1 << std::endl;//--1
    //--A copy of num1 is made
    //--   i. Memory is allocated for an int
    //--  ii. The value of num1 is stored there
    //--The copy of num1 is passed to AddOne
    AddOne(num1);
    std::cout << num1 << std::endl;//--1
}

void AddOne(int num2){
    //--1. A local variable named num2 is created
    //--2. Value of num1 (1) is copied into num2
    num2++;//--3. number is incremented
}//--4. num2 is destroyed
```

Functions | Passing Arguments | Pass by Value



Functions | Passing Arguments | Pass by Value

- Advantages:
  - Arguments passed by value can be variables (e.g., `x`), literals (e.g., `6`), expressions (e.g., `x+1`), `struct` variables, `class` objects, and enumerators.
  - Arguments are never changed by the function being called, which prevents side effects: Pass by value is flexible, safe, and efficient (if passing fundamental data types).
- Disadvantages:
  - Copying `struct` variables or `class` objects (`std::array`, `std::vector`, `std::string`, etc) or user-defined objects can incur a significant performance penalty, especially if the function is called many times.
- *Best practice*:
  - Use pass by value when passing fundamental data types and enumerators, and the function does not need to change the argument.
  - Do not use pass by value when passing `struct` variables or `class` objects.

Functions | Passing Arguments | Pass by Reference

- While pass by value is suitable in many cases, it has some limitations.
  1. When passing large data structures or class objects, pass by value will make a copy of the argument into the function parameter. In many cases, this is a needless performance hit, as the original argument would have sufficed.
  2. When passing arguments by value, the only way to return a value back to the caller is via the functions return value. While this is often suitable, there are cases where it would be more clear and efficient to have the function modify the argument passed in.
- Pass by reference means to use reference parameters to tell the compiler to pass in a reference to the actual parameter. The formal parameter will now be an alias for the actual parameter.
- The called function can modify the value of the argument by using its reference passed in.
- This is different from pass by value where the value of the argument is locally modified (only inside the function).

Functions | Passing Arguments | Pass by Reference

- To pass a variable by reference, we simply declare the function parameters as references (with &) rather than as normal variables. You need to modify the prototype as well.

```
void AddOne(int&);//--prototype
void AddOne(int& num2){
    num2++;
}
```

- When `AddOne` is called, `num2` will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

```
int main(){
    int num1{1};
    std::cout << num1 << std::endl;
    //--num2 becomes a reference to num1
    AddOne(num1);
    std::cout << num1 << std::endl;
}
```

- We did not change anything in the `main` function.

Functions | Passing Arguments | Summary

- Use pass by value when:
  - Passing fundamental data types and enumerators.
  - The function does not need to change the argument.
- Use pass by reference when:
  - Working with data structures from the Standard Library (`std::vector`, `std::array`, `std::string`, `std::stack`, etc) or user-defined types such as `struct` and `class` objects.
    - To make them read-only, pass them as `const` reference:
      `void DisplayPerson(const Person& person)`
  - You want to change multiple variables in the function:
    `void SwapNumbers(int& a, int& b)`

Functions | Default Arguments

- A default argument is a default value provided for a function parameter.
- If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used.
- If the user does supply an argument for the parameter, the user-supplied argument is used.
- Because the user can choose whether to supply a specific argument value, or use the default, a parameter with a default value provided is often called an optional parameter.

```
void AddNumbers(int a,int b=4,int c=5){}//--b and c are optional parameters
        std::cout << a + b + c << std::endl;
}
int main(){
    AddNumbers(1);  //-- a=1 + b=4 + c=5
    AddNumbers(1,2);//-- a=1 + b=2 + c=5
    AddNumbers();   //--error: too few arguments
}
```

Functions | Default Arguments

- Once defined, a default argument can not be re-defined. The default argument can be defined in either the prototype or the function definition, but not both.

```
void AddNumbers(int,int=4,int=5);//--Default arguments
void AddNumbers(int a,int b=4,int c=5){//--b=4 and c=5 re-defined
        std::cout << a + b + c << std::endl;
}
```

- All default arguments must be for the rightmost parameters. The following is not allowed:

```
void AddNumbers(int=1,int,int=5);//--Rightmost parameters only
```

- If more than one default argument exists, the leftmost default argument should be the one most likely to be explicitly set by the user.

Functions | Function Overloading

- Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters. The compiler must be able to tell the functions apart based on the parameter lists and the arguments supplied.

```
int Multiply(int i, int j) {return i*j;}
double Multiply(double i, double j) {return i*j;}
double Multiply(double i, double j, double k) {return i*j*k;}

int main(){
    Multiply(1,2); //--Multiply(int,int)
    Multiply(1.5,2.5);//--Multiply(double,double)
    Multiply(1.5,2.5, 3.5);//--Multiply(double,double, double)
}
```

Functions | Function Overloading

- Note: The `return` type of the function is not considered for uniqueness in function overloading,
- The compiler does not know which one you want to use. The second function will be treated as a re-definition of the first function.

```
int Multiply(int i, int j) {return i*j;}
double Multiply(int i, int j) {return i*j;}
```

Note: Functions with default arguments may be overloaded, however, it is important to note that optional parameters do NOT count towards the parameters that make the function unique.

```
double Multiply(double i, double j=3) {return i*j;}
int Multiply(int i) {return i*2;}
int Multiply(int i, int j=3) {return i*j;}

int main(){
    Multiply();//--Multiply(double, double)
    int Multiply(10);//--Ambiguous
}
```

Functions | Function Overloading | Match

- Making a call to an overloaded function results in one of three possible outcomes:
  1. A match is found. The call is resolved to a particular overloaded function.
  2. No match is found. The arguments can not be matched to any overloaded function.
  3. An ambiguous match is found. The arguments matched more than one overloaded function.

Functions | Function Overloading | Match

1. A match is found. The call is resolved to a particular overloaded function.

```
int Multiply(int i, int j) {return i*j;}
double Multiply(double i, double j) {return i*j;}

int main(){
    int i{2}, j{3};
    Multiply(i,j); //--Multiply(int,int)
}
```

Functions | Function Overloading | Match

2. If no exact match is found, the compiler tries to find a match through promotion.
   - `char`, `unsigned char`, and `short` can be promoted to an `int`.
   - `unsigned short` can be promoted to `int` or `unsigned int`, depending on the size of an `int`.
   - `float` is promoted to `double`.
   - `enum` is promoted to `int`.

```
int Multiply(int i, int j) {return i*j;}

int main(){
    short i{2}, j{3};
    //--No match for Multiply(short, short)
    //-- i and j are promoted to int
    Multiply(i,j);
}
```

Functions | Function Overloading | Match

3. If no promotion is possible, C++ tries to find a match through standard conversion.
   - Any numeric type will match any other numeric type, including `unsigned` (e.g., `int` to `double`).
   - `enum` will match the formal type of a numeric type (e.g. `enum` to `float`).
   - pointer conversions...

```
int Multiply(int i, int j) {return i*j;}

int main(){
    double i{2.6}, j{3.3};
    //--No match for Multiply(double, double)
    //-- i and j are converted to int
    Multiply(i,j);
}
```

Functions | Function Overloading | Exercise #1

- Create a program which computes the area of two shapes, a rectangle and a circle, using function overloading for `ComputeArea`.
  - Rectangle: `ComputeArea` takes two parameters (the length l and the width w), both of type `int`, and returns the area as an `int` type.
  - Circle: `ComputeArea` takes one parameter (the radius r), of type `double`, and returns the area as a `double` type.
- Write the prototypes, the function definitions, and the `main()` function.
- area_rectangle = $l \times w$
- area_circle = $\pi \times r^2$

Functions | `static` Variables

- Function local variables are only active while the function is executing.
- Local variables are not preserved between function calls.

```
void StaticLocalExample(){
    int num {10};
    std::cout << "num is: " << num << std::endl;
    num ++;
}

int main(){
    StaticLocalExample();//--num is 10
    StaticLocalExample();//--num is 10
}
```

Functions | `static` Variables

- To preserve the value of function local variables, you can declare variables to be `static`.
- All `static` variables persist until program terminates.

```
static int num {10};
```

  - The variable is only initialized the first time the function is called.
  - Space for the `static` variable is allocated only once and the value of variable in the previous call gets carried through the next function call.
  - Very useful when previous states of the function need to be stored.

Functions | `static` Variables

```
void StaticLocalExample(){
    static int num {10};
    std::cout << "num is: " << num << std::endl;
    num ++;
}

int main(){
    StaticLocalExample();//--num is 10
    StaticLocalExample();//--num is 11
}
```

Functions | How do Function Calls Work?

- In general, when a normal function call instruction is encountered, the program:
  - stores the memory address of the instructions immediately following the function call statement,
  - loads the function being called into the memory,
  - copies argument values,
  - jumps to the memory location of the called function,
  - jumps back to the address of the instruction that was saved just before executing the called function.
- Compilers have calling conventions.

Functions | How do Function Calls Work?

Functions | How do Function Calls Work?

- Functions use the "call stack" in memory.
  - LIFO method - Last In First Out (Push and Pop).
  - Analogous to a stack of books.
- Stack Frame or Activation Record.
  - A function must return control to function that called it.
  - Each time a function is called, a new activation record (or stack frame) is created and pushed on the stack.
  - When a function terminates, the activation record for this function is popped off the call stack.
  - Now the top of the stack is the function that called the one that was just popped off.
- Call stack finite in size – stack overflow.

```cpp
#include <iostream>

void func2(int &x, int y, int z) {
    x+= y + z;
}

int func1(int a, int b) {
    int result{};
    result = a + b;
    func2(result, a , b);
    return result;
}

int main() {
    int x{10};
    int y{20};
    int z{};
    z = func1(x,y);
    std::cout << z << std::endl;
}
```

Functions | Recursive Functions

- A recursive function is a function that calls itself until a "base condition" is true, and execution stops.
- While false, we will keep pushing activation records on top of the stack.
- In general, a recursive function has at least two parts:
  1. A base condition.
  2. At least one recursive case.

Functions | Recursive Functions

- Many times, a problem broken down into smaller parts is more efficient.
- Dividing a problem into smaller parts aids in conquering it. Hence, recursion is a divide-and-conquer approach to solving problems.
  - Sub-problems are easier to solve than the original problem
  - Solutions to sub-problems are combined to solve the original problem
- Many problems that can be solved with recursive approaches
  - Math: Fibonacci, fractals, factorial, ...
  - Searching and sorting.

Functions | Recursive Functions

- Factorial function: $\text{factorial}(n) = n \times \text{factorial}(n-1)$
  - Base case: $\text{factorial}(0) = 1$
  - Recursive case: $\text{factorial}(n) = n \times \text{factorial}(n-1)$

```
unsigned long long Factorial(unsigned long long n){
    if (n == 0)
        return 1;
    return n*Factorial(n-1);
}
```

Functions | Recursive Functions

- Fibonacci function: $\text{fibo}(n) = \text{fibo}(n-1) + \text{fibo}(n-2)$
  - Base case: $\text{fibo}(0) = 0, \text{fibo}(1) = 1$
  - Recursive case: $\text{fibo}(n) = \text{fibo}(n-1) + \text{fibo}(n-2)$

```
unsigned long long Fibo(unsigned long long n){
    if (n <=1)
        return n;
    return Fibo(n-1) + Fibo(n-2);
}
```

Functions | Recursive Functions

- If recursion does not eventually stop, you will have infinite recursion.
- Recursion can be resource intensive.
- Remember the base case, it terminates the recursion.
- Only use recursive solutions when it makes sense.

Next Class | 10/01

- Reading Material 02 on Flow Control.
- Lecture05: Pointers.
- Quiz on Lectures 04.
- Assignment due.
- Stay safe!