EECE 1080C
Programming for ECE
Lab 5

Sina Eghbal, Petar Acimovic
(eghbalsa@mail.uc.edu, acimovpr@mail.uc.edu)

February 13, 2023

Week 6

1 Objective

The objective of this lab is to make sure we have a good understanding of operators in C++, write a few simple functions, use them in our code, and devise and implement a few simple algorithms as functions in C++.

2 A Few Notes

Functions are sequences of instructions "packaged" as a single unit that can be **called** from anywhere within the code. Functions can modify the **state** of your program, and **usually** return a value. The return type *void* implies that a function cannot return anything. The **return** keyword can however come in handy when writing functions with return type *void*. In those situations, *return* may be used in order to **terminate the execution** of the function. C++ functions are by default **call-by-value** (aka. pass-by-value). That is, arguments passed to a function will be **copied**, so modifying them will not modify the values of the original variables passed to the function (literals [eg. 1, 13, 'a', "hello"] and expressions [eg. a - 3] can be passed to functions **as values**, but they **cannot** be passed **as references** since they are literals and are not stored in the memory outside the scope of the function call). While implementing a function, the programmer can choose to, take some or all of a function's arguments as references (call-by-reference (aka. pass-by-reference)) by putting an & after the type of the arguments it chooses to be passed as references.

Function definitions need to be placed before any attempt to call them. For instance, the following code will not compile since *fn(...)* is defined below main:

```
int main() {
        cout << fn(3);
        return 0;
}

int fn(int i) {
        return i;
}
```

To fix the compilation error, you can simply move the definition of *fn(...)* above *main(...)*.
But we have stated that functions can be called inside each other (just like we called *fn(...)*) from *main(...)*. So what if we have two functions - each calling the other one?

```
int fn1(int i) {
        if (i % 2 == 0)
                return fn2(i);
        return i;
}
int fn2(int i) {
        if (i % 2 == 1)
                return fn1(i)
        return i;
}
```

In cases like this, there is no right order of your function definitions for your code to compile. Therefore, the issue cannot be resolved, at least given by our current current knowledge, and definitely not by changing the order in which the two functions are defined. But that cannot be, and in fact is not the case. C++ compilers **do not** need the **definition** on your functions to be placed before the functions that are calling it. It suffices to place the **declaration** of your functions before the definition of those functions that are using it. A function's declaration consists of its signature followed by a *;*.
For instance, the declaration of *fn(...)* in the first program is:

```
int fn(int i);
```

So instead of moving *fn* to the top, we can fix the code by putting *fn(...)*'s declararion before main.

```
int fn(int i);

int main() {
        cout << fn(3);
        return 0;
}

int fn(int i) {
        return i;
}
```

After seeing the declaration, the compiler will expect to see the definition of the declared function somewhere in your source. And since there is a valid definition for *fn(...)* with the same signature in your source, your code **will compile**.

3 Operators and Functions in C++

Complete the following tasks - each in a different cpp file. Having a different file for each task will allow you to be more organised, be able to go back to them if you need to, and for your TA to check your work easier.

- Rewrite the following code in a .cpp file. Before compiling it, read through the code and guess what the output of your program will be. Compile the program and verify your guesses.

```
int a = 0; int b = a++; int c = ++b; int d = 2;
bool e = false;
cout << a << ", " << b << ", " << c << endl;
if ((a != b) && !e) d *= 12;
cout << d << endl;
cout << (1 >= 3) << ", " << (1 & 3) << ", "
        << (1 | 3) << ", " << (~1) << ", "
        << (1 ^ 3) << ", " << (10 << 1) << endl;
```

- Implement a function that takes a reference to an integer, returns **nothing**, but multiplies its input by 2 and stores it in its single argument. In *main(...)*, declare a variable and initialise it with your favourite number. Print the value of your variable. Call the function on it and print its value again to see if it has changed.

- The factorial function is defined as follows:

$$fact(n) = \begin{cases} n \times n-1 \times \cdots \times 1 & n > 0 \\ 1 & n = 0 \end{cases}$$

  Implement the factorial function **iteratively** in C++, run your code with a few different inputs including a few negative integers, and justify your observations. Try to restrict the types of your function argument and return type to not allow illegal inputs, and make sure your program terminates gracefully (perhaps check for illegal inputs in your function if needed, catch them, and predict and avoid scenarios that can lead to stack overflows, etc.).

- By this point, you have hopefully seen two different implementations for the factorial function. We will now try to come up with a third implementation using a function with the return type *void*.

```
void fact(unsigned int& n) {
        // Write your code here ...
}

int main() {
        unsigned int n = 3;
        fact(n);
        cout << n;
}
```

Implement the factorial function **recursively** inside the body of *fact(...)* such that instead of it returning *fact(n)*, by the end of its execution, it assigns the value of $factorial(n)$ to the variable passed to it. Try both of your implementations with a number of values and make sure there is no inconsistency between the results produced by your two implementations.

- Write a function that takes an integer and returns its absolute value. **Do not use** any libraries or C++'s *abs(...)* function. Test your function with a few different values including a positive integer, 0, and a negative number.

- Suppose the oracle has given us the following functions:

```
unsigned int increment(unsigned int n) {
        return n + 1;
}
unsigned int decrement(unsigned int n) {
        return n - 1;
}
```

Using the above function, you can define **addition** for non-negative numbers using the following inductive (aka. recursive) definition:

$$add(n,m) = \begin{cases} m & n = 0 \\ add(\text{decrement}(n), \text{increment}(m)) & otherwise \end{cases}$$

Implement the above function in C++. Use the *abs(...)* function you implemented in the previous task to make your function work for negative numbers.

- Use the *add(...)* function you implemented in the last excercise to implement the following function:

$$mult(n,m) = \begin{cases} 0 & n = 0 \\ m & n = 1 \\ add(m, mult(\text{decrement}(n), m)) & otherwise \end{cases}$$

Modify the function you implemented to work with negative ns.

- Create a *struct* called *Student* with the following members:
  - name : std::string

  - birth_year : unsigned short
  - mid : std::string

  Create a constructor for it that takes three arguments and assigns them to their corresponding member variables. Write a function that takes an argument of type Student, and prints the values of each of its member variables in the following format:
  "name: {name}, b year: {birth_year}, MID: {mid}"
  , and add a line break at the end of the line. Create a few variables of type *Student* with different values for each of their member variables. Call your function on each of your variables to print their values.

- Write the following program in a .cpp file:

  ```
  int fn1(int i) {
          if (i % 2 == 0)
                  return fn2(i);
          return i;
  }
  int fn2(int i) {
          if (i % 2 == 1)
                  return fn1(i)
          return i;
  }
  ```

  Create a main function for your program and call one of the function with some integer. Try to compile the code and see what error you will get. Fix your program by adding the **minimum number of lines** needed to fix it.

- The greatest common divisor (gcd) of two numbers is the greatest number divisible by both of them. For instance, the gcd of 32 and 16 is 16 since both 32 and 16 are divisible by 16. The gcd of 12 and 9 is 3, and the gcd of 14 and 5 is 1 since they are not divisible by any number greater than 1. Use the % operator to implement a function that takes two non-negative integers, and finds their gcd. You can choose to implement it *recursively* or *iteratively.*

- The Fibbonaci sequence is defined as follows:

  $$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & \text{Otherwise} \end{cases}$$

  Implement the above function in C++. Call it in your *main(...)* function with a few different values. In particular, check how it behaves if a negative integer is passed to it and if it returns 0 for $n = 0$, 2 for $n = 3$, and 13 for $n = 7$.