CSCI 2270 – Data Structures

Recitation 1, Fall 2022

Objectives

1. Variables
2. Command line
3. Explore functions.
4. Read and write files.
5. Structs

1. Variables

A Variable is a location in a computer memory used by the program which can be accessed by their name. This avoids the need for the programmer to remember the physical address of the data in memory and instead use the variable to refer to the data when needed.

In the below example, we see that we are storing 10 in a variable named **a,** and storing 20 in a variable named **b**. The program has no reference to any physical memory address, and instead, all operations on the data 10 and 20 are done using the variables **a,** and **b.**

```
#include<iostream>

using namespace std;

int main() {
    int a = 10;
    int b = 20;
    cout << a + b << endl;
}
```

Note that the code when executed on multiple machines, or at different points in time, allocates different physical memory addresses. The beauty of variables is that YOU as a programmer do not have to remember the locations of the data. You can focus on the code-logic instead.

2. Compiling from command line

Consider we want to compile and execute the following hello.cpp. Assume hello.cpp is within folder lab1

File: hello.cpp

```
#include <iostream>

int main ()
{
    std :: cout << "Hello World!"<< std :: endl ;
}
```

1. Compiling and Executing the file in the Terminal (Command Line)
    a. Open a terminal in your environment. If you are using a linux based system open terminal in your machine. If you are using csel.io open terminal from file-->new-->terminal
    b. Navigate to the lab1 folder in the terminal with cd commands.
    c. Run the following command to compile the cpp file (ensure wsl is selected as the terminal)

```
g++ -std=c++11 hello.cpp -o hello
```

Here,

1. ‘g++’ is the name of the compiler program.
2. The ‘-std=c++11’ option tells the compiler to use the 2011 version of C++.
3. ‘hello.cpp’ is the file to be compiled.
4. ‘-o hello’ tells the compiler to write its output to a file named ‘hello’ (‘hello.exe’ on Windows). If this is missing, the output file will be named ‘a.out’ or ‘a’ by default.
5. If the last command was successful, there should now be another file named ‘hello’ (‘hello.exe’ on Windows).
6. To run the program, run the command ‘./hello’ (or simply hello on Windows).

3. Functions

One of the principles of software development is DRY (Do not Repeat Yourself) aimed at reducing repeated code. One way to achieve this is by using functions. For example, if we would like to add two numbers, and call it multiple times, we can create a function called **add**.

We do this by creating separate header file to declare the function (to avoid another programmer to rewrite the function signature). We start with the first of the three files.

Here, we define the function declaration. Saying, we have a function called **add** and it takes two integer arguments as input.

File: function.h

```
int add (int a, int b);
```

In another file, we define the function by adding logic into it.

File: funcdef.cpp

```
#include "function.h"
int add ( int a, int b)
{
  return a + b;
}
```

Then, we call the declared function multiple times on as many different values of inputs as we need. Moreover, the header and function definition files can be shared with multiple programmers. *Notice that we have only included the header file in the main cpp file.*

File: main.cpp

```
#include <iostream>
#include "function.h"

using namespace std;
int main ()
{
  // Calling the function for 2+3
  cout << "2+3=" << add(2, 3) << endl ;
  // Calling the same function for 4+5
  cout << "4+5=" << add(4, 5) << endl ;
  return 0;
}
```

To compile multiple files, we pass only the cpp files as shown below.

```
g++ main.cpp funcdef.cpp -std=c++11 -o func
```

Handling command line arguments

main is also a function. Can it take arguments? If so then how can we provide those arguments? The answer is command line arguments.

Often, we would like our code to take multiple arguments as input and process it accordingly. This is done by modifying the signature of the main function from this

```
int main ()
```

To this.

```
int main (int argc, char const *argv[])
```

Notice the change in the function signature. It now accepts two parameters: **argc** and **argv.** **argc** stores the count of the total number of arguments you have passed in the command-line on execution, and the second one **argv** is an array of strings storing the arguments passed in the command-line. The following program reads an arbitrary number of arguments from the command line and prints them as output.

File: commandLine.cpp

```
#include <iostream>
int main ( int argc, char const *argv[])
{
   std :: cout << "Number of arguments: " ;
   std :: cout << argc << std :: endl ;
   std :: cout << "Program arguments: " << std :: endl ;

   for ( int i = 0 ; i < argc; i++) {
      std :: cout << "Argument #" << i << ": " ;
      std :: cout << argv[i] << std :: endl ;
   }
}
```

Compile the above code following the same syntax as mentioned before in this document.

```
g++ -std=c++11 commandLine.cpp -o commandLine
```

Example1 : Simulating no arguments

To simulate no arguments, execute the code as follows (pass no arguments)

```
./commandLine
```

Here, the main function only receives one argument, which is the name of the program itself. Thus, **argc** is one, and **argv** is an array of length 1, where the only element in this array is a string "./commandLine".

Example2 : More arguments

We can pass multiple arguments by typing each one after the function name, separated by spaces. So, if we run the program using the command (we pass 3 strings called arg1, arg2, arg3)

```
./commandLine arg1 arg2 arg3
```

Now **argc** is 4 and **argv** is an array of length 4. The first string in the array is the program name "./commandLine", and the rest of them are the strings we typed on the terminal, delimited by spaces or tab.

4. File I/O

Some programs require reading data from files to process them, and some programs process input and outputs large amount of data which cannot be perused if printed to the terminal. C++ allows File I/O (input/output) by using *ifstream* and *ofstream* for reading and writing, respectively.

You replace **cin** and **cout** operators with the following for reading and writing.

First, you declare an instance of file input (the file from which you read the data). Note that this file must be in the same directory as the code executable. Otherwise, you must provide a fully qualified path name as the argument.

```
  ifstream iFile("somefile.txt");
```

Similarly, to output data into a file, you must declare an instance of file output. By default, the file will be in the same directory as the code executable. Otherwise, you must provide a fully qualified path name as the argument.

```
  ofstream oFile("somefile.txt");
```

We can provide an additional argument for output, whether to append at the end of the file, or overwrite the file before printing. Some of those operation modes are given below.

File operation modes:

```
ios::app // Append to the file
ios::ate // Set the current position to the end of the file
ios::trunc // Delete everything in the file
```

File mode example:

```
  ofstream iFile("test.txt", ios::app);
```

CSCI 2270 – Data Structures

Recitation 1, Fall 2022

In the next page, we have provided sample programs for file input and file output.

File output example - oFile.cpp

On compilation and execution, check for the newly created “filename.txt”

```
#include <fstream>
#include <iostream>

using namespace std ;

int main ()
{
  // File Writing
  //Creates instance of ofstream and opens the file
  ofstream oFile ( "filename.txt" );
  // Outputs to filename.txt through oFile
  oFile<< "Inserted this text into filename.txt" ;
  // Close the file stream
  oFile.close();
}
```

File input example - iFile.cpp

```
#include <fstream>
#include <iostream>

using namespace std ;
int main ()
{
  // File Reading
  char str[ 10 ];
  //Opens the file for reading
  // Ensure that filename.txt is present in the same directory
  // as that of the source file
  ifstream iFile ( "filename.txt" );
  //Reads one string from the file
  iFile>> str;
  //Outputs the file contents
  cout << str << "\n" ;
  // waits for a keypress
  cin.get();
  // iFile is closed
  iFile.close();
}
```

5. Structs

Sometimes, to represent a real-world object, simple datatypes are not enough. To represent a Employee, we need some (if not all) of the following: age, gender, date-of-birth, name, address etc.

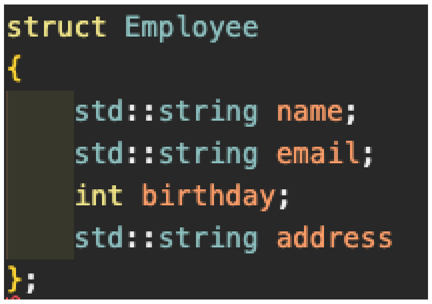
You could write code as follows:

```
std::string name;
std::string email;
int birthday;
std::string address;
```

This becomes increasingly complex if we would like to store multiple Employee representations. One way to solve this is by using aggregated(grouped) user-defined datatype called **Struct**. This datatype can hold different datatypes grouped under a common variable of type Struct.

CSCI 2270 – Data Structures

Recitation 1, Fall 2022



```
struct Employee
{
    std::string name;
    std::string email;
    int birthday;
    std::string address
};
```

Exercise

Your zipped folder for this Lab will have a main.cpp, addEmployee.cpp and addEmployee.hpp files. Follow the TODOs in these files to complete your Recitation exercise!

The task of main function will be:

1. Take filename as command line argument
2. Open the file
3. Create an employee array.
4. Read data from file and call the function addAnEmployee to add the employee in the array

The function addAnEmployee can be found in addEmployee.cpp. It will do the following

1. It will receive the name, email and birthday of the employee as arguments. It will also receive the array and the index as arguments.
2. It will add an employee record in the array at specified index.
3. It will return the next index of the array