



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

Objectives

1. Cloning a Repo
2. Location of Test Cases
3. Compiling code in CSEL
4. Debugging

1. Cloning a Repo

Steps:

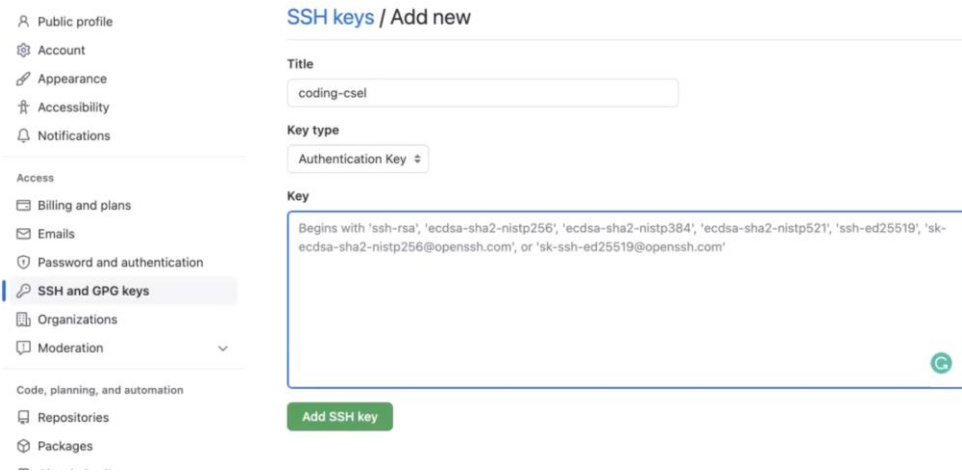
- Log in to GitHub
- Setting up SSH keys on CSEL
- Copy and Paste the keys on GitHub
- Clone the repository on CSEL

For setting up SSH keys on Github and CSEL:

Video overview of how-to setup Git with CSEL:

https://youtu.be/7_x86HeogIQ

You are REQUIRED to watch this whole video



The screenshot shows the GitHub 'SSH keys / Add new' page. On the left is a sidebar with navigation links: Public profile, Account, Appearance, Accessibility, Notifications, Access, Billing and plans, Emails, Password and authentication, SSH and GPG keys (highlighted), Organizations, and Moderation. The main content area has a title 'SSH keys / Add new'. Below it is a 'Title' input field containing 'coding-csel'. The 'Key type' is set to 'Authentication Key'. The 'Key' input field contains a long string of characters, with a hint text above it: 'Begins with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com''. At the bottom right of the key input field is a green circular icon with a white 'C'. At the bottom of the form is a green 'Add SSH key' button.



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

```
Terminal 1
jovyan@jupyter-namy6667:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jovyan/.ssh/id_rsa):
/home/jovyan/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/jovyan/.ssh/id_rsa
Your public key has been saved in /home/jovyan/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:qLDOap8kVsailP2ETSIWW/kvMVTxo4cKpibShvu6HVU jovyan@jupyter-namy6667
The key's randomart image is:
+---[RSA 3072]-----+
|  . . . o .      |
|  + . . .      |
|  + .o.E o      |
|  . = = .o .    |
|  +.O.o=oS.     |
| oo*++o...     |
| ==..O.        |
| =Bo..         |
| *==o         |
+---[SHA256]-----+
jovyan@jupyter-namy6667:~$ cat /home/jovyan/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQGCwluw5/yF60OVZ/k2s15Nw8zdU1+wEdYm+uuHVWKwOCY6+XOVbQsEy8FTb17h9mX43Vnz/FOgb
hlwfyt5chzstZ5mSq13b1CAONfgmPU+26ROXQ2UxcW+ZDPigxGw2lqe4/jSk1+gkB6etsJRIZ6gN0vR2A24Ff2uJlLGLXFo5c/whMrAiHGgEah9y
/i761lauZB09kDpk0GyOYQvmGBs09iNQBPj8vBFy+vtEijhbd0bj6g1cUwqJ5//lKfoPOcMdJLjofIMnPzpBdnADFP8EPaCF6dvvrPEZXOAvPWG3Sq
DBVmJJgI82SFwmH3Q6BzJhKG4cd+YEU6q0Zuq/S6PJ7dQgDueX0v/+axbcPC7bq8t9OB05p1glZdADbCv6CmbLY969+ouIir/H5vMqYNzOXJmF+E
wf4h/Eam4EnoEYfWAKpketqIFDLtLk7AaiLO5UHuJqLJ2QvQwh54RnK0TIPE4mNiY/KGPs88fiRMclRhUu3fW1WLzUOAz6QVflaCfnc= jovyan@
jupyter-namy6667
jovyan@jupyter-namy6667:~$
```

- run ssh-keygen command
- cat the output
- paste the output on Github to create the key

2. Location of Test Cases

Use the git clone command to clone your assignments.

```
jovyan@jupyter-namy6667:~$ git clone git@github.com:cu-csci-2270-fall-2022/assignment-0-namrathamk.git
Cloning into 'assignment-0-namrathamk'...
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 20 (delta 0), reused 15 (delta 0), pack-reused 0
Receiving objects: 100% (20/20), 7.33 KiB | 1.22 MiB/s, done.
```

After cloning the repository you will see a tests folder that has all the testcases which we will run on your code. It should be your starting point for debugging any seg fault or if your code is breaking. In cpp files with test prefix you will see TEST_F functions which are isolated testcases. In the below example there are 2 test cases TestFooA and TestApp_1



CSCI 2270 – Data Structures

Recitation 1, Spring 2023

The screenshot shows a Visual Studio Code editor with a project named 'ASSIGNMENT-0-NAMRATHAMK'. The Explorer sidebar on the left shows the file structure: `.vscode`, `app_1` (containing `main_1.cpp`), `build`, `code_1`, `tests` (containing `test_preview.cpp`), `.gitignore`, `CMakeLists.txt`, `CMakeLists.txt.in`, and `readme.md`. The main editor window displays `test_preview.cpp` with the following code:

```
87
88 TEST_F(test_x, TestFooA){
89     int fooAreturn = fooA();
90     ASSERT_EQ(-3,fooAreturn);
91     add_points_to_grade(50);
92 }
93
94 TEST_F(test_x, TestApp1){
95     string resp = exec("./run_app_1 hello penguin");
96     string desired_result = "i'm a computer\n";
97     ASSERT_EQ(resp, desired_result);
98     add_points_to_grade(50);
99 }
100
```

The bottom panel shows the 'TERMINAL' tab with the following output:

```
bash - build
-- Detecting CXX compile features
-- Detecting CXX compile features - done
>> GTest was found among local libraries.
-- Configuring done
-- Generating done
-- Build files have been written to: /home/jovyan/recitation-1/assignment-0-namrathamk/build
jovyan@jupyter-namy6667:~/recitation-1/assignment-0-namrathamk/build$ make
[ 16%] Building CXX object CMakeFiles/run_tests_1.dir/tests/test_preview.cpp.o
[ 33%] Building CXX object CMakeFiles/run_tests_1.dir/code_1/functions.cpp.o
[ 50%] Linking CXX executable run_tests_1
[ 50%] Built target run_tests_1
[ 66%] Building CXX object CMakeFiles/run_app_1.dir/app_1/main_1.cpp.o
[ 83%] Building CXX object CMakeFiles/run_app_1.dir/code_1/functions.cpp.o
[100%] Linking CXX executable run_app_1
[100%] Built target run_app_1
jovyan@jupyter-namy6667:~/recitation-1/assignment-0-namrathamk/build$ ls
```

3. Compiling code in CSEL

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

File: hello.cpp

```
#include <iostream>

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

1. Compiling and executing the file in the Terminal (Command Line)

- Open a terminal in your environment. If you are using a linux based system open terminal in your machine. If you are using `cse1.io` open terminal from File->New->terminal
- Navigate to the lab1 folder in the terminal with `cd` commands.
- Run the following command to compile the .cpp file

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

Here,

- 'g++' is the name of the compiler program.



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

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).

4. Debugging

Even the best programmers will occasionally make mistakes when writing code. Although we may intend for code to function in a particular way, mistakes are easy to make. It is therefore necessary go through the process of debugging to ensure that code is error free. Debugging is also a vital tool in determining why code is behaving in a particular way. When your code is behaving in an unintended way, you might wish to follow its execution line-by-line to see where it goes wrong.

Basic Debugging: Print Statements

The simplest form of debugging is using print statements to monitor the progress of the program. The program below is an example of a program with a bug in it. When we set my_number to 12, we expect it to execute the middle case and multiple the number by two. We expect the output to be 24, however when run the program prints 576.

```
int main(int argc, char *argv[])
{
    int my_number = 12;

    if(my_number < 10)
    {
        my_number = 0;
    }
    else if(my_number < 20)
    {
        my_number *= 2;
    }
    else;
    {
        my_number *= my_number;
    }
    cout << my_number << endl;
}
```



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

To figure out where the code is going wrong, we will add print statements to track the progress of `my_number` as the code executes:

```
int main(int argc, char *argv[])
{
    int my_number = 12;

    if(my_number < 10)
    {
        my_number = 0;
        cout << "The number is less than 10 with a value of: " << my_number << endl;
    }
    else if(my_number < 20)
    {
        my_number *= 2;
        cout << "The number is greater than 9, but less than 20 with a value of: " << my_number << endl;
    }
    else;
    {
        my_number *= my_number;
        cout << "The number is greater than 19 with a value of: " << my_number << endl;
    }
    cout << my_number << endl;
}
```

When we run the code, we get the following output in the console:

```
The number is greater than 9, but less than 20 with a value of: 24
The number is greater than 19 with a value of: 576
576
```

Something is not right here. The number 12 cannot be both less than 20 and greater than 19. Similarly, the first line of the output has the correct value of 24. Using the information, we can realize that the code in the `else if` AND in the `else` case are running. Looking back at the code, we can see that we mistyped the `else` case, adding a semicolon at the end. The C++ compiler treats this as valid code; however, the mistype causes nothing to happen in the `else` case, and the `my_number *= my_number` and print statement to *always be executed*. Removing the extra semicolon will fix the problem.

```
else;
{
    my_number *= my_number;
    cout << "The number is greater than 19 with a value of: " << my_number << endl;
}
```

A simpler way to debug: The debugger tool (gdb)

Although print statements can be used to track the progress of our program, they require a lot of work to add to our code. Luckily, we can use a *debugger* to accomplish the same task. A debugger is a special tool that allows us to interactively run code line by line and see the result. In this course we will use `gdb` for debugging in CSEL. The `gdb` tool can be used in two ways: from the command line & visually with VS Code. For this tutorial we will use `gdb` visually in VS Code. For more information on using `gdb` from console see the debugging tutorial document on the course Canvas.



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

How to debug in VS Code:

To use the gdb debugger in VS Code, you will need to ensure that your project is open in VS code and that the proper configuration files are present.

To open a homework or recitation assignment in CSEL:

1. Open VS Code IDE *in CSEL*
2. In the top left go to File -> Open Folder
3. Navigate inside of /home/jovyan
4. Select the assignment you wish to open that you have cloned from GitHub
5. VS Code will refresh and say the assignment name as the root of the project (i.e. ASSIGNMENT-0-YOUR-GITHUB-USERNAME)

Important note:

For the debugger to function properly there must be a configuration present in the *root* of project directory. For this course, all assignments will include this in a `.vscode` folder. If VS Code says that no configuration is present, please ensure that you have done the Open Folder step above to open just a single assignment at a time.

Watching our code run using breakpoints:

When we use the debugger, we need to tell it a line of code to pause on. To do this we use breakpoints. Think of them as stop signs for our code. You can add a breakpoint to a line by clicking to the left of the line number in VS Code. When we run the debugger, it will pause on the line of code containing a breakpoint and wait for us to tell it to continue.

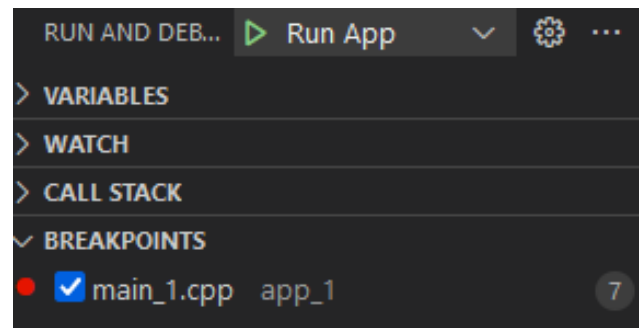
Break point

```
14 // loop through every character in string to
15 int sentence_length = sentence.length();
16 for (int i = 0; i < sentence_length; i++)
17 {
18     char current_char = sentence[i];
19     int index = (int) current_char - 97; //
```

Running the program:

1. On the left toolbar in VS Code select the play button with a bug on it. This will open the debug panel. It will show information when the debugger is running, as well as a list of all breakpoints you have placed.

2. Click the green play button to start the debugger. The program will compile, and assuming there were no errors, the debugger will run the program until it stops at a breakpoint or terminates.





CSCI 2270 – Data Structures

Recitation 1, Fall 2023

Controls for the debugger:

While the debugger is running your program, a toolbar will appear at the top-center of VS Code. Below is an overview of what each button on the toolbar does:

Resume	Step Over	Step Into	Step Out	Start Over	Stop
Runs to next breakpoint	Go to next line of code as written	Go to next line of code that executes	Go to next line, outside of current function call		

If you wish to go line-by-line through the program, you might want to add a breakpoint at the start of the program and use the step-over button to advance the code.

Using the debugger to look at variables:

One of the most powerful features of the debugger is looking at the value of variables as your code executes. While the debugger is stopped at a breakpoint, you can peer into the variables of the program. In the debugging pane, the variables window shows the value of the variables at that instance of time. The example below shows this in action. **Note:** the variable `b` is shown as 0 instead of 2 because the breakpoint was placed on line 8. *Breakpoints stop execution before the line they are placed on is executed.* We can use the “step over” button at this point to see the value of `b` change in the window. Another interesting feature of the debugger is the ability to change the value of variables during program execution. For example, if we wanted to change variable `a` to be 5 *after line 7 has already been executed*, we can double click the variable in the window and set a new value. This feature can be useful to test different values when debugging code.

```
app_1 > main_1.cpp > main(int, char * [])
1  #include <iostream>
2  #include <fstream>
3
4  using namespace std;
5
6  int main(int argc, char* argv[]){
7      int a = 1;
8      int b = 2;
9
10     return a + b;
11 }
```




CSCI 2270 – Data Structures

Recitation 1, Fall 2023

Debugging multiple functions: the call stack

One common problem that you might face when debugging code is knowing where a particular function is being called from. The example below shows a simple function called `add()` that takes two numbers and returns the sum. If we place a breakpoint in the function on line 8, we can see the result in the debugging pane. The call stack window shows us the history of functions called to reach this breakpoint. The call stack is shown in newest to oldest order. So, in this example we can see the first function call (to `main`), followed by the `main` function's call to `add()`. If you click on the name of a previous function call, it will highlight the next caller line in green and show you the variables within that scope.

The screenshot shows the VS Code debugger interface. On the left, the 'VARIABLES' pane shows local variables: `a: 1`, `b: 2`, `result: 0`, and `argc: 1`. Below it, the 'CALL STACK' pane shows the current function `add(int a, int b)` at line 8 of `main_1.cpp`, and the caller `main(int argc, char ** argv)` at line 15. The main code editor on the right shows the `add` function and the `main` function. A breakpoint is set at line 8 of the `add` function. The `main` function calls `add(a, b)` on line 15.

Debugging programs with command line parameters:

Some programs will require the user to launch them with command line arguments. These could be configurations, file paths, etc. These command line arguments can be accessed in your code by using `argc` (the count of arguments) and `argv` (the variables passed). All programs launched in C++ have at least one argument: the name of the program/ file called when launching. For example if you use the following `g++` command to compile your program, and use `./my_program` to launch it, `argc` would be 1 and `argv[0]` would be “my_program”.

```
cwade@SyzygyDesktop:~/TA/recitation-1$ g++ main.cpp addEmployee.cpp -o my_program
cwade@SyzygyDesktop:~/TA/recitation-1$ ./my_program
```

Recitation 1's assignment will require you read in a file name from the command line arguments. If you are using command line you could just simply do:

```
cwade@SyzygyDesktop:~/TA/recitation-1$ ./my_program employees.csv
```

In this case `argc` equals 2 and `argv[1]` equals “employees.csv”

But how do we supply command line arguments when we use the debugger?

To do this, we will modify the configuration file used to tell VSCode how to launch our program.

Steps:

1. In the Explorer panel of VSCode open the '.vscode' folder.
2. Open the 'launch.json' file.



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

3. The file should look like the following. On line 9, you can add command line arguments to the list of “args”.

```
1 {
2     "version": "0.2.0",
3     "configurations": [
4         {
5             "name": "Build and debug",
6             "type": "cppdbg",
7             "request": "launch",
8             "program": "${workspaceFolder}/program",
9             "args": [],
10            "stopAtEntry": false,
11            "cwd": "${workspaceFolder}/",
12            "environment": [],
13            "externalConsole": false,
14            "MIMode": "gdb",
15            "setupCommands": [
16                {
17                    "description": "Enable pretty-printing for gdb",
18                    "text": "-enable-pretty-printing",
19                    "ignoreFailures": true
20                }
21            ],
22            "preLaunchTask": "build",
23            "miDebuggerPath": "/usr/bin/gdb"
24        }
25    ]
26 }
```

4. Below is an example of adding three command line arguments. You must surround each argument string with quotes. Multiple arguments should be separated with commas like shown.

```
"args": ["employees.csv", "4", "dummy"],
```

5. Now when you run the debugger normally, it will add the arguments.

5. Extras: 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.



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

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.



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

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)



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

```
./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.

6. Extras: 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");
```



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

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);
```

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();
}
```



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

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();
}
```

7. Extras: Structs

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.

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



CSCI 2270 – Data Structures

Recitation 1, Fall 2023

8. Extras: Reading C++ compiler error messages

```
brcr5319@BMC-LAPTOP:/mnt/c/Repos/CSCI1300/recitation3$ g++ -std=c++17 lowerToUpper.cpp
lowerToUpper.cpp: In function 'int main()':
lowerToUpper.cpp:9:39: error: expected ';' before 'getline'
   9 |     cout << "Enter a string: " << endl
     |                                     ^
   10 |     getline(cin, input);
     |     ~~~~~

```

File Name: lowerToUpper.cpp
Line Number: 9
Column Number: 39
Google-able Error Name: error: expected ';' before 'getline'

Exercise

The repo for this assignment 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