CSCI 2270 – Data Structures

Recitation 1, Aug 2018

Objectives:

1. Setup moodle and piazza accounts
2. Atom editor and g++ installation
3. Run code using Atom and command line
4. Explore functions
5. Pass by value / address
6. Read and write files

1. Moodle Account

All students in CSCI 2270 this semester will be accessing course materials through the Computer Science Moodle:

http://moodle.cs.colorado.edu

To use Moodle, you need to login using your CU identikey and password, and then enroll in your class. Once you've logged in, select the **'CSCI 2270 - Gupta - CS1: Data Structures'** class and enter the enrollment key: "**datastruct**" to enroll. Once you're enrolled in the class, you should see the course materials that have been uploaded so far, organized by weeks.

Moodle Account Test

To test that you have set up your Moodle account and are able to submit your assignments, **upload a file to the Recitation 0 Submit link**. It doesn't matter what file you submit; this exercise is to verify that you know how to upload your assignments before the first assignment is due. **You should receive a confirmation email once your file is submitted**. Verify that you receive that email. If you do not receive this email, you did not submit correctly.

2. Piazza

Enroll here for CSCI 2270 Computer Science 2 Data Structures class. Login using your colorado.edu account.

3. Installation of Atom editor

We will be using Atom as the default code editor for this course.
Steps to setup the environment:

1. Download Atom editor from here.
2. The Atom website is smart enough to detect your operating system and gives an option to download the software based on your OS.
3. OS Specific installation:
   a. Windows:
      - Download and double click the installer.
   b. Mac:
      - Download the zip file.
      - Extract the zip file by double clicking it.
      - Copy the Atom software to your Applications folder.
   c. Linux:
      - For CentOS, download the .rpm file
        - Double click the .rpm file to install OR
        - Execute “rpm -i <atom-file>.rpm” using command line
      - For Ubuntu, download the .deb file
        - Double click on the .deb file
        - An Installer window will open. Click on install

4. **For Windows users only** who don't have the C++ compiler:
   1. Download the C++ Compiler from this link: https://sourceforge.net/projects/mingw/files/Installer/mingw-get-setup.exe/download
   2. Run the downloaded installer. You won’t need to change anything until you’re prompted for packages to install. Select ‘mingw32-base’, ‘minsys-base’, and mingw32-gcc-g++’. Then select ‘apply changes’ from the ‘installation’ dropdown.
   3. After installing, open the ‘Command Prompt’ program. Type "sysdm.cpl" and press enter.
   4. Select the ‘Advanced’ tab, then click the ‘Environment Variables’ button.
   5. In the ‘System Variables’ section, click on ‘Path’ button then click ‘Edit’.
   6. Click ‘New’, then add the path to where you installed the C++ compiler (default C:\MinGW\bin if you didn’t change any settings during installation) then click ‘Ok’.
   7. Restart your command prompt from step (2). Type ‘g++’ and hit enter: it should say something about ‘no input files, compilation terminated’. If it doesn’t, or if it throws an error about g++ not being recognized as a command, make sure you’ve completed the above steps correctly or contact your TA.

5. For Mac users only who don’t have the C++ compiler:

1. xcode-select --install using terminal. (Installs command line dev tools)
2. Refer here for more details.

6. Adding g++ compiler to Atom

   Open Atom -> Welcome Guide -> Add Ons -> Install a package->
   Open Installer -> Search for gpp -> Select gpp-compiler -> Install

7. Linux users run **sudo apt-get install g++** using command prompt.

4. Programming exercise to run code using Atom, Passing and handling command line arguments

4a. Programming exercise to run code using Atom

- Create a folder (for eg. lab-1) before starting in your home directory. Steps:
  1. Open File tab
  2. Select Open Project
  3. A File explorer window opens
  4. Create a folder "lab-1"
  5. Click OK
  6. A folder will be opened in the Project window in Atom editor.
- Creating a single .cpp file
  1. Right click on the folder in your Atom Editor
  2. Select New File option
  3. Name the file "hello.cpp"
  4. Type the following contents in the file:

File: hello.cpp

```
#include <iostream>

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

- Building the .cpp file using Command line
  - ❖ Open a terminal (Mac/Linux) or command prompt (Windows)
  - ❖ Use the 'cd' command to move to the directory where hello.cpp is present.
    1. The terminal shows you which directory you're currently in.
    2. Use the command 'cd <name>' to move to the directory with that name.
  - ❖ Run the command 'g++ -std=c++11 hello.cpp -o hello'.
    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.
   ‘-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.
4. If the last command was successful, there should now be another file named ‘hello’ (‘hello.exe’ on Windows). To run the program, run the command ‘./hello’.

4b. Passing and handling command line arguments

When you run your program by it starts by running the main function in your source code. Main function can also receive arguments if it is declared like this:

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

Notice that there are two parameters passed to main function from the terminal. The first one, **argc**, is the total number of arguments you passed to the main function when you’re running your program on the terminal. The second one, **argv**, is an array storing all the arguments you passed. Change your program to the following code:

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;
    }
}
```

Example 1: No arguments.

The first example is a straightforward one. Recompile and run your command using the same steps are before. 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”.

Example 2: 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:

```
./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.

5. Functions

Having a separate header file to declare function is useful when we need to reuse the function in multiple source files.

Function declaration in C++ or prototype.
File: function.h

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

Function definition
File: funcdef.cpp

```
#include "function.h"

int add(int a, int b)
{
      return a + b;
}
```

Calling a declared function
File: main.cpp

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

using namespace std;
int main()
{
    cout << "2+3=" << add(2,3) << endl;
    return 0;
}
```

Compiling multiple files

```
g++ main.cpp funcdef.cpp -o func
```

6. Pass By Value/ Address

Pass by Value

Consider the below code:

File: passByValue.cpp

```
#include <iostream>

using namespace std;
void add2(int num)
{
   num = num + 2;
}

int main()
{
   int a = 10;
   add2(a) ;
   cout << a;
}
```

What do you think the output will be? 12?

To your surprise it will be just 10. When we pass a variable as argument to a function in the system stack the function creates the local copy of the variable and performs the operation on that local copy. The caller function (main) has no knowledge of that local copy. Hence the change is local to the callee function (add2).

Pass by Address

Now consider this version-

File: passByAddr.cpp

```cpp
#include <iostream>

using namespace std;
void add2(int* num)
{
   *num = *num + 2;
}
int main()
{
   int a = 10;
   add2( &a );
   cout<< a;
}
```

In this case we are passing the address of a. The function will again create a local copy of the address. However since both of this addresses are same they will refer to to the variable a. Hence changing the value at the pointer will change the value of a and that change will be persisted. **Now examine the following code and try to find why it is persisting the change?**

File arrExmp.cpp

```cpp
void add2(int a[], int len)
{
   for(int i=0;i<len;i++)
      a[i]+= 2;
}
int main()
{
   int a[] = {1,2,3};
   add2( a, 3 );
   for(int i=0;i<3;i++)
      cout<< a[i] << endl;
}
```

Is the previous one similar/different to the one given below?

File arrExmp2.cpp

```
void add2(int* a, int len)
{
   for(int i=0;i<len;i++)
      a[i]+= 2;
}

int main()
{
   int a[] = {1,2,3};
   add2( &a[0], 3 );
   for(int i=0;i<3;i++)
      cout<< a[i] << endl;
}
```

7. File I/O

File I/O is reading or writing from files. C++ uses ifstream and ofstream for reading and writing respectively.

Declaring an instance of file input:

```
ifstream iFile ( "filename" );
```

Similarly for file output:

```
ofstream oFile ("filename");
```

File operation modes:

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

File mode example:

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

File output example - oFile.cpp

```cpp
#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

```cpp
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
}
```