CMSC 202 Fall 2020

Lab 01 – Introduction and Practicing C++

Assignment: Lab 01 – Introduction and Practicing C++
Due Date: Sunday, February 7th by 11:59pm
Value: 10 points

Overview

In this lab you will:

- Review how labs are graded
- Review basic Emacs and Unix commands
- Create and compile a simple C++ program
- Create a simple makefile
- Write a program that calls a function
- Practice using console input and output (including `getline()`)
- Get some experience with control structures in C++

There are two parts to this lab – introduction and practicing input. As some people will have transferred here and never experienced our GL environment, the first part of this lab introduces how to log into GL and how to use emacs. For our labs, you will be expected to display your code running on GL.

We will use “lab” and “discussion” interchangeably this semester.

Part 1 – Introduction

A. Lab Grading

There will be 13 labs assigned over the course of the semester; your best 10 scores will be used to compute your lab average. All lab sessions are led by Graduate Teaching Assistants (TA) or Undergraduate Teaching Fellow (TF). To receive full credit for your lab, you must complete both your prelab quiz and your lab assignment (done with your TA/TF). Your work must be verified by your TA/TF.

Starting with lab 2, you must attend your assigned lab section and the lab assignment must be completed during the assigned lab time on Discord.

During the lab, your TA/TF will explain the lab assignment, help as needed, and record your successful completion of the assignment.

Starting with lab 2, lab grades in CMSC 202 will have two components.

The two components are:

1. Component 1 is a pre-lab quiz on Blackboard (4 points).
    a. Prelab quizzes are released on Friday mornings and are due Monday at 10am.
2. Component 2 is the lab in discussion (6 points).
    a. This is completed during your scheduled time on Discord.

Score	Description
6	Successfully completed the lab assignment
4	Made a good attempt to complete the lab assignment
2	Attended the lab, but made little or no effort to complete the assignment
0	Did not attend the lab

B. Lab Management

We will NOT be meeting for this lab. You will complete this lab on your own. If you have questions, you can log into Discord and ask during office hours (or just generally to any of the TA/TFs).

Take minute to look and see who your TA/TF is for the semester. You should look at your schedule and see which lab section you are registered for. Then look it up in this table:

Section	Faculty	Day	Time	TA/TF	Email Addresses
10	Dixon,Jeremy A	T/TH	10:00AM-11:15AM		
11		T	11:30AM-12:20PM	Kiran Jambhale	kjambha1@umbc.edu
12		TH	11:30AM-12:20PM	Kiran Jambhale	kjambha1@umbc.edu
13		T	1:00PM-1:50PM	Poorvi Harsora	poorvih1@umbc.edu
14		TH	1:00PM-1:50PM	Poorvi Harsora	poorvih1@umbc.edu
15		TH	8:00PM-8:50PM	Akarsh Kashamshetty	akarshk1@umbc.edu
20	Robinson, Dawn	MW	5:30PM-6:45PM		
21		M	7:00PM-7:50PM	Akshay Reddy Akkati	aakkati1@umbc.edu
22		M	4:00PM-4:50PM	Nonso Ugwu	chiugwu1@umbc.edu
23		M	1:00PM-1:50PM	Nonso Ugwu	chiugwu1@umbc.edu
24		M	3:00PM-3:50PM	Nonso Ugwu	chiugwu1@umbc.edu
25		M	10:00AM-10:50AM	Jake Miu	jmiu1@umbc.edu
30	Joeg,Prasanna	T/TH	2:30PM-3:45PM		
31		T	5:30PM-6:20PM	Aristidis Dernelakis	ad20@umbc.edu
32		TH	5:30PM-6:20PM	Eddie Nieberding	edwardn3@umbc.edu
33		T	8:00PM-8:50PM	Hamza Asad	sasad1@umbc.edu
34		TH	10:00AM-10:50AM	Kiran Jambhale	kjambha1@umbc.edu
35		TH	1:00PM-1:50PM	Akarsh Kashamshetty	akarshk1@umbc.edu
40	Joeg,Prasanna	T/TH	4:00PM-5:15PM		
41		T	10:00AM-10:50AM	Kiran Jambhale	kjambha1@umbc.edu
42		TH	2:30PM-3:20PM	Akarsh Kashamshetty	akarshk1@umbc.edu
43		T	2:30PM-3:20PM	Daniel Milani	dmilani1@umbc.edu
44		TH	1:00PM-1:50PM	Poorvi Harsora	poorvih1@umbc.edu
45		T	1:00PM-1:50PM	Poorvi Harsora	poorvih1@umbc.edu
50	Dixon,Jeremy A	MW	1:00PM-2:15PM		
51		M	7:10PM-8:00PM	Akshay Reddy Akkati	aakkati1@umbc.edu
52		W	7:10PM-8:00PM	Nonso Ugwu	chiugwu1@umbc.edu
53		W	2:30PM-3:20PM	Akshay Reddy Akkati	aakkati1@umbc.edu
54		W	2:30PM-3:20PM	Akshay Reddy Akkati	aakkati1@umbc.edu
55		W	4:00PM-4:50PM	Akarsh Kashamshetty	akarshk1@umbc.edu

On Friday, February 5th, your first prelab quiz (for lab 2) will be released on Blackboard at 9am.

Starting the week of February 8th, you should log into Discord for lab 2 at your scheduled time to work on your lab. With few exceptions, all questions about your lab should be directed to your TA/TF. Before your lab, take a few minutes and practice the concepts for the lab meeting.

C. UMBC Computer Environment (GL)

At UMBC, our General Lab (GL) system is designed to grant students the privileges of running a website, storing files, and even developing applications. GL itself is a Linux-based system that we will interact with using a command line interface (CLI). All major operating systems (Windows, Mac, and Linux) can log into GL.

In order to log into GL, you will need to have a Secure Shell (SSH) client. SSH is a network protocol that connects a client computer to a server securely, without the risk of exposing your password or any other sensitive data to anybody watching the network traffic. It is an alternative to Telnet and is a leading standard in remote login. So, what does all this mean? Simply put, SSH allows you to use your home computer to connect to a server (i.e., gl.umbc.edu) and issue commands as if you were sitting at that computer. So,

once you SSH into GL, you can do everything you would be able to do from a computer on-campus. This client is built into Mac and Linux. For Windows, we recommend downloading the open-source client PuTTY. Another option is Bitvise (you want the client not the server). Personally, I use Bitvise as it includes an easy sFTP client.

In Windows 10, you can just use SSH from the command line. To do this, you simply go to the Windows icon in the bottom left hand corner and left click. In the search bar, type “cmd” then return. When you get to the command prompt, you can type ***ssh username@gl.umbc.edu*** where you replace username with *your gl username*. When it logs in, you will enter your password. Please note, your password will not be visible, and the cursor will not move.

In Lab 1, we will be logging onto GL and setting up folders for 202 in your home directory. We’ll create some practice code and then do a programming assignment to submit to Blackboard.

Logging onto GL

For Windows Users Only

1.1.1. Option 1 - PuTTY

Watch this video (https://youtu.be/JCK5ZD0ZWW8) , and download and install the PuTTY program. PuTTY will allow you to log into GL from your machine.

1.1.2. Option 2 – Bitvise (Recommended)

Watch this video (https://youtu.be/nFjcVLziCOI) and follow its instructions to log onto GL using Bitvise client.

For Mac Users Only

1.1.1. Step 1:

Watch this video (https://www.youtube.com/watch?v=UTU-CTH_xLw) and follow its instructions to log onto GL using the Terminal application already installed on your machine.

D. GL and your home directory

In this part of the lab, we will log onto GL and create some folders in your home directory. These folders will be where you store your labs and assignments for 202 this semester.

Since GL is a command line interface, we can't navigate it by using our mouse to click on folders. Instead, we must type in commands to tell it what we want to do. The solid green rectangle (or something similar on your machine) is where our cursor is. After typing in a command, you must hit enter for the system to know you are done.

After logging onto GL in Part 1, you are in your home directory (remember, directory is just another name for folder). We will first make a 202 directory in your home directory. The command we will use to do this is **`mkdir`**, which stands for "make directory." We also have to tell the system what the name of the directory will be. In this case, the name is "202", so type **`mkdir 202`** now, then hit enter. Remember that Linux commands are case sensitive!

```
linux3[1]% mkdir 202
linux3[2]%
```

If you make a mistake in naming your folder, you have two options. You can either remove that directory or you can rename the directory. For example, if you accidentally created a folder called "**`2002`**" you would remove it by typing the command **`rmdir 2002`** and hit enter. Be very careful that you do not remove the wrong directory. There is no recycle bin or "undo" button.

The safer option would be to rename the directory. To rename the directory, you use the "move" command, or **`mv`**. To use it, you would type **`mv 2002 202`**, and hit enter.

```
linux3[2]% mv 2002 202
linux3[3]%
```

The **mv** command would rename our folder from "**2001**" to "**202**" instead. (As another example, if we wanted to rename it from "**2021**" to "**202**" we would use the command **mv 2021 202** to do so.)

Inside of your 202 folder, create two additional folders – labs and projects.

```
linux3[2]% mkdir labs
linux3[2]% mkdir projects
linux3[3]%
```

Inside of your labs folder, create an additional folder for lab0.

```
linux3[2]% cd labs
linux3[2]% mkdir lab0
linux3[3]%
```

You can type **ls** to make sure the folder has been created.

```
linux3[9]% ls
lab0
linux3[10]%
```

You can type pwd to see where you are in the directory structure.

The home directory

```
linux3[9]% pwd
/afs/umbc.edu/users/j/d/jdixon/home/202
linux3[12]%
```

Your user specific folder

E. Additional GL Commands

There are several flavors of Unix: IRIX, Solaris, Linux, etc. For this class, your programs must compile and run on Linux. When you log onto GL, be sure that you connect to the host named gl.umbc.edu or linux.gl.umbc.edu. Connecting to these host names ensures that you are actually working on one of the six Linux servers, linux1.gl.umbc.edu, linux2.gl.umbc.edu, or linux3.gl.umbc.edu, linux4.gl.umbc.edu, linux5.gl.umbc.edu, and linux6.gl.umbc.edu; which of the six Linux servers you are connected to depends on the current load for each machine.

Command	Function
ls	Lists the files in the current directory **ls -al** gives more information about files. -a includes hidden files -l stands for "long" output.
cp	Copies a file. **cp sample.cpp sample2.cpp** makes a copy of sample.cpp and names the new copy sample2.cpp. The file sample.cpp is unchanged.
mv	Renames ("moves") a file. **mv average.cpp mean.cpp** changes the name of the file from average.cpp to mean.cpp. The file average.cpp no longer exists.
rm	Removes or deletes a file. **rm olddata.dat** deletes the file olddata.dat. Be very careful using this command – there is no undo command on GL!
more	Displays the contents of a file on the screen one page at a time **more example.txt** shows the contents of the file example.txt one screenful at a time. You must press the spacebar to advance to the next page. You may type q to quit or b to go back to the beginning of the file.
mkdir	Makes a new subdirectory in the current directory. **mkdir cmsc202** will create a new directory called cmsc202 in the current directory.
rmdir	Removes an empty subdirectory from the current directory.

Simple UNIX Commands (Use for the GL command line)

F. Basic Emacs Commands

G. Starting Emacs

Emacs is flexible and widely-used text editor which is supported on the GL systems. To start the text editor, type the command emacs at the Linux command prompt.

You may also specify a file to open or create on the command line. For example, the command `emacs example.cpp` starts Emacs and loads the file example.cpp, if it exists; if the file does not exist, it will be created when the buffer is saved (e.g. with the Emacs command C-x C-s; see below).

H. Emacs Commands

Emacs commands are entered using key combinations that include the Control or Escape keys. A quick reference to major commands is provided below.

The letter C represents the Control key; the command C-n means "hold the Control key and press n."

The letter M stands for "Meta," which is the Escape key on most systems. For example, the command M-a means "press the Escape key, then press the letter a." *Do not hold down the Escape key for Meta commands.*

File Operations

Command	Result
C-x C-f	Retrieve or open a file
C-x C-s	Save the current file and continue editing
C-x C-c	Save the current file and exit
C-x C-w	Write the buffer contents to a file

Cursor Movement

Command	Result
C-n	Move to the next line
C-p	Move to the previous line

C-b	Move backward one character
C-f	Move forward one character
C-u n C-f	Move forward `n' characters (n should be a number)
C-a	Move to the beginning of the line
C-e	Move to the end of the line
M-f	Move forward one word
M-b	Move backward one word
M-a	Move to the beginning of the sentence
M-e	Move to the end of the sentence
C-v	Move forward one screenful
M-v	Move backward one screenful
M-<	Move to the beginning of the file
M->	Move to the end of the file
M-g	Goto line (prompted for line number)

Editing

Command	Result
C-d	Delete one character
C-k	Kill (cut) from the cursor position to end of line
M-k	Kill (cut) to the end of the current sentence
M-d	Kill (cut) the next word after the cursor
C-y	Yank (paste) the last killed text back

Miscellaneous

Command	Result
C-g	Cancel current command
C-l	Clear screen and redisplay everything, centering the screen on the current cursor position
C-x u	Undoes one command's worth of changes

I. Creating Your First C++ Program

You are going to create your first C++ program. At the command line type **`emacs lab1a.cpp`** to start Emacs. The file lab1a.cpp should not exist yet, so it will be created the first time you save your file. You do not need to turn in **`lab1a.cpp`**.

The program is a simple "Hello, world!" Here is how the output should appear when the program is run:

```
linux3[17]% ./lab1a
Hello, world!
bash-4.1$
```

Note that **`bash-4.1$`** is the Linux shell prompt, not part of the program.

A program template is given below; you will need to add one line to write the message "Hello, world!" You can write to the screen using **`cout`** and the insertion operator (**`<<`**). For example, the following line writes "I love CMSC 202" to the screen:

```
cout << "I love CMSC 202" << endl;
```

A. Program Template

Below is sample code that you can use for your lab submission.

```
#include <iostream>
using namespace std;

int main() {

   // Insert your code here

   return 0;
}
```

Once you have typed your code in Emacs, be sure to save (**`C-x C-s`**) or save and quit (**`C-x C-c`**).

B. Compiling and Running

To compile your program, enter the following command at the Linux prompt:

```
g++ -Wall lab1a.cpp -o lab1a
```

This command runs the GNU C++ compiler (**g++**). The option **-Wall** instructs the compiler to be verbose in its production of warning messages; the option **-o lab1a** (hyphen followed by the letter "o", not the digit zero), instructs the compiler to give the executable program the name **lab1a**. If the program compiles correctly, the executable file **lab1a** will be created in the current directory.

At the Linux prompt, enter the command **./lab1a** to run your program. It should look like the sample output provided above. You do not need to submit this part of the assignment. You only need to submit lab1.cpp to Blackboard (which is below).

C. MakeFiles

The Unix **make** utility can be used to automate the compilation and linking of programs. It may be difficult to see how this could be useful at this point, but you will appreciate make's functionality when you work on complex, multi-file projects later in the semester. For now, we will learn how to use make for simple program builds.

make requires that the programmer provide instructions to build the program in a text file called a *makefile*. The makefile file will typically be named **Makefile** with a capital "M"; the **make** program looks for this file by default.

A makefile consists of multiple sections, called *rules*, that specify how the program is built. Each rule consists of a header line, listing a specific item to build (called the *target*) and the files the target depends upon (the *prerequisites*), followed by zero or more lines specifying the commands necessary to construct the target (the *build actions* or *recipe*). The target is typically a file name, such as the name of an executable program. **make** compares the last-modified timestamp of the target with those of the prerequisites it depends on, and if any of the prerequisites is newer, it will execute the recipe to bring the target up-to-date.

Without further ado, here are the contents of a very simple makefile:

```
program: program.cpp
		g++ -Wall program.cpp -o program
```

In this example, **program** is the target, **program.cpp** is the only prerequisite, and the recipe is **g++ -Wall program.cpp -o program**. When **make** is run using this Makefile, it will check to see if **program.cpp** has been modified more recently than **program** and, if so, will execute the recipe to recompile the program.

There is a shortcut if you are "making" a single .cpp file. If the file is named **lab1a.cpp** then you can make it simply by typing **make lab1a**
NOTE: Be very careful – if you accidentally compile over your lab1a.cpp file, it will be gone forever!

Note: the whitespace at the beginning of a recipe *must* be a tab. Emacs will insert a tab correctly so long as your makefile is named **Makefile** – Emacs knows not to replace tabs with spaces when working on makefiles.
Now, create a makefile (named **Makefile**) to build your C++ program **lab1a.cpp**. Create the makefile in Emacs, entering the lines from the sample, modified appropriately. Once you have created the makefile, save and quit Emacs. Test the makefile by entering the following commands at the Linux prompt:

```
touch lab1a.cpp
make
```

The **touch** command changes the last-modified time of **lab1a.cpp** to the current time, ensuring that **lab1a.cpp** is "newer" than the executable **lab1a**; thus the **make** command will execute the recipe to rebuild the executable.

D. Wrapping Up Part 1

Watch this video (https://www.youtube.com/watch?v=ci42iOOtTRI) for an explanation of the “life cycle” of remote programming on GL. This is the same for 201 and 202.

Continue to Part 2 – Practicing Input

Part 2 – Practicing Input

I. Getting Input

In C++, we can read in a single variable the way we learned in class:

```
#include <iostream>
using namespace std;

int main() {
  int myVar;
  cout << "Please enter a variable:";
  cin >> myVar;
}
```

This will read from standard in and store the result into **`cout`**. The program will stop reading once it hits whitespace.

We can also read an entire line with the **`getline()`** command. The **`getline()`** method takes two arguments--the variable we are going to store into, and the length of the string.

In the example below, we are going to read in data and store it into a c-string. A c-string is an array that is designed to hold characters. An array is a simple structure that holds similar things. For example, an array of integers could hold many integers, or an array of doubles could hold many doubles. In this case, we are holding many characters.

Notice that we declared a c-string first with the line **`char line[80];`**

```
using namespace std;

int main() {
  char name[80];
  cout << "Enter a name: ";
  cin.getline(name, sizeof(name));
  cout << "Your name
  cout << "The variable 'name' is " << sizeof(name) <<
```

```
          " characters long." << endl;
  return 0;
}
```

There is another small function that is being used in this example. It is called **`sizeof()`**. **`Sizeof()`** can take in: 1. An expression (in this case, line). If you use **`sizeof(*variable*)`** it will return how many characters are in that variable. As each character is 1 byte, it shows us how many characters are in the line. 2. A data type (like **int** or **double**). If you use **`sizeof(*data type*)`** it will return how many bytes that data type uses.

Here are a couple of examples of the output using a single word name and a first and last name (with a space).

```
-bash-4.1$ ./lab1a
Enter a name: Mike
Your name is Mike
The variable 'name' is 80 characters long.
-bash-4.1$ ./lab1a
Enter a name: Mike Doe
Your name is Mike Doe
The variable 'name' is 80 characters long.
-bash-4.1$
```

Notice how both had the same **`sizeof()`**. That is because the c-string was set to hold 80 characters (and it doesn't return the size of the c-string itself).

II. Problems with `getline()`

When mixing **`getline()`** and other input methods, there are a few things to consider. **`getline()`** reads and discards the ending **`'\n'`** character, but the other input methods do not--they stop just before it. They do skip over **`'\n'`** as leading whitespace, though. This can lead to unexpected results.
For example:

```
#include <iostream>
using namespace std;
int main () {
  int age;
  char name1[80];
  char name2[80];
```

```
  cout << "Enter your age: " << endl;
  cin >> age;
  cout << "Enter your first name: " << endl;
  cin.getline(name1, 80);
  cout << "Enter your last name: " << endl;
  cin.getline(name2, 80);
  cout << "age = " << age << endl;
  cout << "first name = " << name1 << endl;
  cout << "last name = " << name2 << endl;
  return 0;
}
```

If we run this code, notice what happens:

```
 bash-4.1$ ./lab1
Enter your age:
10
Enter your first name:
Enter your last name:
Doe
age = 10
first name =
last name = Doe
-bash-4.1$
```

What? That isn't what we wanted! We wanted it to get the age and then the first name and the last name. It skipped the first name completely. That is because it brought the newline character from the age and then subsequently stripped it off. There are a few ways to fix this. First, we can add an additional **`getline()`** so that it will handle the additional newline character. Second, we can add a command that will remove any additional data in the string by using the **`ignore()`** command.

If we want to fix the input using the **`ignore()`** command we can add a couple of lines of code. Here is the code that we can add:

```
if (cin.peek() == '\n')
    cin.ignore();
```

This code adds two additional functions from **`cin`**. The first function is called **`cin.peek()`** and it returns the next character in the input sequence, without

extracting it. This means that it doesn't actually change the input stream but just checks what the next character is. In this case, if the next character is a newline character (`'\n'`) then it runs the **`cin.ignore()`** function. **`cin.ignore()`**, when it isn't given any parameters, will just ignore anything waiting in the input stream (or buffer).

Here is the complete code with the updated peek and ignore functions:

```
#include <iostream>
using namespace std;

int main() {
  int age;
  char name1[80];
  char name2[80];
  cout << "Enter your age: " << endl;
  cin >> age;
  if (cin.peek() == '\n')
    cin.ignore();
  cout << "Enter your first name: " << endl;
  cin.getline(name1, 80);
  cout << "Enter your last name: " << endl;
  cin.getline(name2, 80);
  cout << "age = " << age << endl;
  cout << "first name = " << name1 << endl;
  cout << "last name = " << name2 << endl;
  return 0;
}
```

The new output would look like this:

```
-bash-4.1$ ./lab1
Enter your age:
41
Enter your first name:
Mike
Enter your last name:
Doe
age = 41
first name = Mike
```

```
last name = Doe
-bash-4.1
```

Hooray! It worked.

III. Getline with Strings

Now that we have looked at using **`getline`** with c-strings, let's take a quick minute and look at **`getline`** with strings. As a reminder, if we want to use strings, we need to use #include <string>.
If we wanted to ask the user for their first and last names, we could use this code:

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  string name;
  cout << "Enter your first and last names: " << endl;
  cin >> name;
  cout << "Your name is: " << name << endl;
  return 0;
}
```

The problem is, when we run the code, the output looks like this:

```
Enter your first and last names:
Mike Doe
Your name is: Mike
-bash-4.
```

As you notice, the "Doe" last name was omitted from the output. That is because the **`cin`** command is only designed to hold up to the first whitespace that it encounters. So, after it inputs the name "Mike", it drops the last name completely. Even scarier, the last name is not actually gone! It is waiting for another **`cin`** statement to use the last name –it is being held until another **`cin`** statement calls it.
In order to read in both the first name and the last name, we need to use the **`getline()`** command. Let's try this again using a **`getline()`.**

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  string name;
  cout << "Enter your first and last names: " << endl;
  getline(cin, name);
  cout << "Your name is: " << name << endl;
  return 0;
}
```

And when it runs, it looks like this:

```
#-bash-4.1$ ./lab1c
Enter your first and last names:
Jeremy Dixon
Your name is: Jeremy Dixon
-bash-4.
```

The only thing a little bit different is the syntax for the **`getline()`** command when you are using a string rather than a c-string. Otherwise, we have the exact same pitfall as above (where we need to use the **`cin.peek()`** and **`cin.ignore()`** functions).

The **`cin.peek()`** and **`cin.ignore()`** functions can be used in multiple ways. Consider what might be done, for example, if the user wants the program to have a "basic" function that checks whether the input variable is an integer. Likewise, what if the user input is a combination of alphanumeric and special characters? The code below illustrates these possibilities:

```
while(cin.fail()){
  cout << "Please enter your age, it should be an
integer: " << endl;
  cin.clear();
  cin.ignore(256, '\n');
  cin >> age;
```

```
}
```

In step one, **`cin.fail()`** determines whether the current input variable matches the declared data type. If the input does not match the declared date type, **`cin.clear()`** function then clears the input. After this, the **`ignore(256, '\n')`** function is used to skip the specific number of characters until a specified breakpoint or delimiter is reached.

The preceding is an effective method, but there are significant limitations, the most important of which is that the approach is not very adaptable, reliable, or efficient. For example, it can only identify three "wrong" situations related to integer input:

Situation 1: The input is full of symbols, including punctuation and special characters;

Situation 2: The input contains only characters;

Situation 3: The input contains only characters and symbols.

Such limitations mean that inputting numbers along with characters and symbols leads to a coding error, as shown below:

```
#-bash-4.1$ ./lab1d
Enter your age:
99Thanos
Enter your name:
age = 99
name = Thanos
```

The **`cin.fail()`** function is unable to recognize, for example, the input of alphabetical characters combined with numbers, and would therefore consider the input valid. This means that, under this method, were the user to input "**`99Thanos`**" and then hit the enter key, the **`cin.fail()`** function would result in the word and number being output separately, with the number parsed erroneously as the user's age.

The error occurs because C++ reads the first part "**`99`**" as an integer of the previously declared data type. Then it reads the second part, "**`Thanos`**", and

simply categorizes it as a special case in the buffer, meaning that it is automatically placed in the next input field, "**`Enter your name`**". Though some programmers may argue that this flaw in the code should be allowed to remain, such a course of action isn't recommended, as this code could easily lead to numerous functionality issues for the user. We will explore additional ways to correct this later in the semester.

IV. Programming Assignment

- For this lab, we are going to practice a few different aspects of programming in C++.
- Practice using **`cin`** and **`cout`**.
- Practice using variables
- Practice with loops and validation

For this lab, you are going to create the code named **`lab1.cpp`** that asks the user for their cat's name (storing it as a c-string) and the cat's age (as an integer). The user can then do one of three things: pet the cat, feed the cat, or chase the cat. The most important thing is that if the user enters invalid values for the age or the menu, it will prompt the user again. After any three choices, the program should exit. You can see the sample output here:

```
[jdixon@linux1 lab1]$ ./lab1
What is your cat's name?
Kitty
How old is your cat?
55
Please enter your cat's age, it should be an integer
between 0 and 22
3
Cat Name = Kitty
Cat's age = 3

What would you like to do?
1. Pet Kitty
2. Feed Kitty
3. Chase Kitty
Enter your choice:
Attack Kitty
```

```
Enter your choice (1-3):
1
You pet Kitty and they purr.
What would you like to do?
1. Pet Kitty
2. Feed Kitty
3. Chase Kitty
Enter your choice:
Feed Kitty
Enter your choice (1-3):
3
You chase after Kitty and they hiss and try to bite.
What would you like to do?
1. Pet Kitty
2. Feed Kitty
3. Chase Kitty
Enter your choice:
2
Kitty meows gently as you feed them.
That is enough for today. Good-bye
```

Please note: mess around with the data types for the input and the average variable so you can see how it affects the output. A “while loop” or “do while loop” is the best method to use to check the validity of input. Below is an example:

```
#include <iostream>
using namespace std;

int main()
{
  // Starts to declare and initialize an int data
type variable
  int a = 99;
  // As long as a is less than 120, keep increment
by 1 each time
  while (a < 120)
    {
      a += 1;
    }
  // Display the output of a after the while loop
  cout << "What is a now: " << a << endl;

  return 0;
```

An issue that can arise with just declaring a variable is what old data may exist in that memory space. After we declare a variable, we set its initial value (called initialization). This is to ensure that the data in that memory location is valid. So, for this, we may want to initialize a variable to 0 as in **`int myInt = 0;`**

If you have problems with this, please email your lab TA/TF which is on Blackboard under “Lab Sections” or below.

Section	Faculty	Day	Time	TA/TF	Email Addresses
10	Dixon,Jeremy A	T/TH	10:00AM-11:15AM		
11		T	11:30AM-12:20PM	Kiran Jambhale	kjambha1@umbc.edu
12		TH	11:30AM-12:20PM	Kiran Jambhale	kjambha1@umbc.edu
13		T	1:00PM-1:50PM	Poorvi Harsora	poorvih1@umbc.edu
14		TH	1:00PM-1:50PM	Poorvi Harsora	poorvih1@umbc.edu
15		TH	8:00PM-8:50PM	Akarsh Kashamshetty	akarshk1@umbc.edu
20	Robinson, Dawn	MW	5:30PM-6:45PM		
21		M	7:00PM-7:50PM	Akshay Reddy Akkati	aakkati1@umbc.edu
22		M	4:00PM-4:50PM	Nonso Ugwu	chiugwu1@umbc.edu
23		M	1:00PM-1:50PM	Nonso Ugwu	chiugwu1@umbc.edu
24		M	3:00PM-3:50PM	Nonso Ugwu	chiugwu1@umbc.edu
25		M	10:00AM-10:50AM	Jake Miu	jmiu1@umbc.edu
30	Joeg,Prasanna	T/TH	2:30PM-3:45PM		
31		T	5:30PM-6:20PM	Aristidis Dernelakis	ad20@umbc.edu
32		TH	5:30PM-6:20PM	Eddie Nieberding	edwardn3@umbc.edu
33		T	8:00PM-8:50PM	Hamza Asad	sasad1@umbc.edu
34		TH	10:00AM-10:50AM	Kiran Jambhale	kjambha1@umbc.edu
35		TH	1:00PM-1:50PM	Akarsh Kashamshetty	akarshk1@umbc.edu
40	Joeg,Prasanna	T/TH	4:00PM-5:15PM		
41		T	10:00AM-10:50AM	Kiran Jambhale	kjambha1@umbc.edu
42		TH	2:30PM-3:20PM	Akarsh Kashamshetty	akarshk1@umbc.edu
43		T	2:30PM-3:20PM	Daniel Milani	dmilani1@umbc.edu
44		TH	1:00PM-1:50PM	Poorvi Harsora	poorvih1@umbc.edu
45		T	1:00PM-1:50PM	Poorvi Harsora	poorvih1@umbc.edu
50	Dixon,Jeremy A	MW	1:00PM-2:15PM		
51		M	7:10PM-8:00PM	Akshay Reddy Akkati	aakkati1@umbc.edu
52		W	7:10PM-8:00PM	Nonso Ugwu	chiugwu1@umbc.edu
53		W	2:30PM-3:20PM	Akshay Reddy Akkati	aakkati1@umbc.edu
54		W	2:30PM-3:20PM	Akshay Reddy Akkati	aakkati1@umbc.edu
55		W	4:00PM-4:50PM	Akarsh Kashamshetty	akarshk1@umbc.edu

2. Compiling and Running

To compile your program, enter the following command at the Linux prompt:

```
g++ -Wall lab1.cpp -o lab1 or
make lab1
```

This command runs the GNU C++ compiler (**`g++`**). The option **`-Wall`** instructs the compiler to be verbose in its production of warning messages; the option **`-o lab1`** (hyphen followed by the letter "o", not the digit zero), instructs the compiler to give the executable program the name **`lab1`**. If the program compiles correctly, the executable file **`lab1`** will be created in the current directory.

At the Linux prompt, enter the command **`./lab1`** to run your program. It should look like the sample output provided above.

3. Completing your Lab

Submitting your lab has two steps:

1. Set up a symbolic link in your home directory to your submission folder. You will only do this ONCE for the entire semester. Execute these commands:

   a. **`cd ~`**

   b. For the next command, copy it exactly – you should not need to modify it at all (**`$USER`** will automatically populate your username on GL). Also, notice the space after **`$USER`** and before **`~/cs202proj`**.

   ```
   ln -s /afs/umbc.edu/users/j/d/jdixon/pub/cmsc202/$USER
   ~/cs202proj
   ```

   c. To check that the symbolic link was built successfully, you can type:

      i. **`ls ~/cs202proj`**

      ii. **`It should list proj1, proj1-late1, proj1-late2 through proj5-late2 and lab1`**

2. Copy the lab files into your lab1 folder. Execute these commands:

   a. cd to your lab1 folder. An example might be:

   `cd ~/202/labs/lab1`

   b. **`cp lab1.cpp ~/cs202proj/lab1`**

You can check to make sure that your files were successfully copied over to the submission directory by entering the command

`ls ~/cs202proj/lab1`

You can check that your program compiles and runs in the **`lab1`** directory, but please clean up any **`.o`** and executable files. Again, do not develop your code in this directory and you should not have the only copy of your program here.