Name:

Student ID: *Lab Instructor:*   Borja Sotomayor

	Do not write in this area			
1	2	3		**TOTAL**

Maximum possible points: 20 + 2

NOTE: The lab session will take place in the Linux machines of the Maclab. This lab assumes that you know your way around a UNIX system. If you don't, please read the addendum "Getting Acquainted". However, take into account that you are free to do the lab exercises (and homework assignments) on Mac machines.

Compiling C/C++ programs

[This part of the lab will not be graded. Its purpose is to help you get acquainted with the steps involved in compiling a C/C++ program]

The compiler we will use in this course is the GCC compiler (http://gcc.gnu.org/) , a part of the GNU project (http://www.gnu.org/). To get acquainted with this compiler, we will start by writing, compiling, and running the following simple program seen in class:

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

int main()
{
    cout << "Hello, world!" << endl;
}
```

Using your favorite editor, type the above code and save it in a file called *helloworld.cpp*. We suggest you start creating an organized directory structure in which to place your lab files. For example, inside your home directory create a *cmsc15200* directory, then a *labs* directory inside it and, finally, a *lab01* directory. So, your *helloworld.cpp* file would have the following path:

~/cmsc15200/labs/lab01/helloworld.cpp

The University of Chicago
Department of
Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2005
Lab #1 (07/27/2005)

Now, we will compile the program using GCC. In particular, we will use the g++ command (the C++ compiler). We must run g++ like so:

```
g++ helloworld.cpp -o helloworld
```

When running g++ notice how we specify the following:

➢ The name of the file we want to compile: helloworld.cpp

➢ The name of the executable we want to create. This is done using the optional "-o" argument. If we don't specify an executable name, the default would be used ("a.out").

Now, try to run the program. To do this, simply type the name of the executable file in the command line:

```
./helloworld
```

You should see the following:

```
Hello, world!
```

Now, repeat the above steps for the following program (save this file as *convert.cpp*):

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

int main()
{
    float miles, km;
    cout << "Type number of miles: ";
    cin >> miles;
    km = miles * 1.61;
    cout << "Kilometers: " << km << endl;
}
```

Exercise 1 <<5 points>>

In an effort to improve relations with metric system countries, the US Government has embarked itself on an ambitious project to develop software that converts temperatures from the Fahrenheit scale to the Celsius scale. You must write a C++ program that asks the user for a Fahrenheit temperature, and then outputs the equivalent Celsius temperature. Hint: $C = (F - 32) \cdot \frac{5}{9}$ Hint 2: Here's a couple of values you can use to test your application: 0ºF = -17.78ºC , 32ºF = 0ºC , 99.5ºF = 37.5ºC

```
Please enter a temperature in Fahrenheit: 32
32 F = 0 C
```

Exercise 2 <<10 points>>

To disprove the widespread belief that computers are unable to reason, we have been asked to develop an application capable of asking the user a set of questions and then articulating an intelligent response. Our application must ask the user the following questions:

> ➢ What is your age?
> ➢ What is your GPA (Grade Point Average)?

If the age of the user is less than or equal to 14, then the program must output the following:

```
You're too young to have a GPA!
```

If the user's age is between 15 and 18, and the GPA is greater than 3.8, then the program must output the following:

```
Wow, you must be gifted!
```

In all other cases, the program must output:

```
I have nothing to say about you!
```

For full credit, your program should check that the data introduced by the user is valid. In other words, the GPA must be a number between 0.0 and 4.0, and the age must be greater than 0.

Exercise 3 <<5 points>>

After conducting several hearings in the US Senate, the US Government has come to the conclusion that they would ideally like a program that can convert more than one temperature in a single run. Modify the program from exercise 1 so that after making a conversion, the user will be asked if he wants to perform another conversion. If the user answers "Yes", then the user is asked for another Fahrenheit temperature. If the user answer "No", the program will exit.

```
Please enter a temperature in Fahrenheit: 0
0 F = -17.78 C

Would you like to enter another temperature? Y

Please enter a temperature in Fahrenheit: 32
0 F = 0 C

Would you like to enter another temperature? Y

Please enter a temperature in Fahrenheit: 99.5
0 F = 37.5 C

Would you like to enter another temperature? N
```

Hint: As we have not seen string manipulation, you should ask the user for a single character ('Y' or 'N').

Extra credit (2 points): When asking if the user wants to perform another conversion, check that the character is 'Y', 'y', 'N', or 'n'. If the user introduces any other character, show an error message and ask him again whether he would like to perform another conversion.

The University of Chicago
Department of
Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2005
Lab #1 (07/27/2005)

Addendum: Getting Acquainted

The purpose of this addendum is to provide you with the bare minimum knowledge necessary to write and compile a C++ program on a UNIX system. However, working with a UNIX system is well outside the scope of this course, so pointers to more complete documents are provided as well. If you are completely new to UNIX systems, we encourage you to read these documents and consult with your lab instructor if you have any questions.

Interacting with a UNIX system

UNIX systems in general, and Linux distributions in particular, usually provide users with two interfaces:

- Command-line interface (or *shell*): Allows the user to interact with the system through the use of commands which you must type in using the keyboard. There are many different types of shells, such as BASH, CSH, TCSH, ...

- Graphical interface: Allows the user to interact with the system through the use of graphical elements such as windows, buttons, text fields, etc. mainly using the mouse. Although the taxonomy of graphical interfaces in UNIX can be a bit complex, most new users will generally interact at first with high-level desktop environments such as KDE and GNOME, which are similar in many respects to graphical interfaces in Windows and Mac systems.

If you are new to UNIX, we encourage you to use the KDE desktop environment, which is very intuitive and easy to use, specially if you have previous experience in Windows systems. To use KDE, make sure you choose a "KDE Session" before logging in (you can do this clicking on the "Session" button in the login screen).

However, there will be times when using a shell will be unavoidable (for example, when compiling a program). You can bring up a console in KDE by choosing "Terminal console" in the System menu (inside the KDE menu, similar to the Windows "Start" menu).

Text editing

Text editing can be performed using text-interface programs (from a shell) or graphical programs (from a graphical interface). You are free to use any text editor you want during the course. Graphical editors (such as kedit, kate, gedit, ...) are easy to use, but generally lack C/C++ functionality. Text-based editors (vi and emacs) are powerful and

versatile, but have a greater learning curve. If you want to learn more about vi and emacs, you can find several interesting links here:

http://people.cs.uchicago.edu/~kaharris/cs152/#TextEdit

The UNIX file system

The UNIX file system, like most modern file systems, provides users a layer of abstraction over the data contained in storage mediums such as hard disks. In particular, file systems allow us to think in terms of *files* and *directories*. Of special interest in the UNIX file system is the *home directory*, which is where we will be able to place our files and work with them. For the purposes of this course, we will be unconcerned with other directories in the file system.

The UNIX shell

As mentioned above, the shell will allow us to interact with the system through the use of commands which you must type in using the keyboard. When we bring up a shell (either because we are directly using a pure command-line interface or because we have called one up from a graphical interface), we will be shown a *prompt* where we can type in a command. In particular, our prompt will show something like this:

```
user@machine:~$
```

For example:

```
borja@mahogany:~$
```

This denotes that the current user is *borja*, logged into machine *mahogany*, and with the *current directory* being the home directory (the tilde character ~ is short for the home directory). The current directory is an important concept in the shell, as we will generally refer to files *relative* to the current directory (this will be explained shortly).

To start tinkering with the command-line, we can run a simple command called *fortune* that will present us with a fortune message (akin to the ones found in fortune cookies). Simply type *fortune* and press enter to run the command.

```
user@machine:~$ fortune
```

You should then see a message printed out. For example:

```
Sometimes a cigar is just a cigar.
            -- Sigmund Freud
```

We can invoke literally hundreds of commands from the command-line. For this course we will only need a handful of commands which are explained below. For a more complete text on the command-line, take a look at the following links:

http://support.uchicago.edu/docs/misc/unix/tutorial/

http://people.cs.uchicago.edu/~kaharris/cs152/index.html#IntroUnix

Creating new directories

We can create new directories using the *mkdir* command. For example, suppose we want to create a *cmsc15200* directory inside our home directory. We will type the following command:

```
~$ mkdir cmsc15200
```

● For simplicity, we are omitting the "user@machine" part.

● The *mkdir* command, unlike the *fortune* command, accepts an *argument*. In particular, ther argument tells it exactly what directory to create. In general, arguments are used to pass options to the commands.

Listing files

We can see a listing of files and directories contained in the current directory by running the *ls* command.

```
~$ ls
```

You should see the following single line:

```
cmsc15200
```

The current directory

Previously, we introduced the concept of *current directory*, and said that we will generally refer to files *relative* to the current directory. This is important because there are many commands that require us to specify a file (e.g. when compiling a program, we need to specify what C++ source file we want to compile).

The University of Chicago
Department of
Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2005
Lab #1 (07/27/2005)

For example, if the current directory is the home directory and the home directory contains a file named *foo.cpp*, we would refer to it from the command line simply like this:

```
foo.cpp
```

However, if the file were inside the *cmsc15200* directory, we would refer to it like this:

```
cmsc15200/foo.cpp
```

Another example: the *ls* command seen above (without any arguments) assumes that we want to see the list of files and directories contained in the *current* directory.

To change the current directory, we need to use the *cd* command, specifying the new current directory. For example, suppose we are in the home directory and want to make *cmsc15200* the current directory. We would run *cd* like so:

```
~$ cd cmsc15200
```

The prompt would change to reflect that the current directory has changed:

```
~/cmsc15200$
```