1 Homework 1, CSCE 240, Fall 2018

1.1 Points

This is a 10-point assignment.

1.2 Overview

This assignment's only purpose is to verify that you can run programs as you will need to run them in order to get full marks in this class.

This document will go over the *process* of doing a programming assignment for this course, including the Linux stuff.

COMMENT: I will use the terms "Unix" and "Linux" interchangeably. At the level we will be working, there won't be a difference.

This assignment description will cover most of the Linux stuff you will need for this course, and it isn't much, because I am a minimalist. If you don't follow all the details, go look them up in the 215 text or online. This stuff isn't hard, there isn't that much of it, and you will be much better off if you learn it.

1.3 The Test Directory

For every assignment, you will get a "test" directory. This first is named

```
hwprog1testdirectory
```

and the others will be named analogously.

Inside this directory will be several files and directories. My current version reads as follows if I do `ls -l` in the directory. (At least, this is what I got some time ago when I did the `ls -l`. If I were to do it today, I would get something a little different, but not so much different that you would not recognize what's there.) Note that I am not following the standard Unix convention of not having filename extensions. That's because if I do happen to put a `.txt` extension on a filename, then I can use some of the Mac utility programs to open and look at those files. If I took off that extension, I would have to do some other nonsense to be able to open those files automagically with a double-click.

```
-rwx--x--x   1 dabuell  staff   1103 May 14  2016 README.txt
drwxrwxrwx  11 dabuell  staff    374 May 10  2016 Utilities
-rw-r--r--   1 dabuell  staff  61168 Aug  4  2016 hw1.pdf
-rwxr-xr-x   1 dabuell  staff    150 Jun  6  2016 makefile
drwx--x--x   8 dabuell  staff    272 Oct  1  2016 mydirectory
-rwx--x--x   1 dabuell  staff    710 May 14  2016 zaZipUpScript.txt
-rwx--x--x   1 dabuell  staff    518 May 14  2016 zbFileCopyScript.txt
-rwx--x--x   1 dabuell  staff    296 May 14  2016 zcCompileScript.txt
-rwx--x--x   1 dabuell  staff    258 May 14  2016 zdExecuteScript.txt
```

1.4 The README

The `README.txt` is exactly what it is—a file describing what's here. This is the Unix/Linux convention for a file you should read to understand what's going on.

1.5 Utilities code

The `Utilities` directory has three sets of utility functions you will probably want to use. These are

- `utils.h` and `utils.cc`, which have boilerplate functions for doing things like opening files and then crashing with a useful error message, and so forth.

- `scanner.h` and `scanner.cc`, which provides a C++ version of the Java `Scanner` class. Input/output in C++ is rather annoying and painful when done in the raw, and this class provides the functionality of the Java `next`, `nextdouble`, `nextint`, and `nextline` functions. These functions read "tokens" one at a time from input, with white space like blank spaces, newline characters, etc., being the way the function determines the end of a token, cleverly detecting that multiple blank spaces are treated as if they were only one, etc. For some reason, the "tokenizing" function that would provide something like the `next` of Java is a deprecated function in C++. So I wrote an equivalent.

- `scanline.h` and `scanline.cc`. In Java, if you pass an open file to `Scanner`, it reads from the opened file. If you pass a `String` to `Scanner`, it reads from that string as if it were from a file. It seems hard to create one class in C++ that does the same thing, so I wrote `scanner.cc` to take a C++ `string` as an argument and treat that as a filename to be opened, and `scanline.cc` to take a C++ `string` as an argument and treat that as a string of characters to be scanned.

These two together permit the following input structure.

```
while not end-of-file
    thisline = scanner.nextline reads an entire line of input
    while not end-of-line
        token = scanline(thisline) reads token by token
```

What's the point? The point is that you can read input lines with a variable number of tokens, one line at a time, until you are out of input data.

COMMENT: The `Utilities` directory will always be in the same relative place in all your test directories. This is important, because it means that the lines of code that tell the compiler where to find the code don't have to change from one program to the next.

That is, you have a directory hierarchy of

```
hwprog1testdirectory (grandparent)
  Utilities (aunt)
  mydirectory (parent)
    hackerjrandom_hw1 (your directory)
      your code files
```

so all your code files in `hackerjrandom_hw1` can refer to `../../Utilities` and the path will not need to be changed. I have tried to standardize this for you.

COMMENT: You do NOT for any assignment in this class have to document the `Utilities` code in any way. You may treat this as if it were already-compiled code and not source code that you include with your own code for compilation. You do not need to put your name in header files or the implementation files.

1.6 The assignment itself

`hw1.pdf`: You should always have the programming assignment txt itself inside the test directory.

1.7 Makefiles

`makefile`: The convention in Unix is that you have a `makefile` for compiling and doing other things with files and programs. If you have forgotten how they work, go read up on them. I don't use complicated makefiles, so this one has nearly all of what you will see this semester.

```
GPP = g++ -O3 -Wall -std=c++11

M = main.o

Aprog: $(M)
    $(GPP) -o Aprog $(M)

main.o: main.cc
    $(GPP) -o main.o -c main.cc

clean:
    rm Aprog
    rm *.o
```

The convention is that symbols for `make` are in all caps. The `GPP` line defines the symbol `GPP` to be the command line command `g++ -O3 -Wall -std=c++11`, namely,

- Compile using the `g++` compiler, using optimization level 3 for full performance optimization.
- Have the compiler generate all possible warnings by using the `-Wall` option.
- Have the compiler include all the 2011 C++ functions with the `-std=c++11` option.

  NOTE: You get the 2011 stuff by default on a Mac with the standard Mac compiler. You don't get the 2011 stuff by default on the Linux machines in the lab. (Or at least you didn't get it by default last time I checked; things may have changed.) I have no idea what happens on Windows.

  It turns out not to cause problems if you ask for it, as above, and it turns out it's already the default.

The `M = main.o` line says that I want the symbol `M` to have the value `main.o` which will come later on.

The lines in a `makefile` that have a symbol followed by a colon are directives to the `make` program. In this case, `make` reads the next two lines as follows.

I need the most up-to-date version possible of a file named `Aprog`. That file depends on the most up-to-date version of `M`, which we defined to be `main.o`. If there is, or should be a new version of `Aprog` (because something was edited that affects `main.o` and thus affects `Aprog`), then execute the next line of the `makefile` to create an up-to-date version of `Aprog`.

That line says to run the `GPP` command, which is a command to compile some code. The option `-o Aprog` says to name the file that is the output of the compilation with the name `Aprog`. Finally (the thing on which all the action is to take place is always the last argument on the line), this step is to take place on the most up-to-date possible version of the `main.o` file.

So the first two `Aprog` lines tell `make` how to create the up-to-date version of `Aprog` and that that version depends on the most up-to-date version of `main.o`.

The next lines tell `make` how to get the most up-to-date version of `main.o`. First, we need the most up-to-date version of `main.cc`, so if there have been any edits to `main.cc` since the last time we created `main.o`, we have to execute the next line(s).

That line says: Issue the `GPP` command, which is a command to compile some code. The `-o main.o` says that the compiler should name the file that is the output of the compilation as `main.o`.

The `-c` says the the compilation step should only compile down to the “dot-oh” level, not all the way down to an executable (which would have a `.exe` file extension on a Windows machine, or would be named `a.out` by default on a Unix machine).

And the thing on which to act is the last thing on the line, namely the `main.cc` file itself.

1.8 More Comments

- The default on Unix is to name the executable `a.out`. I have for years labelled my executables `Aprog` because that file will sort to the top of a list of files when I do an `ls`, so I can always see the file. And I choose always to name the file rather than use a default, so if I need several versions, I can create several versions of the executable with different file names.

- Similarly, you will notice that all my input and output files will start with the letters `x`, `y`, or `z`. That's so they sort down to the bottom of the list of file names, and I don't have to search for them.

- The `make` program allows you to use only part of a `makefile`. If you type `make main.o` at the command line, then `make` will start with the `main.o:  main.cc` line of the `makefile` and only do what is necessary to ensure that an up-to-date version of `main.o` exists.

  It turns out in C++ that if you are compiling several files into one executable, sometimes the previous dot-oh is out of date but the source that created it has not changed. If this happens, it's possible for the program to fail to compile even though the source code is correct. If this happens (or when in doubt), it's useful to have a "clean" command. With this makefile, if you type `make clean` at the command line, it will remove all the dot-oh files and the executable and thus force `make` to rebuild everything.

  COMMENT: I strongly recommend you figure out the right way to do a "clean" command, either in the `makefile` or in the `bin` directory. It's almost a guarantee that if you don't do that, at some point during the semester you will do a fat-finger and type

  ```
  rm * .o
  ```

  instead of

  ```
  rm *.o
  ```

  and the difference is both striking and disastrous.

- Finally, you will not really see anything more complicated than this for the entire semester. The only changes you will see are that later makefiles will compile many separate files and classes into separate dot-oh files that are combined into one executable. But that's it.

1.9 `mydirectory`

You will always have a directory called `mydirectory`. Inside that will be a directory probably named with my name, like `hackerjrandom_hw1` (or something similar to this; I seem to keep changing my mind about the default name). Inside `hackerjrandom_hw1`. will be the initial code for you to start with and usually some sample output data, etc.

You will probably find several sample input files in the `hwprog1testdirectory` directory.

You should change the directory name to be YOUR university login name followed by the tag at the end that identifies which assignment. And you should edit your code inside this directory. If you do this, then all the scripts will work just fine, and your code in later assignments will find the utilities without changing code, etc. If you edit in some other directory, things may not work when I have your program tested for grading.

1.10 Testing scripts

The programming assignments will be tested using scripts. For this assignment, you have four scripts

- `zaZipUpScript.txt`
- `zbFileCopyScript.txt`
- `zcCompileScript.txt`
- `zdExecuteScript.txt`

For every assignment, you will get these four scripts. All but the last one will be identical for the entire semester, and the only difference that will happen in the last one is a change of where input comes from and output goes to.

YOU ARE RESPONSIBLE FOR UNDERSTANDING THE CONTENT OF THESE SCRIPTS. THAT IS, YOU SHOULD READ THE SCRIPTS AND BE SURE YOU UNDERSTAND WHAT THEY ARE DOING, ESPECIALLY WITH REGARD TO SOURCE OF INPUT AND DESTINATION OF OUTPUT FOR THE EXECUTION OF YOUR PROGRAM BY THE LAST OF THESE SCRIPTS.

1.11 zaZipUpScript.txt

```
#!/bin/bash
echo "Remove the old 'assignment' directory"
rm -r assignment
echo "Create a new 'assignment' directory"
mkdir assignment
#
echo "Descend into 'mydirectory' directory"
cd mydirectory
#
for item in *
do
    echo "Descend into directory" $item
    cd $item
    echo "Execute 'uname' to verify system"
    uname -a > SystemUNAME.txt
    echo "Return from directory" $item
    cd ..
    echo "tar directory" $item
    tar -cvf tarfile$item $item
    echo "gzip directory" $item
    gzip tarfile$item
    echo "cp gzipped file to 'assignment'"
    cp tarfile$item.gz ../assignment
    echo "rm the gzipped file from " $item
    rm tarfile$item.gz
done
#
echo "Return from 'mydirectory' directory"
cd ..
```

- Remove any old directory named `assignment`. Note that if the directory does not exist, you'll get a nasty message from the Unix shell, but execution of the script will continue.
- Create a directory named `assignment`.
- Descend into the directory `mydirectory`.

- For each item in this directory (and remember, these items will be the directory names similar to my original `buellduncan_hw1`),
  - Descend into each of these directories
  - Issue the `uname` command to write a file that has the system name for the computer on which all this is taking place
  - Back up to the `mydirectory` level
  - Issue the `tar` command to tar up each of these directories into a file with the item name as part of the name
  - Compress the tar file using `gzip`
  - Copy (NOT MOVE!) the zipped, tarred, file into the `assignment` directory
  - Delete the zipped file
- Back up to the main directory level of the script itself

What you should upload to Moodle is the tarred and zipped file that is put into the directory named `assignment` by the script `zaZipUpScript.txt`.

You may write your code in any environment you wish, but before you submit anything you should test your code in an appropriately-named directory underneath `mydirectory`. You should then use the `zaZipUpScript.txt` to create the appropriate file for submitting to Moodle.

If you can successfully run the four scripts and get the output file as needed, then your program will compile and execute correctly when I test your code.

Please note that when I test your code for a grade, I WILL NOT BE using the `zaZipUpScript.txt` script. Instead, I will download a zip file from Moodle and will upzip that big file into my own version of the `assignment` directory, so I will get each of your submissions as zipped tarred files in that directory, and the next three scripts let me upzip, compile, and execute.

IF YOU CREATE A FILE FOR SUBMISSION USING THIS SCRIPT, YOU CAN THEN USE THE NEXT THREE SCRIPTS TO TEST YOUR CODE IN ABSOLUTELY EXACTLY THE WAY IN WHICH I WILL TEST YOUR CODE.

1.12 zbFileCopyScript.txt

```
#!/bin/bash
echo "Remove the old 'testdirectory' directory"
rm -r testdirectory
echo "Create a new 'testdirectory' directory"
mkdir testdirectory
echo "cp all the submissions from 'assignment' into 'testdirectory'"
cp assignment/* testdirectory
#
echo "Descend into 'testdirectory' directory"
cd testdirectory
#
for item in *
do
    gunzip $item
done
#
for item in *
do
    echo "untar directory" $item
    tar -xvf $item
done
#
echo "Remove the tar files"
rm tar*
#
echo "Return from 'testdirectory' directory"
cd ..
```

- Remove any directory named `testdirectory` if such a directory exists.
- Create a directory named `testdirectory`.
- Copy all the individual files from `assignment` into `testdirectory`.
- Descend into `testdirectory`.
- For each item in this directory (that is, for each zipped tarred file from each student), unzip the file. (Note that the zipped file disappears and

a new file is created without the extension.)

- For each item in this directory (that is, for each tarred file from each student), untar the file.
- Remove all the tar files.
- Back up to the main directory level.

This script re-creates a `testdirectory`, but instead of what you see, which is one directory for only your code, I get a directory for each student.

1.13 zcCompileScript.txt

```
#!/bin/bash
echo "Descend into 'testdirectory' directory"
cd testdirectory
#
for item in *
do
  echo " "
  echo "COMPILING" $item
  cd $item
  rm *.o
  rm Aprog
  make -f ../../makefile
  cd ..
done
echo "Return from 'testdirectory' directory"
cd ..
#
echo " "
echo "COMPILING COMPLETE"
echo " "
```

- Descend into `testdirectory`.
- For each item in that directory (that is, for each directory, one per student),
  - Descend into that directory
  - Remove any dot-oh or `Aprog` executables.
  - Run `make`. NOTE THAT THIS USES THE `makefile` TWO LEVELS UP IN THE MAIN DIRECTORY AND WILL TOTALLY IGNORE ANY `makefile` IN THIS CURRENT DIRECTORY.
  - Back up one level back to the `testdirectory` level.
- Back up one level back to the main level.

This script will compile all your programs with one script.

1.14 zdExecuteScript.txt

```
#!/bin/bash
echo "Descend into 'testdirectory' directory"
cd testdirectory
#
for item in *
do
  echo " "
  echo "EXECUTING" $item
  cd $item
  Aprog >zoutput1
  cd ..
echo "EXECUTION COMPLETE"
done
echo "Return from 'testdirectory' directory"
cd ..
echo " "
```

This script descends into all the subdirectories of `testdirectory` and executes the correctly-named executable file created by `make` in the previous step. When you run this script, you should only have one directory. When I run this script, it will execute all the code from all the student submissions.

THIS IS THE ONLY ONE OF THE FOUR SCRIPTS THAT WILL CHANGE OVER THE COURSE OF THE SEMESTER, AND THE ONLY CHANGE YOU WILL SEE IS THE LINE

```
  Aprog >zoutput1
```

THAT ACTUALLY EXECUTES THE PROGRAM. THE CHANGES YOU WILL SEE WILL BE THAT INPUT AND OUTPUT WILL SOMETIMES COME FROM STANDARD INPUT OR GO TO STANDARD OUTPUT, AND WILL SOMETIMES COME FROM FILES OR GO TO FILES.

You may well find several input files, perhaps named `xin1.txt`, `xin2.txt`, `xin3.txt` ... in the main directory, but the script will have an execute line that directs `Aprog` from `xinput.txt`.

What to do?

Well, doh! Copy one of the `xinX.txt` files to a file named `xinput.txt`, and all is well. This allows you to run the program on multiple test inputs.

It also helps prevent one of the major fat-finger mistakes, of somehow telling the program that `xinput.txt` is the OUTPUT file, which will cause your input to be erased.

1.15 Assignment

This assignment is not about writing C++ code but rather is about verifying that you know how to submit programs for grading.

The ONLY THREE THINGS YOU MUST DO are to rename the directory to be YOUR name and not mine; to edit the `main.cc` file to have YOUR NAME in the documentation and in the output instead of mine; and to run the `zaZipUpScript.txt` to produce a gzipped tar file that can be uploaded from the `assignment` directory to the Moodle site.

HOWEVER, since the reason for having this assignment is so you can verify that you can submit programs that will run using my scripts, you are advised to run all four scripts in order. You should notice that your program compiles and you should notice that your program executes and produces the correct output.

You might also want to make a complete copy (with a different directory name) of your code, just to see how the scripts work if there is more than one directory to be worked with.

YOU ARE STRONGLY ENCOURAGED to read the documentation on the Moodle site about grading standards, programming style standards, and such. One of the crucial differences between this course and CSCE146 is that unlike Java, of which there is only one version and that is supposed to be promised to run exactly the same on any computer platform, you are now going to be writing code in a language with many compilers and under Unix/Linux guidelines that make it necessary for you to take care of certain nasty differences between Windows, Mac, and the various versions of Unix/Linux.

It is because there are these nasty variations that we have declared the Linux machines to be the reference machines. Your program might very well run on you personal computer and not compile on machines in the Linux lab. If that happens, you will get a zero. "But it runs on my computer" is no longer acceptable as any sort of excuse.

1.16 Linux Issues

You are responsible for reading and understanding the four scripts in the test directory and for reading and understanding the `makefile`.

This is not a course in Linux wizardry, and these scripts will not really change over the semester. The only script that will change is the "execute" script, and the only changes there will be that the line that invokes the executable module will change to take input from and deliver output to standard input/output or differently named files provided in the command line. A line

```
Aprog <xinput >xoutput
```

for example, will take input from standard input and send output to standout output. But a line

```
Aprog zinput.txt zoutput.txt zlog.txt
```

might take input from the file `zinput.txt`, send the formal output to `zoutput.txt`, and use `zlog.txt` as a log file.

We will talk about those things throughout the semester.

Similarly, the `makefile` will change only in that we will this first assignment only has one file of code to compile, but most of the time there will be several lines. So there will be "more" stuff added to the `makefile`, but it will not be any more complex as a directive to the `make` program than this one is.

1.17 Do Yourself a Favor

It has been my experience that a substantial number of 240 students have difficulty with 240 not because they don't understand the programming but because they choose not to make use of the enormous capability that Linux/Unix provides for making the programming process easier to do. You are strongly encouraged to read the list of Linux commands and become facile with the simple tools provided in Linux for making your life easier. Things will go faster and you will be able to spend more time thinking and less time fighting the computer.