Primary Objectives

1. Learn about directory trees and how to use them to your advantage.
2. Learn a few more UNIX commands: `mkdir`, `more`, `cat`, `rm`, `cd` and `pwd`.
3. Get some practice using the commands you discovered last week.
4. Gain experience editing a C++ program.
5. Compile and execute a C++ program.
6. Run a samba program.
7. Ask questions.

Before Getting Started

Before getting started you should setup an edit window for constructing a lab `handin.txt` file. The idea is that as you go through the lab, whenever it is indicated that something must be handed in, you can just copy and paste it from your terminal window into the `handin.txt` edit window. At the end of the lab you save the `handin.txt` file, print it, and then hand it in. If you aren't quite finished at the end of lab then you can save the file anyway and then when you return to finish the lab open an editor window and load the partly finished `handin.txt` file.

Every `handin.txt` file *must* start with a header containing your name, lecture section, lab section, date, etc. At this point open an edit window in `emacs` by pulling down the `Files` menu and choosing `Open File...`. This should bring up a directory path in the minibuffer to which you can append the file name `banner`, the file you are going to create. Create a banner by entering the following form:

```
//##########################################################
```

```
//            Name: Happy Student
//  Course : CSCI 203 / Prof. LectureProf / MWF ?:00
//  Lab Section: Monday 6-8AM    Section Number: nn
//     Assignment: Lab #?, or Project #?
//            Date: mm/dd/yy
//
//##############################################################
```

Make the appropriate changes, and save this file. Having done this, you should not have to repeat the construction.

Now to begin a `handin.txt` file for Lab 2, open a new edit window for a file named `handin.txt`. Insert the banner file by invoking the `Files` menu and selecting `Insert File...`. Enter the name of the banner file, i.e., `banner`. You should repeat this process at the beginning of every laboratory session. Now save this file as it currently is. Retain this edit window in the corner of your workspace so that when instructed you can copy data into it, thus gradually building up the file which you print and hand in.

Now, on to the lab exercises...

A Few Words About Directory Trees

As you progress through the semester you will be creating a lot of files. As a means of keeping your account organized, UNIX provides the ability to create a special file known as a "`directory`" — on a PC or Mac you may know these as "folders".

In computer file systems, directories are organized in a hierarchal fashion. Your account is, itself, a directory which is most often referred to as your *home* directory. In the directory hierarchy of Bucknell's file system there is a directory named "`student`" which contains 26 more directories named "`a`" through "`z`". Each directory named for a letter of the alphabet contains all the home directories of students whose account names begin with the respective letter. For instance, the student John Smith might have his directory in "`s`", if his account name is "`smith`"; if, on the other hand, his account name is "`jsmith`", then his home directory would be in the directory "`j`".

Each directory in the file system can contain several other directories. We refer to a directory within a directory as a "sub-directory" of the higher

level directory and the higher level directory as the "parent" of the sub-directory. For example, if your home directory were named "`jsmith`" it would be a sub-directory of the directory named "`j`", and `j` would be the parent directory of your home directory.

Recall that a directory is just a special type of file. It should not surprise you that a directory can contain text and other types of files as well as numerous directories. In this lab, you will create and use directories and sub-directories to help you keep your account organized; while this may not seem like a major issue at the moment, by the 4th or 5th lab you will have come to appreciate this simple yet powerful concept.

Naming Directories

Open a terminal window and at the UNIX prompt issue the "`ls`" command. You will probably only see the files you worked with last week.

Enter the following two commands at the terminal window's prompt.

```
mkdir cs203
```

and then

```
mkdir CS203
```

List your home directory again and you should see the following:

```
CS203/  cs203/
```

These entries illustrate two points. First, the "/", forward slash, character at the end of a file name indicates that the file is a directory. **Note:** this character "/" is *not* part of the file name! It is added to emphasize that the file is actually a directory. Second, UNIX distinguishes between upper and lower case letters; which can be annoying if you think you typed something correctly, but you substituted a lower case letter for an upper case one or visa-versa. One popular convention for naming files is to capitalize the first character of directory names and use a lower case letter for the first character of all other kinds of files. This makes it easy to distinguish the directories from files of other types.

Since you only need one directory for your course work, delete one of the directories by issuing the following command.

```
rmdir cs203
```

In future labs we will always refer to your `CS203` directory as the place where your work for the course is kept.

UNIX Commands

Knowing how to use directories and sub-directories and being able to manipulate the files in them will prove to be useful throughout this course. In this section of the lab you get a lot of practice with UNIX and hopefully a clear understanding of the file system. Now, read through all of sections 4.3 and 4.4 of The Guide and do the examples as indicated. Be very careful to watch for error messages which may occur — these are an indication that things didn't go as intended. Also, make frequent use of the commands `ls`, to determine directory content, and `pwd`, to determine your current working directory. In fact, do these two instructions now to see what they tell you.

Do sections 4.3 and 4.4 in The Guide.

When you have completed all the examples be sure you are in your home directory (use `pwd` command). If you aren't there issue the `cd` command with no directory argument. Now that you are in your home directory issue the following command:

```
ls -R
```

Cool, huh? The `-R` option instructs UNIX to *recursively* descend through the directories and perform the specified action, `ls` in this case, in each subdirectory.

Open your `handin.txt` file and, using the **Copy** and **Paste** keys you learned about last week, (these keys are on the left side of the keyboard), copy the data generated by the above command "`ls -R`", into your `emacs` window. What you need to copy begins with:

```
host-name#% ls -R
```

And ends with:

```
host-name#%
```

Having copied the data to the `handin.txt` window you should put a header on the copied data. Skip a couple of blank lines after the banner and then add 5 lines as follows (before your data)

```
//##############################################################
//
//   Exercise 1
//
//##############################################################
```

If you are unsure that things have gone as they should, have your professor or TA verify that you have done this correctly.

The time will come when you want to delete a directory for one reason or another. In section 4.3 of The Guide, you learned how to remove, delete, files with `rm`. Using this same command you can also remove directories, including all the sub-directories and files they may contain; can you guess why `rm` should be used with caution? Try the following:

```
rm progs
```

The message you see informs you that `progs` is a directory and just using `rm` will not delete a directory. The `rm` command used in conjunction with the option '`-r`' instructs the system to delete all the files in a directory (including any subdirectories) and the directory itself. Issue the following command:

```
rm -r progs
```

The system asks you if you want to look into the directory `progs`, respond with "`y`". Next the system asks you if you want to delete the file named `jack_theory`, reply with "`n`", then respond with "`y`" when asked if you want to remove `progs`. Because you instructed the system not to remove the file `jack_theory`, the directory is not empty and the system will not delete it even though you instructed it to. This helps to guard against deleting files and directories accidentally. Go ahead and issue the

```
rm -r progs
```

command again only this time answer with a "`y`" to all prompts. Now remove the `labs` directory and all its contents (this only requires one com-

mand). Next remove the file "`myhumpty`" from your home directory.

Using your newly gained knowledge:

1. Be sure that you are in your `CS203` directory (use the `pwd` command). Create two sub-directories of your `CS203` directory with the names `Projects` and `Labs`.

2. Enter the `Labs` directory and create two sub-directories named `Lab1` and `Lab2`.

3. Move the files you worked with last week into your `CS203/Labs/Lab1` directory.

If you are unsure, have your professor or TA check what you have done.

Your First C++ Program

Today you get your first "real" experience with a C++ program. Turn to page 60 in *Mercer* (your textbook), and read and do Programming Project

2A. To create a new file, open `emacs` and in the **Files** menu, select **Open File...**. Correct the directory path given in the minibuffer if necessary to indicate to *your* `Lab2` directory. Append `hello.cc` to the directory path in the minibuffer, making sure that a slash (`/`) separates the file name from the preceding directory names. Enter the program as it appears in the textbook and save it. Make sure, your terminal window is in the correct directory. Now compile this file by typing at the Unix prompt:

```
g++  hello.cc -o hello.exe
```

and press return. This instructs the system to compile the C++ program in the file `hello.cc` and create an executable named `hello.exe` — in other words, the compiler translates the program description in `hello.cc` into a machine language program which it puts in the file `hello.exe`. It is not important that you understand this `g++` command completely just yet — you will get a lot of experience with it this semester. If you have entered everything correctly, there will be no error messages printed from the compiler. If you have error messages, compare what you typed to what is in the text, make any corrections, save the file, and recompile it. Once the program

compiles without errors, continue.

Type the following at the system prompt:

```
hello.exe
```

If all has been entered correctly, you should see the words

```
Hello world!
```

appear on the screen. If not, raise your hand and have your professor or TA assist you.

Now complete the following activities using `hello.cc` as a base. (Recall that we used `endl` in the output statement (`cout`) to cause the screen output to go to a new line. You will need to use this to cause the output to have the appropriate blank lines.)

1. Modify the program so that, in addition to "Hello World!", the program also displays your name, with a blank before and after — your name should come after "Hello World!"

2. Compile and run the program and verify that your name appears as described.

Open the `handin.txt` edit window and create a new header (actually, just copy and paste the "exercise 1" header and change the name). Label this new header "`hello.cc`". Copy this enhanced "Hello World" program code to your `handin.txt` file along with the output from a test run of the program. (Use the insert file option from the **Files** menu to put the program code in and then do copy and paste to transfer the result of the execution.

Experience with Coding Errors

To give you more practice with the error messages that a compiler might give you, we have modified Lab Project `1C` from the Mercer text. Before you begin, issue the command

```
pwd
```

to see what your current directory is. The response to the command should be something like the following.

```
.../loginname/CS203/Labs/Lab2
```

If you don't see this then get your instructor's or TA's attention and ask for help.

Now to begin: you will need another file, `errors.cc`. Copy this file from `~cs203/Labs/Lab2` into your own `Lab2` directory.

Below is a series of steps to be performed along with some questions that you will need to answer. To answer the questions, create a file named `exercise-2`, using `emacs`. Type the answers as you go into the `handin.txt` edit window — you should make a new header and label it "`errors.cc`".

1. Compile and run `errors.cc`. Give a brief explanation of what the program does. In particular, what does the third line in the output indicate?

2. Modify the program according to the instructions below. For each

modification, compile the program and examine the error that you get. Provide a brief explanation of 1) what is wrong with the program, and 2) how the compiler reports the error. You may cut and paste error messages from the Terminal window if you wish to include them in your explanation.

Note: Since this file is short, it will be easy to make changes based on line number by just counting the lines. With larger files, this is more difficult. To move to a specific line in a file, type `C-c g`. After entering that command, type in the line number in the minibuffer. The cursor will then jump to that line.

Note: you should undo each change before moving on to the next part.

- Comment line 4 by adding `//` in front of `{`.
- Remove the opening angle "`<`" from the line

```
#include <iostream>
```

- Comment line 7.

- Remove closing double quotes from line 9.
- Replace << with >> in line 9.
- Replace the `()` after the word `main` with a `;` in line 3.

As a precaution you should save your `handin.txt` file at this time. Don't close the window because there is more to come, but it is a good habit to save your edit sessions frequently.

A Face — Using Samba

In the course we will be using a graphics-based tool called Samba.

Type in the following C++ Program

```
// A happy face drawing program
#include "/home/hydra/COURSES/cs203/include/Face.h"
```

```
int main ()
{
                          // create a Face object called
                          // myFace
   Face myFace;
   myFace.Display();      // a call to a member function
   pause();
   return 0;
}
```

Save the program as `face.cc`. The `#include` allows you to use some C++ code that we wrote. The include file defines the class `Face`.

Compile the Program

Compiling the program actually involves two activities. First, the program in `face.cc` is translated into machine code. But before it will work, the translated version must be linked together with other translated code for the

`Face` class. Both activities are specified in the same command. To compile, enter the following command line.

```
g++ face.cc -L/home/hydra/COURSES/cs203/lib -lSamba -o face.exe
```

The `-L` phrase specifies where the compiler should look for the other translated code, and the `-l` specifies the name of the file (called a library) which contains the code.

To save retyping that long line, you can retrieve the line at a Unix prompt by pressing the up-arrow key. Now you can edit the line using the left and right-arrow keys. If this does NOT work, ask your TA or instructor for help.

Run the Samba Program

To run this program you must type the following at a UNIX prompt.

```
face.exe | samba
```

The vertical bar is called a *pipe*. What this pipe does is to direct the output of your `face.exe` program to the input stream of the `samba` program. The

`samba` program will produce the nice graphical output. You should see a new window with **Samba** at the top. Move this window down and you should see a little window that says **Polka Control Panel**. Press **Start** on the **Polka Control Panel**. You should see a pretty face.

1. Show the face to your Instructor or TA.

2. Press **Quit** on Polka's window to quit.

Before leaving this program try the following. Execute the `face.exe` program on its own, without piping the output to `samba`. In other words, just enter the following command at the UNIX prompt and see what is displayed.

```
face.exe
```

What you will see is several lines of strange looking output which is exactly what was piped into `samba`. The lines you see are actually instructions which the `samba` program inputs and carries out.

What To Hand In

Save your `handin.txt` file at this point. You might want to look it over to make sure that what is there is in a nice format and that nothing is missing.

Print this file using the new `a2ps` command you learned last week.

```
a2ps handin.txt
```