1 Purpose

1. Exploring the emacs editor further.
2. Exploring the use of UNIX make facility.
3. Exploring multiple selection in C++ programs.

2 The emacs Editor

Create a Lab7 directory and do your work there.

In the following sections, you will be introduced to additional features of emacs editor. These features become increasingly useful as you begin to develop more complex programs.

2.1 Directory Editor — dired

You have learned about the Sun system's file manager and its capabilities. The emacs editor has a similar feature which makes managing and accessing files easier. This feature of emacs, called the Directory Editor or dired, will allow you to list the contents of directories and to easily move up and down in the directory tree. This first exercise will help you to navigate through the directory structure.

Begin by entering dired with the keystrokes C-x d — alternatively you can click on the folder icon (the second one) in the tool bar at the top. This should bring up a directory path in the mini-buffer at the bottom of the emacs window. Modify the path to indicate your Home directory and then press RET (the Return or Enter key). This action should produce a list of files similar to what one might see if the command ls -l is entered on the command line. Commands can now be entered for particular entries to accomplish things that might normally be done at the UNIX prompt — we will investigate some of these capabilities in later labs.

For now, using the mouse or arrow keys, move the cursor to the entry for

the CS203 directory which should be in the listing — click the left mouse button to set the cursor point on that entry (small black rectangle). To enter the directory press RET.

Continue into your Labs directory, and then into your Lab7 directory. Return to your Home directory by using the ^ key twice — this keystroke has the effect of 'cd ..' on the command line. Now go back to the Lab7 directory, and this time return to the Home directory by selecting the .. directory twice. So there are two ways of going down the directory tree.

You may select a file for editing by using the mouse or arrow keys to select the file in the directory listing and then pressing RET.

2.2 Compiling Programs From Within emacs

The emacs editor also has features which are designed specifically to help programmers with a specific programming language. You should be familiar with the way emacs will format your C++ code, but this is not the only C++ help provided. It is also possible to compile a program from emacs.

Some of you may have already discovered the emacs C++ menu at the

top-right of the menu bar in the emacs window. The entries on the C++ menu have been added by the Bucknell staff to make emacs a more convenient environment for the Computer Science students here at Bucknell. Click on the menu button now and look at the list of options listed there.

The first option we will discuss is the **Compile Buffer** action. Open up a C++ file using the technique described above. When the program file is loaded into emacs notice on the information line at the bottom of the emacs window that it says ('C++ Fill') — this indicates that a special C++ editing environment has been loaded into emacs.

You can compile the file which you see in the edit window simply by choosing the **Compile Buffer** command on the **C++** menu. Your edit window should split into two windows, with your program in one window and the compilation results (e.g., any error messages indicating syntax errors and/or completion messages) in the other. This second window has its own background menu to help you find your errors. To use the menu, move the cursor point into the compilation-results window and click, to set the cursor in that window. Now from the C++ menu, choose **First Error** and the cursor should switch back to the other window and locate the first error in

the buffer. You can then find the subsequent errors by using the **Next Error** item in the menu, or by using its equivalent `C-x ‘` keystroke (that’s a backwards single quote!

Another useful item in the C++ menu is the **Comment block** action and the related **Uncomment block** action. This has nothing to do with compiling, but it allows you to comment out multiple lines of code very quickly. You use the comment action by selecting (highlighting) the appropriate sequence of lines of your code and then choose the **Comment block** item in the menu. When commenting a selection, be sure to select the entire line for each line in the selection range. To remove the comment from that region, place the cursor anywhere in the commented region and choose **Uncomment block** in the menu.

2.3 Using a `Makefile`

You have seen in the last week or so that when compiling a C++ program it is sometimes necessary to compile more than one file and then to combine them in a step called *linking*. As programs get larger and have more classes,

the compiling process gets more complex. There is a command in UNIX, the make command, which is designed to make it easier to do routine activities with a collection of files. It is the command most commonly used to automate the compilation process.

The make command makes use of a special text file, called a Makefile — it is traditional to name the file Makefile. The Makefile has a special format which, if understood, can be used by a programmer to considerably simplify the compiling and linking process.

Here is the usual process followed by a programmer at the beginning of a new project. First create a new directory in which all files for the project will be maintained. Then a Makefile is created for the project — it may require modification as the project goes along. While the format for a Makefile is a bit complex, creating a new one isn't that hard. In fact many programmers just take one from a previous project and make appropriate modifications for the next project. Here are some rules to follow.

2.3.1 A Makefile Entry

A Makefile entry refers to a sequence of 1 or more lines which are used to accomplish a particular file operation; an entry has two parts — the target name part and the command part. The target name part comes first and consists of a name followed by a colon and then followed by a list of file names on which the entry depends. The command part comes next and begins on a new line; it begins with a TAB character and then a UNIX command. Here is an example:

```
Clicker.o: Clicker.cc Clicker.h
        g++ -O -Wall -c Clicker.cc
```

This entry creates a target name Clicker.o which depends on the files named Clicker.cc and Clicker.h. The command part, notice, is just the UNIX command to compile the Clicker.cc file. With this command in a file named Makefile, you can just enter on the command line the command

```
make Clicker.o
```

and the corresponding command for the target name will be executed.

Just a couple of comments about the command part of the entry above. With the "-c" option, g++ only compiles, i.e., it does not try to link, and creates the object file named Clicker.o. We list two files Clicker.cc, Clicker.h as *dependencies* on the first line meaning if we ever edit either of them, we want make to recompile and make a new version of Clicker.o.

During the software development phase of a project, we may need to recompile many times. The make tool keeps track of the last time a file was modified and recompiles only those files that have been modified. With a larger project, this can result in a significant saving in your time.

Study the Makefile we used in Lab 6. You should be able to understand it now.

Creating Your Own Makefile

Create a file called Makefile and type in the following simple example of a Makefile. Be sure to begin each *indented* line (i.e., the command part) with a TAB key, and not spaces. You will use this Makefile to compile the

programs that you create in the following three exercises.

```
# A comment for CSCI 203 Lab 7 Makefile
all: ex1.exe ex2.exe ex3.exe

ex1.exe: ex1.cc
	g++ -O -Wall -o ex1.exe ex1.cc

ex2.exe: ex2.cc
	g++ -O -Wall -o ex2.exe ex2.cc

ex3.exe: ex3.cc
	g++ -O -Wall -o ex3.exe ex3.cc

clean:
	rm -f core *.o ex1.exe ex2.exe ex3.exe
```

This Makefile works by examining the dependencies set down in the file and whether any modifications to the files imply that an executable needs to be re-compiled. For example, the second line of the file indicates that to build the target, **all**, the three files ex1.exe, ex2.exe, and ex3.exe are needed. It then proceeds to build each according to the rules found later in the Makefile. The rule for building ex1.exe says that it depends upon ex1.cc. Now if ex1.exe does not exist or ex1.cc is newer than (i.e., been modified since last build) ex1.exe, then it executes the commands under the dependency, in this case, the compile of ex1.cc.

You run make from the UNIX prompt by typing the following:

```
make <target>
```

where <target> is the target to make. If you type nothing after the make command, the make facility creates the *first* target in the file, in this case, all.

You may also run make by typing M-x compile while inside emacs.

Although the advantages of the make facility may not be completely clear at this time, you can appreciate the advantage with regards to compil-

ing more than one file at a time as you were doing in the previous lab with the `Clicker` class. The **Compile** via the `emacs` menu does not work in that case but `M-x compile` does.

3 Multi-way Selection

3.1 Exercise 1

Write a C++ program to solve the following problem. Call your file `ex1.cc`. to conform with your `Makefile`.

"The National Earthquake Information Center has asked you to write a C++ program implementing the following table to characterize an earthquake based on a Richter scale number. Your program should ask the user for the Richter number, *n*, and display the appropriate message."

Richter Scale Number n	**Characterization**
$0 \leq n < 5.0$	little or no damage
$5.0 \leq n < 5.5$	moderate damage
$5.5 \leq n < 6.5$	serious damage
$6.5 \leq n < 7.5$	disaster
$n \geq 7.5$	catastrophic damage

- Write the program using only a `main` function — no input validation needed.
- Also remember to use `make ex1.exe` to compile your program.
- Place in your `handin.txt` file a listing of your working program, and test runs for the following values for n: 4.4, 5.2, 5.5, 6.6, 7.7.

3.2 Exercise 2

Make a copy of your file `ex1.cc` and call it `ex2.cc`. Modify `ex2.cc` to include an input-validation function which is called appropriately in the `main`. The input function has the following interface.

```
double getBoundedLow(const string & inPrompt, double inLow)
   // Pre:  inPrompt and inLow have values AND
   //       cin == <v1 v2 ... vn ...>       AND
   //       v1 <= inLow, v2 <= inLow ,..., v(n-1) <= inLow AND
   //       vn > inLow
   // Post: cin == <...> AND returns vn
```

This function will read (i.e., input) a value from the user and will keep reading a value from the user until that value is greater than the parameter `inLow`. In your call to `getBoundedLow()` use 0.0 as the lower bound for input value.

Place in your `handin.txt` file a listing of your working program and test runs for the following values for n: 0.0, -10.0, 5.5, 10.0, 100.0. Again, remember to use `make ex2.exe` to compile your program.

3.3 Exercise 3

An if construct may be used inside of another if construct. This is referred to as "nested ifs". We indent the constructs to show the nesting as shown below: (Note: using the emacs tabbing feature makes formatting easy! To do this, press the Tab key **before** you press Return key when entering a line.)

```
if (betType == "win") {
    if (position == 1)
        cout << "Bet pays $20." << endl;
    else
        cout << "Too bad!" << endl;
} else {
    if (position == 1)
        cout << "Bet pays $10." << endl;
    else if (position == 2)
        cout << "Bet pays $5." << endl;
```

```
    else
        cout << "Bet pays $2." << endl;
}
```

This problem is somewhat complex but not difficult to understand. The main problem is to carry out different actions depending on whether there is a "win" or not. The action for each of these two cases is also a selection problem. So the nesting structure comes naturally from the nature of the problem.

1. Rewrite the selection statement above making use of a a switch statement rather than the if structure. Copy the new version of the code (the whole thing) to your handin.txt file.

2. Using nested ifs, write a C++ program where the user types in "small" or "large", for a small or large pizza, and the number of toppings. Depending on the size and number of toppings, the program should print the appropriate price of the pizza based on the chart below. Use

the `string` class to input the pizza size. You may use string literals to output the pizza cost. Here are the details on size/toppings/cost.

Toppings	**small**	**large**
0	$4.99	$6.99
1–2	$5.99	$7.99
3–5	$6.49	$8.19
6–9	$6.99	$8.89
10+	$7.99	$9.49

Be sure to implement the data input using functions, but you don't have to worry about validating the data — you can assume it is entered correctly.

Call your file `ex3.cc` to conform with your `Makefile`.

(a) Go through the selection design process to design the selection

based on size. Make a table and describe the action in each case in words.

(b) Now go through the same process for each action in the table just produced. You should find that each action also involves selection.

In this part, design each of these selections separately representing each design in a table.

Complete the program, compile it with `make ex3.exe` and test for 6 different size and number of toppings combinations.

Place in your `handin.txt` file a listing of your working program and the output from your test runs.

Try `make all` and see what happens.

Now do a `make clean` to remove the `.exe` files and redo the `make all`. What happened?

At the end of the lab, remember to do a `make clean` to remove any unwanted files.

4 What To Hand In

Your `handin.txt` file should now contain:

- Exercise 1 — program and test runs
- Exercise 2 — program and test runs
- Exercise 3 — code for the `switch` statement; program and test runs