1 Purpose

1. Learn more about `emacs`:Indentation, Filename Completion, Search, Search-and-Replace.
2. Practice getting input from files
3. Work with the C++ `vector` class
4. Combine input from files with vector operations
5. Practice the use of: `ifstream`, `for`, `while`, `vector` and `vector::push_back()`.

2 More Details on the `emacs` Editor

Create a `Lab8` directory and do your work there.

2.1 Coloring and Indentation

Select **Open File** and create a new file called `lab8`. Type in the following C++ code:

```
#include <iostream>
#include <string>

int getValue(const string & inPrompt) {

    int val;
    cout << inPrompt;
    cin  >> val;
    return val;
}

int main() {

    int value;
    value = getValue("Type in a quiz grade: ");
    cout <<  "value = " << value << endl;

    return 0;
}
```

But, the TAB key does not work properly and no colors. Why? Because the file name did *not* end in .cc. emacs uses the file name to determine if a particular "mode" should be used when editing the file. The basic no-frills mode is the 'Fundamental' mode, which you should see indicated on the status line (the darkened one) at the bottom of the emacs window. To get emacs into the proper mode 'C++ Fill' mode you need to rename the file you are editing. To do this select '**Save Buffer As...**' from the **File** menu and use the name lab8.cc. As an alternative, you can use the command M-x rename-buffer. Notice that the file name (at the left end of the status line) is the new name and that the mode indicator says (C++ Fill). Also notice that the text in the window is now colorized properly. If you put the cursor on the top line you can do the trick with the tab key to get all the lines properly indented.

There are many emacs modes for different languages and situations, for example, C mode, Fortran mode, Lisp mode, and LaTeX mode.

2.2 File Name Completion

Copy the files Fraction.h and Fraction.cc from ~cs203/include directory into your Lab8 directory.

```
cp ~cs203/include/Fraction.*  .
```

(Yes! That command will copy both.)

Now in emacs, select **Open File** and type *only* the character 'F'. Now press the space bar. What happened? emacs searched all the file names in the directory and completed as much of the name as possible. If there is only one file starting with 'F' then the entire file name will be filled in (completed). Then when you press return the file will be opened.

But in this case, you have two files starting with 'F' so the file name is only completed as far as the common part goes — i.e., up to the dot. Press the space bar again emacs splits your screen into two windows and displays the file choices which match your request so far. Move the mouse to the window with the file names and click (once) on Fraction.h — it will be highlighted. Now press return and the file you selected is opened.

Pressing the space bar to complete a file name is a feature which many emacs users use heavily!

2.3 Search and Replace

2.3.1 Search

If you ever need to search for some text in a program, e.g., an object or function name, you can use the *incremental search* feature of `emacs`. Here's a demonstration. Press `C-s` and type '`i`' or select **Search** in the **Search** menu (Don't press the `RET` key). Now, press '`n`'. Now press '`t`'. Notice that `emacs` finds the first '`i`', then from there the first '`in`', then the first '`int`'. Continuing to press `C-s` will move the pointer to the next occurrence.

But what do you do once you find the thing (or don't find the thing) you are looking for. If you want to stay where you find the last occurrence, then click the left mouse button near the string you found. This will quit the search process. If you want to quit searching you can also press `C-g`, but that will take you back to where you originally began the search.

After you have quit the search, if you press `C-s C-s`, the same search will be repeated. `C-r` does the same as `C-s` except it searches backwards through the buffer.

2.3.2 Replace

Suppose, in the text of the `lab8.cc` program, you want to replace all of the object names `value` with the name `cost`. Move the filled rectangle anywhere before the first occurrence of '`value`' (to the beginning of file is often a good idea). From the **Search** menu select **Query Replace...**. When `emacs` prompts you at the bottom of the window, type in the string to be replaced, in this case, `value`, then press `RET`. Now type in the new text, in this case, `cost` followed by a `RET`. `emacs` will move the filled rectangle to the first occurrence of `value` and wait. You either type '`y`' to replace it or '`n`' to not replace it. Then `emacs` goes to the next occurrence of the text and again you type '`y`' or '`n`' and so on. To get out prematurely, just press `RET`. Change all occurrences of `value` to `cost` in the file by using query replace.

3 Input from Files

For the first part of the lab, you will write a C++ program that counts the number of words in a message in a file, then displays the count.

- Open a file `message.txt` and type a message of your choice into it.
- Open a file `wordCount.cc`. In it, write a program that opens the file `message.txt`, then reads the words one at a time, counting each one as

it is read. A few things to note:

- You will need to use the directives

  ```
  #include <fstream>
  #include <string>
  ```

  to make your program work.
- Words can be read by using a loop. Use a while loop for your counting, with the usual loop condition — i.e., `while(inFile >> words)`.

After the words are read, print out each on a line by itself and display the message

```
There are <k> words in my message.
```

where `<k>` is replaced by the count of the words in the file.

- Run your program. Now edit your file `message.txt` to make the message longer. Run your program again (you do not need to recompile!) to see the changed count.
- Create a `handin.txt` file, and include your your original message and the output of running `wordCount.exe` on it, and your revised message and the result of running `wordCount.exe` on it. Print `handin.txt` and `wordCount.cc`.

4 The C++ `vector` Class

To use the vector class in your programs, you need to include the following header file (interface file):

```
#include <vector>
```

The general form of defining a vector object is the following:

```
vector<class-name> object-name(number-of-elements);
```

For example, to define an object `x` with 100 elements of the class `int`, we would type:

```
vector<int> x(100);
```

It's important to remember that vector indices always start at 0 in C++.

4.1 Part A

Write a C++ program that creates a vector of 25 integers and with a definite loop assigns the values 1 to 25 to the elements. Output the vector on a line with a space between each value. Add the listing of the program and run to your lab `handin.txt` file.

The class of the elements in a vector is *not* restricted to only built-in simple classes like `int` and `double`. It may be *any* class that has a default constructor (no parameters). For example, one can use a vector of `Fraction`.

4.2 Part B

For this part of the lab, you are to write a program that computes the average of ten user-input values of type `double`. Your program should do the following:

1. Create a vector of ten `double` values.
2. Prompt the reader to enter a value, then read that value into the next unassigned entry on the vector. This should be repeated 10 times, which will require the use of the repetition control structure. Remember that vectors use C++ indexing just as strings do; the first entry is in position 0, and the last will be in position 9.
3. *After input is complete* use a second loop to compute the average of the entries in the vector and output the average.
4. Print your program. Save a sample run in another file (such as `output.txt`) and print it too.

5 Combining Vectors and File Input

In this part of the lab you will combine reading data from a file with vectors. You will also practice using `push_back()` when you don't know in advance how big the vector should be.

Copy the program file `fracs.cc` and the data file `fractions.dat` from `~cs203/Labs/Lab8`. This program is meant to read fraction data from a file and then display the fractional sum of that data. The data file contains a sequence of lines, each containing two integer values representing the numerator and denominator (in that order) of a fraction. You should read them from the file using a `while(inFile >> num >> denom)` then use `num` and `denom` to insert another fraction into the `Fraction` vector.

The program file you have copied is incomplete. You should do the following with the file.

1. Put the appropriate declaration in `main()` for the `Fraction` vector. Notice that the rest of `main()` assumes the vector is named '`fracs`'. Don't declare the vector to have any particular size.

2. Complete the two calls in `main()` by making sure the parameter lists are correct and complete.

3. Complete the function `getFractions`. Use `push_back` to add elements to your vector so that it will grow as necessary.

4. Complete the function `sumFractions`. This should look familiar. Use the `vector` method `size()` to determine the size of the vector.

Compile and execute the program — remember to build the name of the data file into the program. You will also have to compile `Fraction.cc` in the `~cs203/include` directory along with your modified `fracs.cc`. You may do this step by either referring to the file directly on the compile line or copying over `Fraction.cc` into your space. It is preferable if you refer directly to the file. Also, prepare a `Makefile` so that you will not need to retype the compile line each time. An added benefit of a `Makefile` is that it is a record of what you need to do to build an executable for when you return to this project at a later date.

Save a sample run of your program in an output file and then print your program and the sample run.