Due Thursday, 1999 Nov 18, at the beginning of class.

1 Reading

Read Sections 6.1–6.3 and 10.3.

2 Compilation Using Xemacs Tip

In Homework 6, we introduced the idea of Makefiles to ease compilation. In homework 7, we described how to compile programs using Xemacs. Instead of typing ESC-x `compile`, you can click the compile icon. If you later decide to compile another program, choose the "Compile . . . " option from the Tools menu.

3 Problems

When submitting a program, please submit a printed copy of the program's code prepended with comments briefly describing its input and output. Please indent your code to make it readable. See Section 2.5 of the textbook for style hints. If you desire, you can also submit examples demonstrating your program's correctness.

1. In this problem, we will demonstrate that our rational number implementations compute exact rational values while using floating-point numbers can produce inexact answers.

   Write a program that computes the total profit from a set of stock purchases and sales. Each line of the input consists of a number of shares, a purchase price of one share, and a selling price of one share. An American or Canadian stock price is represented using an integer and a fraction with a small denominator. (Everywhere else in the world, decimals are used.) For examples, see the sample input file.

   Write two variants of the program: one using floating point numbers and one using a rational number implementation. When printing out the total profit (or loss if negative) using floating-point numbers, print the total and then print it again using `setprecision(20)` to see more of the decimal digits. See p. 239 of the textbook for an example of using the precision manipulator.

   Some time next year, American and Canadian stock prices will convert from fractional stock prices with denominators of 2, 4, 8, and 16 to floating point numbers having at most two decimal digits.

   A large software firm specializing in financial software telephones you asking for a two or three sentence statement backing the firm's claim that the conversion will not eliminate all the problems with using floating numbers rather than exact rational numbers to deal with stock prices. What statement would you give backing their claim? (Remember you are paid only if you support the company's claim.)

   Another large software firm specializing in financial software telephones you asking for a two or three sentence statement backing the firm's claim that using floating point numbers

does not currently cause problems and will not cause problems after the conversion. What statement would you give backing their claim? (Remember you are paid only if you support the company's claim.)

Yet another large software firm asks it can be sure to compute exact answers without rewriting its programs to use a rational number class. How do you respond?

2. In this problem, we extend the grep program we wrote in Homework 7 to print lines that match one of many possible patterns.

   Given a set of pattern specified using standard input and a file, multiGrep prints all the lines that contain at least one of the patterns. For example, given a file

```
1776
hello, world
21478756282
goodbye, world
```

   and two patterns of " world" and "2", all but the first line will be printed. Each line that has at least one pattern should be printed exactly once regardless of how many patterns match the line and how many times the patterns match.

   Your program should obtain the patterns using the standard input and obtain the filename using command-line arguments (not yet discussed in CS1320-4) or by querying the user.

   Reasonable Assumptions: Assume the user enters at most, say, fifty patterns, but make no assumptions about the file's length. It could be much larger than the computer's memory. Make *no* assumptions about the length of the patterns or the lines in the file. Patterns may contain spaces.

   Tip: The `string` class is presented in Section 10.3. A `string` object is very similar to a string stored in a character array. When strings are stored in character arrays, the arrays' lengths must be declared. Using `string` objects avoids this restriction by automatically acquiring enough space to store each string.

   Furthermore, there are many functions that make using `string`s easier than using strings stored in character arrays. See Display 10.11 on p. 632 for some of these functions. For the entire list of functions, see the Standard Template Library WWW pages. Here are some functions which might be useful:

   - *istream&* **`getline`**(*istream& stream*, *string& str*)
     - Read the characters of the next line from *stream* into the `string` *str*.
     - All characters (including leading whitespaces) are extracted.
     - The line delimiter (`'\n'`) is extracted but not appended.
   - `string::size_type` *str*.**`find`**(*const string& substring*)
     - This function searches for the first substring *substring* in the `string` *str*.
     - The function returns the index of the first character of the substring when successful or `string::npos` if it fails.

   `string::npos` is a constant with type `string::size_type` found in the `string` library. One can create an array of `string`s in the same way as other arrays of other types are created.

   We have posted some sample code. Note the initial

```
#include <string>
```

at the beginning of the program.