Problem Set 0

Due October 22, 2009, 11:59PM

Now that you've gotten a chance to see C++ in action, it's time to get your hands dirty and experiment with the material we've covered so far. This problem set will be an excellent starting point for you to explore exactly how much firepower C++ packs.

There are seven questions in this problem set, of which you only need to complete **five** to receive credit. That is, you can skip two of the questions without penalty. Of course, I recommend completing all of the exercises, since each question considers a different aspect of C++, but you are under no compulsion to do so.

You will probably want to have the course reader on-hand while answering these questions, as well as the lecture code that has been posted on the CS106L website. As always, feel free to email me with questions at htiek@cs.stanford.edu.

Good luck!

Question 0 (Adjusting to the Compiler)

You might want to read Chapter 1 of the course reader before answering this question.

Before playing around with the material we've covered this quarter, let's get warmed up by setting up a C++ project for compilation. If everything goes well, once you're done with this question you'll know how to compile and run C++ programs.

On the CS106L website I've provided a highly obfuscated C++ program called `sample.cpp` that performs a simple task. Download `sample.cpp`, compile it, and run it. What does the program do? *(Note: this is **not** the RickRoll program from lecture! You will actually need to compile this program to see what it does.)*

Question 1 (Console I/O)

You might want to read Chapter 3 of the course reader before answering this question.

Modify the code for `GetInteger` (found either on page 38 of the course reader or on the CS106L website) to create a function `GetReal` that reads a real number from the user. How much code did you need to change?

Question 2 (The X Macro Trick)

You might want to read Chapter 13, especially the latter half of the chapter, before answering this question.

Suppose that you are given the following `struct` representing a point in two-dimensional space:

```
struct pointT
{
    int x, y;
};
```

For simplicity, let's assume that we're working in the Cartesian plane, so *x* increases from left to right and *y* increases from bottom to top. That is, the point (3, 0) is to the east of (-1, 0) and the point (2, 4) is north of (2, 0).

Now, suppose that we want to codify the notion of a *direction* in two-dimensional space. To do this, we introduce an enumerated type called `directionT` with values `North`, `South`, `East`, and `West`. We want to support the following operations on `directionT`:

1. `DirectionToString`: Given a `directionT`, returns a string representation of that direction.
2. `OppositeDirection`: Given a `directionT`, returns its opposite.
3. `MovePointInDirection`: Given a `directionT` and a `pointT`, returns the `pointT` formed by moving the input `pointT` one step in the specified direction.

For example:

```
cout << DirectionToString(East) << endl;                       // Prints: East
cout << DirectionToString(OppositeDirection(North)) << endl; // Prints: South

pointT origin;
origin.x = origin.y = 0;
pointT dest = MovePointInDirection(dest, West); // dest = (-1, 0)
```

Using the X Macro trick, create a file `directions.h` containing information about the four directions and use it to implement the three above functions and to automatically generate a definition for `enum directionT`. You may want to use the colors example from the course reader and the lecture code as a reference. To test whether your setup works correctly, you should be able to introduce directions `NorthEast`, `NorthWest`, `SouthEast`, and `SouthWest` to `directions.h` without any hassle. *(Hint: While you're free to solve this as you see fit, it might help to have each macro encode a direction, its opposite, and the change in x and y associated with one step in the direction.)*

Question 3 (`vector` vs. `deque`)

You might want to read Chapter 4 of the course reader before answering this question.

As we discussed in lecture, inserting and removing elements at the end of a `deque` is considerably faster than performing the same operations on a `vector`. However, the `vector` has a useful member function called *`reserve`* that can be used to increase its performance against the `deque` in certain circumstances. The `reserve` function accepts an integer as a parameter and acts as a sort of "size hint" to the `vector`. Once you have called `reserve`, as long as the size of the `vector` is less than the number of elements you have `reserve`d, calls to `push_back` and `insert` on the `vector` will execute more quickly than normal. Once the `vector` exceeds the size you `reserve`d, operations revert to their original speed.*

The C++ standard library exports a header file called `<ctime>` that provides the `clock()` function and the `CLOCKS_PER_SEC` constant, which can be used to measure time. `clock()` returns the number of "clock ticks," a processor-specific measure of time, that have occurred since the start of the program. Using `CLOCKS_PER_SEC`, you can convert a number of clock ticks into a number of seconds. For example, here's some code timing how long it takes the user to enter a string:

```
cout << "Enter a string: ";

clock_t startTime = clock(); // clock_t is a type for storing clock cycles.
string line = GetLine();

/* Compute the number of clock ticks that have executed since we started
 * timing, then divide this value by CLOCKS_PER_SEC to get a number of
 * seconds back.  The static_cast<double> avoids rounding errors.
 */
double duration = (clock() - startTime) / static_cast<double>(CLOCKS_PER_SEC);

cout << "You took " << duration << " seconds to enter a string." << endl;
if(duration >= 10.0)
   cout << "(Slowpoke!)" << endl;
```

Write a program that uses `push_back` to insert a large number of `string`s into two different `vector`s – one which has had `reserve` called on it and one which hasn't – as well as a `deque`. The exact number and content of the `string`s is up to you, but large numbers of long strings will give the most impressive results. Using the tools from `<ctime>`, compute how long it takes to finish inserting the `string`s into each of the `vector`s and the `deque`. Did calling `reserve` help to make the `vector` more competitive against the `deque`? Now repeat this trial, but this time insert the elements at the *beginning* of each container rather than at the end. Is your conclusion still valid in this case?

* Calling `push_back` n times always takes O(n) time, whether or not you call `reserve`. However, calling `reserve` reduces the constant term in the big-O to a smaller value, meaning that the overall execution time is lower.

Question 4 (`map`)

You might want to read Chapters 6 – 7 of the course reader before answering this question.

Suppose that we have two `map<string, string>`s called `one` and `two`. We can define the *composition* of `one` and `two` (denoted `two ○ one`) as follows: for any string `r`, if `one[r]` is `s` and `two[s]` is `t`, then `(two ○ one)[r] = t`. That is, looking up an element `x` in the composition of the maps is equivalent to looking up the value associated with `x` in `one` and then looking up its associated value in `two`. If `one` does not contain `r` as a key or if `one[r]` is not a key in `two`, then `(two ○ one)[r]` is undefined.

Write a function `ComposeMaps` that takes in two `map<string, string>`s and returns a `map<string, string>` containing their composition.

Question 5 (General C++)

Consider the following game: We begin with a bag filled with five red chips, five blue chips, and five yellow chips. We then randomly choose two chips out of the bag and do the following:

- If both chips are yellow, we add one yellow chip and two blue chips to the bag.
- If one of the chips is yellow and the other isn't, we add three blue chips to the bag.
- If one of the chips is blue and the other isn't yellow, we add five red chips to the bag.
- Otherwise, if both chips are red, we put one red chip into the bag.

We keep repeating this process while there are more than two chips in the bag. It can be shown that this process is eventually guaranteed to terminate, but it may take a while to do so.*

A friend approaches you with a proposition. He bets you ten dollars, even money, that if you play this game, the process will take more than one hundred steps to terminate. Unsure of whether this is a good idea (moral objections aside), you decide to write a simulation to see, on average, how long this game takes to complete.

Write a program that simulates this game one hundred times and reports the average number of steps it takes for the game to end. To write this program, you'll need to be able to generate random numbers; see the FreeCell lecture code or Chapter 5 of the course reader for more information. Based on your results, should you accept the bet?

Question 6

Do you own a copy of the course reader? If so, are you finding it useful? Any suggestions for future editions?

How long did this problem set take you? How hard was it? Any suggestions for future problem sets?

(And yes, this does count as one of the five problems you need to do to complete this problem set.)

Deliverables

To submit this problem set, email your answers to htiek@cs.stanford.edu. If possible, please send all answers either in a plain-text format (C++ source files are plain-text) or as a PDF.

* This problem was inspired by an exercise in *The Calculus of Computation* by Aaron Bradley and Zohar Manna, which asks you to formally establish this result.