Problem Set 1

Due November 12, 2009, 11:59PM

In this problem set, you'll get a chance to play around with the concepts we've covered over the past few lectures. Some of these problems exercise specific aspects of the material, while others are more open-ended and should help give you a better feel for C++ as a whole. This problem set has five problems, of which you need to complete **four** to receive credit. As always, feel free to email me if you have any questions.

Question 0 (C Strings)

You might want to read Chapter 12 of the course reader before attempting this problem.

a. In lecture, we saw that the following program prints nonsensical output:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string myString = "" + 5;   // <-- Error here!
   cout << myString << endl;
}
```

Unfortunately, although the first line of `main` looks like it concatenates the number five onto the empty string, the code does something entirely different. What does the first line of `main` actually do? Why does this cause the program to print garbage characters?

b. Recall from our discussion of C strings that creating a deep copy of a C string is a two-step process. First, we allocate storage space for the copy of the C string, including space for the null terminator. Second, we copy the string to the new buffer using `strcpy`.

A friend calls you up asking for help on a C++ programming project. His program contains a bug that periodically causes it to crash. You look over his code and notice the following function, which is supposed to duplicate a C string:

```
char* StringDuplicate(const char* source)
{
     char* buffer = new char[strlen(source + 1)]; // <-- Error here!
     strcpy(buffer, source);
     return buffer;
}
```

You notice that this function contains a bug, which is highlighted in bold. Using your newfound knowledge of pointer arithmetic, explain why this code does not work correctly. *(Hint: Compare this code to the code we wrote in lecture and see if you notice any differences. The error is very subtle.)*

Question 1 (STL Algorithm Implementation)

You might want to read Chapter 14 and attempt some of the practice problems before starting this problem.

The *Cartesian product* of two collections A and B, denoted A × B, is the set of all pairs (*a*, *b*) where *a* is an element of A and *b* is an element of B. For example, given A = {a, b, c} and B = {1, 2}, the Cartesian product A × B is {(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)}. In this exercise, you'll implement a function in the style of the STL algorithms that computes the Cartesian product of two iterator ranges. We'll call this function `cartesian_product`.

Every element of a Cartesian product is an ordered pair, which we will represent using the standard `pair` type. This means that if we provide `cartesian_product` a range of iterators over elements of type `T1` and a range of iterators over elements of type `T2`, the function will generate a range of elements of type `pair<T1, T2>`. For example, calling `cartesian_product` on a range of iterators from a `vector<int>` and a `set<string>` will produce a range of values of type `pair<int, string>`.

Here is one possible prototype for `cartesian_product`:

```
template <typename ForwardItr1, typename ForwardItr2, typename OutputItr>
OutputItr cartesian_product(ForwardItr1 start1, ForwardItr1 stop1,
                            ForwardItr2 start2, ForwardItr2 stop2,
                            OutputItr where);
```

This function prototype is dense, so let's take a minute to look at exactly what it does. The first four parameters to this function are two pairs of iterators that correspond to the input ranges that we want to take the Cartesian product of. The last parameter (`OutputItr where`) is an iterator to the beginning of a range where the result should be stored. By convention, `cartesian_product` returns the iterator one past the last location written to. For comparison, consider the `set_intersection` algorithm we covered in lecture, which has a similar signature and semantics.

To summarize – the `cartesian_product` function computes the Cartesian product of the ranges [`start1`, `stop2`) and [`start2`, `stop2`) and writes the result to the range whose first element is pointed at by `where`. It then returns the iterator one past the last location written to.

a. Implement `cartesian_product`. *(Hint: use the `make_pair` function to generate the pairs of elements you will store in the output range. This lets you generate `pair`s even if you don't know the particular types of the elements in each range.)*
b. Show how to use the `cartesian_product` function to compute the Cartesian product of a `vector<int>` and a `deque<double>` and store the result in a `set<pair<int, double> >`. *(Hint: You'll need to use an `inserter` to solve this problem.)*

Question 2 (Copy Functions)

In lecture, we saw how to write the copy constructor, assignment operator, and destructor for a class in terms of the functions `clear` and `copyOther`. An alternative implementation of the assignment operator uses a technique called *copy-and-swap*. The first step in writing a copy-and-swap assignment operator is to write a member function that accepts a reference to another instance of the class, then exchanges the data members of the receiver object and the parameter. For example, when working with the `DebugVector`, we might write a function called `swapWith` as follows:

```
template <typename T>
void DebugVector<T>::swapWith(DebugVector& other)
{
    swap(array, other.array);
    swap(logicalLength, other.logicalLength);
    swap(allocatedLength, other.allocatedLength);
}
```

Here, we use the STL `swap` algorithm to exchange data members. Notice that we never actually deep-copy the elements stored in the `array` data member. Instead, we simply have the two `DebugVector`s exchange which dynamically-allocated arrays they point at.

Once we have an implementation of `swapWith`, we can use it to build the assignment operator as follows:

```
template <typename T>
DebugVector<T>& DebugVector<T>::operator= (const DebugVector& other)
{
    DebugVector temp(other);
    swapWith(temp);
    return *this;
}
```

a. Trace through this implementation of the assignment operator and explain how it sets the receiver object to be a deep-copy of the parameter. What function actually deep-copies the data? What function is responsible for cleaning up the old data members?
b. When writing an assignment operator using the pattern covered in lecture, we had to explicitly check for self-assignment in the body of the assignment operator. Explain why this is no longer necessary using copy-and-swap, but why it still might be a good idea to insert the self-assignment check anyway.

Question 3 (General C++)

You might want to skim Chapters 16 - 21 before attempting this problem.

On the previous problem set, we asked you to use the `clock()` function and the `CLOCKS_PER_SEC` constant from `<ctime>` to time how long it took to insert elements into two STL `vector`s and an STL `deque`. If you'll recall, you can time an event with these tools as follows:

```
/* Store the current time in "clock ticks." */
clock_t startTime = clock();

/* ... perform some complex task here ... */

/* Convert elapsed time (clock() - startTime) from clock ticks to seconds. */
double duration = (clock() - startTime) / static_cast<double>(CLOCKS_PER_SEC);
```

This code, while correct, is very difficult to read. The problem is that the code is too *mechanical* – it computes time by counting how many clock ticks elapsed between the start and stop times, then converting from clock ticks back into seconds. This is analogous to measuring a duration by counting how many times a pendulum swings back and forth in that time, then reporting the total time as the number of pendulum swings times the period of the pendulum. Sure, this will correctly measure how long the event takes, but it's considerably more work than it's worth.

Just as in the real world we can use a stopwatch to measure a duration, in C++ it's possible to design a stopwatch *class* that provides a clean interface for timing events. For example, the class would give us some means of designating some point in time as a "start time," as well as a means of querying how many seconds have elapsed since then. To time an event using the stopwatch object, we tell the object to start timing, perform some task we want to time, and then ask the object how much time elapsed. Of course, behind the scenes this stopwatch object would be implemented in terms of `clock()` and `CLOCKS_PER_SEC` using the above techniques, but because the stopwatch object encapsulates this behavior clients never need to know this. Think about a real stopwatch – we don't care how it works internally (maybe it's mechanical and uses an unwinding spring and array of gears, or perhaps it's digital and uses a vibrating quartz crystal) as long as we can easily read the time off of it.

Design and implement a `Stopwatch` class that provides two operations – allowing the user to mark the current time as the "start time" and reporting how much time has elapsed since the start time. You are free to design the class as you see fit, but it should adhere to standard C++ coding conventions (e.g. it should be `const`-correct, should either support copying or disallow it, etc.) Think from a client's perspective when designing the interface. Should you export a function that prints the elapsed time to the console, or should the function hand back the elapsed time to the caller? If you do return the elapsed time to the caller, should you return a value in clock ticks, seconds, or milliseconds? And what should happen if the client asks for an elapsed time without first having provided a start time?

As a sanity check on your design, this class should not be particularly complicated – my implementation is only about fifteen lines long.

Question 4

How long did this problem set take you? How hard was it? Did the questions help you get a better understanding of the material? *(And yes, this does count as one of the four problems you need to complete).*

Deliverables

To submit the assignment, email any files you've created 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.

Good luck!