Problem Set 2 Solutions

Problem 0

We can reverse a `stack` by funneling its elements into a `queue`, then reading them back out again.

```
void ReverseStack(stack<int>& toReverse)
{
    queue<int> buffer;
    while(!toReverse.empty())
    {
        buffer.push(toReverse.top());
        toReverse.pop();
    }
    while(!buffer.empty())
    {
        toReverse.push(buffer.front());
        buffer.pop();
    }
}
```

Problem 1

Generating a generalized Fibonacci sequence can be done using a standard for loop:

```
const int NUM_FIBONACCI_NUMBERS = 20;
void GenerateGeneralizedFibonacciNumbers(vector<int>& out, int one, int two)
{
    out.push_back(one);
    out.push_back(two);
    for(int k = 2; k < NUM_FIBONACCI_NUMBERS; ++k)
        out.push_back(out[out.size() - 1] + out[out.size() - 2]);
}
```

As an interesting aside, notice that if `NUM_FIBONACCI_NUMBERS` is defined to be less than two (say, 0 or 1) that the `GenerateGeneralizedFibonacciNumbers` function will not work correctly since it always adds two elements. We might therefore want to put an `assert` statement in to check whether `NUM_FIBONACCI_NUMBERS` is at least two. While this will help us catch any errors stemming from a poorly-defined `NUM_FIBONACCI_NUMBERS` constant, we will only detect the error if we actually call the `GenerateGeneralizedFibonacciNumbers` function. For the sorts of programs and examples we're dealing with in CS106L, it's reasonable to assume that all of the functions we write will be called at some point, but when working on larger code bases it might take several program runs before the function is called and the error discovered.

Notice, however, that the condition we're checking is determinable at compile-time. It seems as though there should be some way for the C++ compiler to check at compile-time whether `GenerateGeneralizedFibonacciNumbers` is indeed at least two and to report an error otherwise. This would make it much easier to isolate the root cause of the problem.

While the current incarnation of C++ lacks support for this sort of checking,* C++0x will contain a `static_assert` keyword which, like `assert`, checks whether a condition holds and reports an error otherwise. However, `static_assert` only checks conditions at compile-time, meaning that it is designed to catch an entirely different class of errors than `assert`. Using `static_assert`, we can rewrite the above function as follows:

```
const int NUM_FIBONACCI_NUMBERS = 20;
void GenerateGeneralizedFibonacciNumbers(vector<int>& out, int one, int two)
{
    static_assert(NUM_FIBONACCI_NUMBERS >= 2,
                  "NUM_FIBONACCI_NUMBERS must be at least two.");

    out.push_back(one);
    out.push_back(two);
    for(int k = 2; k < NUM_FIBONACCI_NUMBERS; ++k)
        out.push_back(out[out.size() - 1] + out[out.size() - 2]);
}
```

Problem 2

Here are the three versions of `Cycle`:

```
void CycleQueue(queue<int>& myQueue, int numIterations)
{
    for(int k = 0; k < numIterations; ++k)
    {
        myQueue.push(myQueue.front());
        myQueue.pop();
    }
}

void CycleQueue(deque<int>& myDeque, int numIterations)
{
    for(int k = 0; k < numIterations; ++k)
    {
        myDeque.push_back(myDeque .front());
        myDeque.pop_front();
    }
}

void CycleVector(vector<int>& myVector, int numIterations)
{
    for(int k = 0; k < numIterations; ++k)
    {
        myVector.push_back(myVector.front());
        myVector.erase(myVector.begin());
    }
}
```

The `CycleVector` function is less efficient than the `CycleDeque` function because `vector` does not support efficient removal of elements at the beginning of the sequence. `deque`, on the other hand, is optimized for this case.

---

* Such a result can be achieved through various template magicks – the Boost C++ libraries contain a macro called `BOOST_STATIC_ASSERT` which checks conditions at compile-time.

Problem 3

There are many possible solutions to this problem, one of which is printed below:

```
#include <iostream>
#include <vector>
#include <deque>
#include <ctime>
#include <string>
using namespace std;

const int NUM_STRINGS = 50000;
const string SOURCE_STRING = "Ce n'est pas une chaîne de caractères.";

void TimeVector(const bool reserveFirst)
{
    vector<string> trialObject;
    if(reserveFirst) trialObject.reserve(NUM_STRINGS);

    clock_t startTime = clock();

    for(int k = 0; k < NUM_STRINGS; ++k)
        trialObject.push_back(SOURCE_STRING);

    startTime = clock() - startTime;

    cout << "Time  " << (reserveFirst ? "with" : "without") << " reserve: "
         << (static_cast<double>(startTime) / CLOCKS_PER_SEC) << endl;
}

int main()
{
    TimeVector(true);
    TimeVector(false);
}
```

Problem 4

There are many good solutions to this problem. My personal favorite is this one:

```
void CountLetters(ifstream& input, map<char, int>& freqMap)
{
    char ch;
    while(input.get(ch)) ++freqMap[ch];
}
```

This code is pretty dense and relies on several properties of the IOStream library and the STL. First, the `ifstream` member function `get` accepts as input a `char` by reference, then reads in a single character from the stream. On success, the function fills in the `char` with the value read. On failure, the value is unchanged and the stream goes into a fail state. The `get` function then returns a reference to the stream object that did the reading, meaning that `while(input.get(ch))` is equivalent to

```
while(true)
{
    input.get(ch);
    if(!input) break;

    /* ... body of loop ... */
}
```

And since `!input` is equivalent to `!(input.fail() || input.bad())`, this one-liner will read in a character from the file, then break out of the loop if the read failed.

Once we've read in the character, we can simply write `++freqMap[ch]`, since if the key already exists we're incrementing the older value and if not a new key/value pair will be created with value 0, which is then incremented up to one.

Problem 5

Sorting a `vector` with a `multiset` can be written concisely if we use the STL containers' `insert` and `clear` functions:

```
void SetSortVector(vector<int>& toSort)
{
    /* The map, set, multimap, and multiset each define a constructor
     * which accepts a range of values, then populates the container
     * with the elements in that range.
     */
    multiset<int> sorter(toSort.begin(), toSort.end());

    /* Clear out the vector, then insert the elements from the multiset. */
    toSort.clear();
    toSort.insert(toSort.begin(), sorter.begin(), sorter.end());
}
```

The `multiset` is necessary so that if the `vector` contains any duplicate values, the sorted version of the `vector` also contains those duplicates.

Problem 6

One solution looks like this:

```
const int NUM_ELEMS = 100;

list<int> myList;

for(int k = 0; k < NUM_ELEMS; ++k)
    myList.push_back(k);

for(list<int>::const_iterator itr = myList.begin();
    itr != myList.end(); ++itr)
    cout << *itr << endl;
```

Problem 7

Using a modified version of the code from Problem 5, we have this solution:

```
void ListSortVector(vector<int>& toSort)
{
    list<int> sorter;
    sorter.insert(sorter.begin(), toSort.begin(), toSort.end());
    sorter.sort();
    toSort.clear();
    toSort.insert(toSort.begin(), sorter.begin(), sorter.end());
}
```