Problem Set 1 Solutions

Problem 0

a. The reason that the code doesn't work correctly is that `"" + 5` is interpreted as "the pointer five steps past the beginning of the empty string" rather than "the string formed by concatenating the empty string and the number five." Taking five steps past the beginning of the empty string walks off the end of string and into random memory. The C++ standard says that this falls into the realm of undefined behavior, so the actual string you get back might depend on your operating system, what compiler you're using, or the alignment of the planets. Of course, you might end up with a pointer to memory that you don't own, so printing this string might completely crash the program.

b. The problem with the function is that the while line

```
char* buffer = new char[strlen(source + 1)];
```

looks like it allocates enough storage space for the C string, including the null terminator, it instead does something else. The `+ 1` in the above statement means "take one step past the beginning of `source`" rather than "reserve one more byte than the length of the string." You can see this better by comparing it against a correct implementation:

```
char* buffer = new char[strlen(source) + 1];
```

The string `source + 1` is the same string as `source`, but with the first character removed. Assuming that `source` is n characters long, this means that the faulty implementation only allocates space for $n - 1$ characters. It is therefore underallocating by two bytes – one byte for the missing character of `source` and one byte for the null terminator. Copying the contents of `source` into this buffer therefore writes into invalid memory and will almost certainly crash the program.

Problem 1

I was pleasantly surprised by how many of you were able to come with the solution to this problem. The actual code you need to write is not particularly long, but it's fairly dense and you need to have a solid understanding of templates to appreciate how it works.

Here is one possible implementation:

```
template <typename ForwardItr1, typename ForwardItr2, typename OutputItr>
OutputItr cartesian_product(ForwardItr1 start1, ForwardItr1 stop1,
                            ForwardItr2 start2, ForwardItr2 stop2,
                            OutputItr where)
{
    for(ForwardItr1 one = start1; one != stop1; ++one)
    {
        for(ForwardItr2 two = start2; two != stop2; ++two)
        {
            *where = make_pair(*one, *two);
            ++where;
        }
    }
    return where;
}
```

One issue that many of you ran into on this problem is how to figure out what types of elements `ForwardItr1` and `ForwardItr2` iterate over. It turns out that without using some fairly advanced techniques we haven't talked about, it is impossible to determine these types. That is, given just the type `ForwardItr1`, we can't tell whether it iterates over `int`s, `double`s, `string`s, or something more complex. The way to get around this is through `make_pair`. Since template functions automatically deduce the types of their arguments, `make_pair` lets us create a `pair` whose type matches the type of the elements being iterated over without us ever having to spell that name out explicitly. That is, the statement `make_pair(*one, *two)` will construct a pair of the right type even though we don't actually know what the types of `*one` and `*two` are.

If you're curious about how to recover the type of element being iterated over from an iterator type, you can use the `iterator_traits` class, exported by `<iterator>`. `iterator_traits` contains a wealth of information about the properties of iterator types, ranging from the class of iterator (input, output, forward, bidirectional, random-access) to the type of element the iterator iterates over. The catch, though, is that `iterator_traits` makes code nightmarishly difficult to understand. For reference, here's code for `cartesian_product` that uses `iterator_traits` to explicitly create the `pair` of elements that gets written into the output range:

```
template <typename ForwardItr1, typename ForwardItr2, typename OutputItr>
OutputItr cartesian_product(ForwardItr1 start1, ForwardItr1 stop1,
                            ForwardItr2 start2, ForwardItr2 stop2,
                            OutputItr where)
{
    for(ForwardItr1 one = start1; one != stop1; ++one)
    {
        for(ForwardItr2 two = start2; two != stop2; ++two)
        {
            pair<typename iterator_traits<ForwardItr1>::value_type,
                 typename iterator_traits<ForwardItr2>::value_type>
               toInsert(*one, *two);
            *where = toInsert;
            ++where;
        }
    }
    return where;
}
```

That's quite a mouthful, which should make you appreciate why the version using `make_pair` is so elegant.

We can use `cartesian_product` to generate the Cartesian product of a `vector<int>` and `deque<double>` as follows:

```
vector<int> myVector = /* ... */;
deque<double> myDeque = /* ... */;

set<pair<int, double> > result;
cartesian_product(myVector.begin(), myVector.end(),
                  myDeque.begin(), myDeque.end(),
                  inserter(result, result.begin());
```

It is crucial that you use `inserter` here as the destination instead of just `result.begin()`. Remember that all STL algorithms, including our new friend `cartesian_product`, expect that there is enough room in the destination range to hold all of the elements that will be generated. Using `inserter` ensures that space exists for the elements, since as elements are generated by `cartesian_product` the `inserter` will create space for them. If we were to pass `result.begin()` as the final parameter, the algorithm would try to write values into the range beginning with `result.begin()`, which is a problem because that range is empty. Some of you tried to get around this by adding elements to the `set` prior to calling `cartesian_product` to ensure that space exists, but this is also problematic because the elements generated by `cartesian_product` are not guaranteed to be produced in sorted order. Overwriting `set` elements with unsorted values disrupts the `set`'s internal ordering, which as we saw in lecture can cause all sorts of problems. Consequently, `inserter` is the way to go.

Problem 2

I've reprinted the implementation of `operator=` below for reference:

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

This assignment operator works as follows:

1. The statement `DebugVector temp(other)` uses a the `DebugVector` copy constructor to create a temporary `DebugVector` initialized as a copy of `other`.
2. The statement `swapWith(temp)` then exchanges the data members of the receiver object and `temp`. This means that the receiver's data members are now a deep-copy of `other`'s data members, and `temp` now holds all of the receiver object's old data members.
3. `temp` goes out of scope as the function returns, invoking its destructor and cleaning up its the receiver object's old data members.

This means that the copy constructor copies `other`'s elements and the destructor cleans up the receiver's resources. In other words, copy-and-swap implements the assignment operator in terms of the copy constructor and destructor.

Using copy-and-swap also renders the self-assignment check `if(this != &other)` unnecessary. If we assign an object to itself, then the copy-and-swap approach will make a deep-copy of the object before it cleans up its object's resources. Consequently, the object will be assigned a copy of itself before its old value is cleared, rather than the other way around. Of course, we still might want to put in a self-assignment check anyway to avoid the cost of copying objects unnecessarily, but doing so isn't going to affect the correctness of the function.

Problem 3

This question is extremely open-ended and I was very pleased with the variety of answers I received. There is no one "right answer" to this question and in the feedback emails I'll send back about the assignment I will try to offer specific critiques of your particular design. There are, however, a few general design decisions that seem better than others. In particular:

- *The class should return a time rather than printing it to the console.* It is tempting to create a `Stopwatch` class that reports times by displaying them to `cout`. Unfortunately, this makes `Stopwatch` inflexible. First, it means that the rest of the program can't use the information about the elapsed time. If we want to measure time in order to adjust some program property (such as a conveyor belt rate), we need to get the value back in a way that can be manipulated by the rest of the program. Second, it means that clients that want to display the resulting time using some other visualization (either graphical or console-based) cannot use `Stopwatch` because the output format is hard-coded into the class.
- *The class should return times in seconds rather than in clock ticks*. Just as a digital watch doesn't report times in numbers of quartz crystal vibrations, a `Stopwatch` class shouldn't report times in clock ticks. The real world measures events in seconds (or milliseconds, nanoseconds, etc.), and if we hand back a value in clock ticks the client will have to manually perform the conversion to seconds, reducing `Stopwatch`'s usefulness. If you handed back a value in milliseconds or microseconds, that's fine too.

One question which is more open-ended is what to do if the user asks for how much time has elapsed without first starting the stopwatch. While there are many options, one option is to sidestep the issue by having the `Stopwatch` constructor automatically start the timer. This means that querying for an elapsed time before setting a start point measures how much time has elapsed since the `Stopwatch` was created.

Given these considerations, here's one possible implementation of `Stopwatch`:

```
class Stopwatch
{
public:
    Stopwatch();

    void startTimer();
    double secondsElapsed() const;
private:
    clock_t start;
};

Stopwatch::Stopwatch()
{
    startTimer();
}

void Stopwatch::startTimer()
{
    start = clock();
}

double Stopwatch::secondsElapsed() const
{
    return (clock() - start) / static_cast<double>(CLOCKS_PER_SEC);
}
```