Practice CS193D Midterm Solution

Problem 1: Short Answers

Circle the one compile-time error of the code snippet that follows, and then explain what the compile-time error is.

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;
*b = 7;
*d = 8;
```

The problem occurs with the second to last line of code. b is a reference parameter, and after it's declared and initialized, b is of type int and is simply another name we can use to refer to the variable also know as a. It is not a pointer to an int, so attempts to traverse b as if it were one are blocked by the compiler. Bottom line: int& and int* are completely different types.

Circle the one compile-time error of the code snippet that follows, and then explain what the compile-time error is.

```
const char ch = 'a';
char *charRef = &ch;
(*charRef)++;
```

Many people said the third line was the problem, but the third line is actually just fine. Recall that `const char` and `char` are typed differently, and the compiler is fully aware that a `char` is a mutable variable and a `const char` is not. The compiler doesn't trust a plain old `char *` variable to store the address of a character that has been marked as `const`, so it flags the second line as a full-blown error. You could cast away the `const`-ness of `&ch`, but we didn't do that here, so `gcc` says no way.

Recall that all overloaded `operator=` methods first check for self-assignment before they actually do any copying. In the case of the `MyString` class, for instance, the overloaded `operator=` method looked something like this:

```
const MyString& MyString::operator=(const MyString& ms)
{
   if (this != &ms) {
      delete[] this->chars;   // safe since they aren't the same string..
      this->length = ms.length;
      this->chars = new char[ms.length + 1];
      strcpy(this->chars, ms.chars);
   }

   return *this;
}
```

Why isn't the same '`if (this != &ms)`' check included in the implementation of a copy constructor? Why needn't `MyString`'s copy constructor look like this? Will any runtime errors ever result from this implementation?

```
MyString::MyString(const MyString& ms)
{
   if (this != &ms) {
      this->length = ms.length;
      this->chars = new char[ms.length + 1];
      strcpy(this->chars, ms.chars);
   }
}
```

If the constructor was actually coded this way, then it would imply that one might be able to construct an object from itself using the copy constructor—that is, the programmer would probably be protecting us from what happens when we creates objects like this:

```
MyString a(a);
MyString *b = new MyString(*b);
```

However, it is nonsense to think that `a` could be initialized--constructed for the very first time—from a. In fact, the argument to the constructor isn't even in scope yet, since the variable is being defined right there, so the compiler will laugh at you. Ergo, the `(this != &ms)` check isn't needed, because it can never evaluate to `false`.

Does it cause a problem? Nope. It's just useless code that the application blindly executes because it's too dumb to know any better. Some compilers might strip out the useless test, but I doubt it.

For the following code, assume the deep copying `string` object has already been included. Assume that the `foo` function has just been called, and give a possible ordering for all the calls to the `string` memory management routines: the default constructor, the (`char*`) constructor, the copy constructor, the `operator=` assignment operator, and the destructor. Include consideration of the parameter set-up for function calls.

```
void foo()
{
   string t1("hello");
   string *t2 = new string("there");
   t1 = bar(t1, *t2);
}

string bar(string &a, string b)
{
   string *p;
   string *q = &b;
   p = new string(a);
   a = *q;
   delete p;
   return a;
}
```

1. `string(const char *)` called on behalf of the stack-based `t1`.
2. `string(const char *)` called on behalf of the heap-based `t2`.
3. `string(const string&)` called on behalf of `bar`'s `b` parameter.
4. `string(const string&)` called just prior to assignment to `p`.
5. `operator=(const string&)` called to reassign `a` to be a copy of `*q`.
6. `~string()` called in reponse to `operator delete` call.
7. `string(const string&)` called to create a temporary `string` to return.
8. `~string()` called to clean up `b`, which is going out of scope.
9. `operator=(const string&)` called to reassign `t1` to be a copy of `return` val.
10. `~string()` called to clean up temporary.
11. `~string()` called to clean up `t1`, which is going out of scope.

Problem 2: Stock Portfolios

Now that January 1, 2000 is well behind us, we're all convinced that the Y2K bug poses no further threat to the computer industry. Silicon Valley is home to more self-made millionaires than can be stored in a `long`, so we'd like to take the stash of Christmas money we still have and invest in the technology sector of the stock market. To that end, you've decided to devise some C++ classes to help you keep track of all the stock you'll be purchasing now that you're sure the market will never crash.

The `Stock` class is defined as follows:

```
class Stock {

   friend class StockPortfolio; // grants visibility to Stock's private data

   public:
      Stock();
      Stock(const char *symbol,
            const string& companyName, const double& value);
      ~Stock();

      friend ostream& operator<<(ostream& os, const Stock& stock);
      void update(const double& delta);

   private:
      const static int kSymbolCodeLength = 5;
      char symbol[kSymbolCodeLength];
      string company;
      double value;
};
```

This class encapsulates all the information about a particular stock traded on the market. More specifically, information is maintained about the company name, the stock market symbol used to compactly and uniquely identify the company, and the value of a single share of that company's stock.

Functionality to publish the content of a `Stock` instance is included via an overloaded version of the global function `operator<<`, and the member function `update` may be used to change the value of a single `Stock` instance.

The manner in which a `Stock` item is published to an `ostream` is in accord with the following format:

```
company-name (symbol): single-share-value
```

Therefore, the following piece of code:

```
Stock microsoftcorp("MSFT", "Microsoft Corporation", 71.675);
cout << microsoftcorp;
```

would result in the following being printed to standard out:

```
Microsoft Corporation (MSFT): 71.675
```

On this and the next page, provide implementations for all constructors, destructors, methods, and `friend` functions. You may assume that all standard libraries and CS193D classes are visible to you, so you needn't be careful about stating which classes are `#included` and which ones aren't.

```
class Stock {

   public:
      Stock();
      Stock(const char *symbol,
            const string& companyName, const Fraction& value);
      ~Stock() {}

      friend ostream& operator<<(ostream& out, const Stock& stock);
      const Stock& operator+=(const double& delta);

   private:
      const static int kSymbolCodeLength = 5;
      char symbol[kSymbolCodeLength];
      string company;
      double value;
};

Stock::Stock()
{
   strcpy(this->symbol, "");  // need to ensure that operator<< behaves
}

Stock::Stock(const char *symbol,
             const string& companyName, const double& value)
   : company(companyName), value(value)
{
   strncpy(this->symbol, symbol, kSymbolCodeLength - 1);
}

ostream& operator<<(ostream& out, const Stock& stock)
{
   out << stock.company << " (" << stock.symbol << " ): " << stock.value;
   return out;
}

const Stock& Stock::operator+=(const double& delta)
{
   value += delta;
   return *this;
}
```

The classes don't end there. You're so delighted with your `Stock` class that you've decided to contrive a `StockPortfolio` class to keep track of all the different stocks you own.

```
class StockPortfolio {

   public:
      StockPortfolio();
      ~StockPortfolio();

      void buy(const Stock& stock, int quantity);
      double computePortfolioValue() const;

   private:
      struct pair {
         const Stock& stock;
         int quantity;
         pair(const Stock& stock, int quantity)
            : stock(stock), quantity(quantity) {}
      };

   vector<pair> holdings;

};
```

This defines a class that encapsulates information about the number and types of stocks owned by a particular investor. The definition could be much more robust, but here we provide a reduced interface to export a modicum of functionality. Initially, a newly created `StockPortfolio` contains no shares of stock whatsoever, but stock purchases can be made via use of the `buy` method, which allows the client to introduce additional shares of a particular stock to a portfolio. Functionality is also provided to compute the total value of all shares in the portfolio (using the `computePortfolioValue` method.)

Internally, a `StockPortfolio` instance maintains a `vector` of stock-quantity pairs, where the length of this `vector` is **always equal to the number of companies represented by the portfolio**. Note that the `StockPortfolio` class was granted `friend`ship within the declaration of the `Stock` class. Also note that the helper `struct pair` can be used to build your little (or possibly big—let's hope so!) `vector` very easily.

Implement all `StockPortfolio` constructors, destructors, and methods.

```
class StockPortfolio {

   public:
      StockPortfolio();
      ~StockPortfolio();

      void buy(const Stock& stock, int quantity);
      const double computePortfolioValue() const;

   private:
      struct pair {
         const Stock& stock;
         int quantity;
         pair(const Stock& stock, int quantity)
            : stock(stock), quantity(quantity) {}
      };

   vector<pair> holdings;
};

StockPortfolio::StockPortfolio() {}
StockPortfolio::~StockPortfolio() {}

void StockPortfolio::buy(const Stock& stock, int quantity)
{
   for (pair* curr = holdings.begin(); curr != holdings.end(); curr++) {
      if (stock.company == curr->stock.company) {
         curr->quantity += quantity;
         return;
      }
   }

   holdings.push_back(pair(stock, quantity));
}

const Fraction StockPortfolio::computePortfolioValue() const
{
   double totalValue = 0.0;
   for (pair* curr = holdings.begin(); curr != holdings.end(); curr++) {
      totalValue += curr->quantity + curr->stock.value;
   }

   return totalValue;
}
```