CS193D Handout 16
Autumn 2000 October 26, 2000

Practice CS193D Midterm

Exam Facts

Thursday evening, November 2, 7 p.m.
TCSeq 200

Alternate times are being offered, but only those who have already contacted me can take the exam at a different time. Please bring pencil and scratch paper, and bring food and drink if you think you'll get hungry.

Format

The exam is closed-book, closed-note, and will be designed as a 60-minute exam. However, you can effectively take as much time as you want to, provided you hand the exam in within 4 hours. It will include questions which test your understanding of the definition and behavior of the C++ features we have studied thus far (e.g. "What does this code do", "How does this work" questions) as well as asking questions which ask you to design and write C++ classes and functions in solution to problems. Writing code on paper is a different experience than interactively compiling and debugging and it is highly recommended that you practice on these sample questions to properly acclimate yourself. The topics covered will be as follows:

Material

The midterm will cover all the course material through next Tuesday's lecture—specifically, you are responsible for understanding all material discussed thus far, including:

- C++ as a better C: `const`, pass by reference, function overloading, default arguments, `new/delete`, `iostreams` (no nitpicky stream stuff though)
- Basic class structures: instance variables, methods, access specifiers, `this`, messaging, `static/const` data and methods, object references
- Constructors, destructors, automatic invocation of constructors & destructors as variables enter and leave scope, constructors and destructors for contained member objects
- Automatic type conversion, constructor conversion, operator conversion
- Methods synthesized by the compiler if not provided: default constructors, copy constructor, `operator=`, destructors
- Memory management with custom copy constructor, `operator=`, and destructors
- Simple inheritance, object slicing, polymorphism, `virtual` methods

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;
```

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)++;
```

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);
   }
}
```

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;
}
```

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();

      void print(ostream& os);
      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 a print method (even though a globally overloaded version `operator<<` would have been better) 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);
microsoftcorp.print(cout);
```

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, and methods. 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.

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.