CS193D Practice Final

Exam Facts

Saturday, December 14th, 3:30 - 6:30 p.m.
Terman Auditorium

Should you need to take the exam at another time, you may do so by downloading the exam from the web page, self-administering it, and then getting the exam back to me before Friday, December 15th at 5 p.m. You may not take the exam early, as it will not be ready prior to the normally scheduled slot.

Format

The exam is closed-book, closed-note, and will be designed as a 120-minute exam. However, you can effectively take as much time as you want to, although we are constrained by the final exam schedule, so in all reality the exam will need to end after 3 hours and 15 minutes. It will include questions which test your understanding of the definition and behavior of the C++ features we have studied (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 final will cover the same material covered on the midterm (see the practice midterm for details there) and also the following.

- Operator overloading, friends functions.
- Defining and using template classes, template functions, template specialization.
- Advanced mechanics of inheritance: subclassing, overriding, overloading, class compatibility (upcasting / downcasting / slicing), constructors / destructors, `static` data / functions, compile-time versus run-time binding, `virtual` methods, polymorphism, abstract base classes, pure `virtual` methods.

Answer Key

An answer key is not being supplied, specifically because I want to you do these problems and compare your answers with others. You are free to ask the staff about any problems you may have, but we will not be supplying code or answers, because doing seems to encourage students to read the answer as they take the exam, and that's silly. You should think of this as an untimed, ungraded exam that doesn't count, and do the proper amount of research and coding to convince yourself that you are right on all of them.

Problem 1: Generic Algorithms (25 points)
The generalized algorithm `rotate` rotates the elements of a range, in place. That is, it moves the element at position `middle` to position `first`, it moves the element at position `middle + 1` to position `first + 1`, and so on. A much simpler way to understand `rotate` is that it exchanges the two ranges `[first, middle)` and `[middle, last)`.

```
template <class ForwardIter>
inline void rotate(ForwardIter first, ForwardIter middle, ForwardIter last)
{
   if (first == middle || middle == last) return;

   ForwardIter leftIter = first;
   ForwardIter rightIter = middle;
   while (true) {
      swap(*leftIter++, *rightIter++);
      if (leftIter == middle && rightIter == last) return;
      if (leftIter == middle)
         middle = rightIter;
      else if (rightIter == last)
         rightIter = middle;
   }
}
```

- Consider the following code block:

```
   char scaleNotes[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
   const int scaleLength = 7;
   rotate(scaleNotes, scaleNotes + 2, scaleNotes + scaleLength);
```

Draw the state of the `scaleNotes` array after exactly **three** full iterations of `rotate`'s `while` loop. A full iteration of the `while` loop includes the character swap and any and all updates to the iterators. Specify the order of the characters, and the positions of each of the five `ForwardIter` iterators.

- In the context of what the `rotate` algorithm is trying to accomplish, briefly explain why `rightIter` is assigned the value of `middle` whenever `rightIter` advances to match the value of `last`.

- Recall that the STL provides a `list` class template, which operates much like a `vector`, except that the underlying implementation is a doubly-linked linear structure, not an array. Explain why:

  - the exchange of two ranges could be supported much more efficiently for the STL `list`, regardless of its element type, by a `rotate` **method** (i.e. `list<T>::rotate`)

  - the `rotate` algorithm cannot be specialized for the STL `list` class so as to take advantage of the embedded doubly-linked structure.

Problem 2: Revisiting the Polynomial Template (31 points)

The notion of an algebraic polynomial is probably a familiar one, and probably has been since eighth grade. Our C++ representation of a polynomial is actually fairly straightforward—we define the `Polynomial` class to encapsulate a `vector` of coefficients, where the coefficient of x^0 is stored at index 0, the coefficient of x^1 is stored at index 1, the coefficient of x^2 is stored at index 2, etc. (Storing the coefficients in reverse order makes addition and multiplication that much easier to support).

For example, the polynomial $\frac{2}{3}x^6 + \frac{2}{9}x^4 - 1\frac{1}{7}x + 4\frac{2}{5}$ would be represented by the `vector`:

$^{22}/_{5}$	$^{-8}/_{7}$	$^{0}/_{1}$	$^{0}/_{1}$	$^{2}/_{9}$	$^{0}/_{1}$	$^{2}/_{3}$

Note that the zeroes really are there in the `vector`; they're holding the place for the intermediate powers of x that just happen to have zero contribution and therefore don't deserve to be shown in the normal textual representation.

Presented here is a reduced interface for the `Polynomial` template class. It's a template, because the programmer should have the flexibility over how they store the coefficients. More specifically, coefficients could be `ints`, `doubles`, `Fractions`, or `complex` numbers, just to name a few, and all of them make perfectly good sense in the context of a polynomial.

```
template <class Arith>
class Polynomial {

   friend ostream& operator<<(ostream& os, const Polynomial<Arith> p);

   public:
      Polynomial() {}
      Polynomial(const vector<Arith>& initCoeffs) : coeffs(initCoeffs) {}

      const Polynomial<Arith>& operator*=(const Arith& factor);
      const Polynomial<Arith>& operator+=(const Polynomial<Arith>& rhs);
      bool operator==(const Polynomial<Arith>& rhs) const;
      const Arith operator()(const Arith& val) const;

   private:
      vector<Arith> coeffs;
};

// provided as an example.
template <class Arith>
bool Polynomial<Arith>::operator==(const Polynomial<Arith>& rhs) const
{
   if (coeffs.size() != rhs.coeffs.size()) return false;
   return equal(coeffs.begin(), coeffs.end(), rhs.coeffs.begin());
}
```

Provide implementations for the `operator*=`, `operator+=`, and `operator()` methods. Read the provided method comments for an adequately detailed description of what each method should do. Use one page for each implementation.

```
/*
 * Method: operator*=
 * ------------------
 * Multiplies the relevant polynomial by
 * the specified factor, and returns the result
 * of the self-multiplication so that cascaded
 * assignments can be supported.  Don't worry about
 * a factor value of zero.
 */

template <class Arith>
const Polynomial<Arith>& Polynomial<Arith>::operator*=(const Arith& factor)
{
```

```
/*
 * Method: operator+=
 * ------------------
 * Adds the specified polynomial to the receiving
 * polynomial object, and returns the result
 * of the self-addition so that cascaded operations
 * can be supported.
 */

template <class Arith>
const Polynomial<Arith>&
Polynomial<Arith>::operator+=(const Polynomial<Arith>& rhs)
{
```

```
/*
 * Method: operator()
 * ------------------
 * Invoking the function call operator evaluates
 * the receiving polynomial at the specified value.
 * Example?
 *    Polynomial<int> parabola(vector<int>(3, 2));
 *    assert(parabola(0) == 2);  // 2 + 2 * (0)^1 + 2 * (0)^2 is 2
 *    assert(parabola(1) == 6);  // 2 + 2 * (1)^1 + 2 * (1)^2 is 6
 *    assert(parabola(2) == 14); // 2 + 2 * (2)^1 + 2 * (2)^2 is 14
 */

template <class Arith>
const Arith Polynomial<Arith>::operator()(const Arith& val) const
{
```

Over the next few pages, provide justified answers to the following series of questions. Each of your responses is expected to be only one or two short sentences, and might include a small but relevant snippet of code. Runaway, wordy, or vague responses will not receive full credit.

- Which of the three methods you justimplemented would compile and run for a variable of type `Polynomial<string>`? Is there any way to prevent such a declaration in the first place?
- Why not make `operator==` a global `friend` function instead of a method?
- Why not templetize the `Polynomial::operator==` method to compare polynomials with distinct but otherwise comparable (in the == sense) coefficient types? What complications arise when you try to support the following?

```
Polynomial<int> myIntPolynomial;
Polynomial<Fraction> myFractPolynomial;
if (myIntPolynomial == myFractPolynomial) { ...
```

- Assume the existence of an `ohtmlfstream` class; it extends the `ofstream` and is designed to support the insertion of text into what will be treated as an HTML file. Without modifying the `ohtmlfstream` class, explain how you could change the interface and implementation of your `Polynomial` template so that polynomials printed to an `ohtmlfstream` could, for the sake of the exponents, take advantage of HTML tagging such as <sup> and </sup>, standard HTML tags delimiting text to be displayed in a smaller font than usual, higher on the line than usual.

Use next the two pages to answer each of these questions.

- Which of the three methods you implemented would compile and run for a variable of type `Polynomial<string>`? Is there any way to prevent such a declaration in the first place?

- Why not make `operator==` a global `friend` function instead of a method?

- Why not templetize the `Polynomial::operator==` method to compare polynomials with distinct but otherwise comparable (in the == sense) coefficient types? What complications arise when you try to support the following?

```
Polynomial<int> myIntPolynomial;
Polynomial<Fraction> myFractPolynomial;
if (myIntPolynomial == myFractPolynomial) { ...
```

- Assume the existence of an `ohtmlfstream` class; `ohtmlfstream` extends the `ofstream` and is designed to support the insertion of text into what will be treated as an HTML file. Without modifying the `ohtmlfstream` class, explain how you could change the interface and implementation of your `Polynomial` template so that polynomials printed to an `ohtmlfstream` could, for the sake of the exponents, take advantage of HTML tagging such as `<sup>` and `</sup>`, standard HTML tags delimiting text to be displayed in a smaller font than usual, higher on the line than usual.

Problem 3: Gustav Holst and You (16 points)

Inspect the following nonsense code where all implementations have been `inlined` for convenience (assume all methods are `public` and all instance methods are `const`):

```
class orion {
   virtual void jupiter() = 0;
   void mars() { cout << "orion::mars" << endl; neptune(); }
   virtual void neptune() { cout << "orion::neptune" << endl; uranus(); }
   virtual void saturn() = 0;
   static void uranus() { cout << "orion::uranus" << endl; }
};

class signus : public orion {
   virtual void jupiter() { cout << "signus::jupiter" << endl;}
   virtual void neptune() { cout << "signus::neptune" << endl; saturn(); }
   static void uranus() { cout << "signus::uranus" << endl; }
};

class vulpecula : public orion {
   virtual void jupiter() { cout << "vulpecula::jupiter" << endl; }
   virtual void neptune() { cout << "vulpecula::neptune" << endl;
                            orion::neptune(); }
   virtual void saturn() { cout << "vulpecula::saturn" << endl; }
   static void uranus() { cout << "vulpecula::uranus" << endl; }
};

class gemini : public signus {
   void mars() { cout << "gemini::mars" << endl; uranus(); }
   virtual void saturn() { cout << "gemini::saturn" << endl; mars(); }
};
```

The `explore` function is designed to take a `const orion` pointer as its only parameter:

```
static void explore(const orion* constellation)
{
   constellation->uranus();
   constellation->mars();
}
```

What are the possible types that `constellation` may be addressing at runtime? For each possibility, trace through a call to `explore` and present its output. Please use the next page for your answers.

Problem 4: Understanding Inheritance (28 points)

Over the next few pages, provide justified answers to the following series of questions. Each of your responses is expected to be only one or two sentences, and might include a small but relevant snippet of code. Runaway, wordy, or vague responses will not receive full credit.

- How is it that the true runtime type (aka dynamic type) of an object can be inferred when sent a message, but not when passed as an argument? (Your response should explain why `Triangle::intersects(Shape&)` would get invoked on behalf of the following code segment.)

```
class Shape {

   public:
      virtual bool intersects(const Shape&) = 0;
};

class Triangle : public Shape {

   public:
      virtual bool intersects(const Shape&);
      virtual bool intersects(const Triangle&);
};

int main()
{
   Triangle yieldSign;
   Shape& sign = yieldSign;
   assert(sign.intersects(sign));
   return 0;
}
```

- Should constructors of abstract base classes always be marked as `private` to emphasize thier inability to be instantiated?
- Why should destructors be marked `virtual` and constructors not?
- Why can't methods be pure (i.e. abstract, `= 0`) without being `virtual`?

Use next the two pages to answer each of these questions.

- How is it that the true runtime type of an object can be inferred when sent a message, but not when passed as an argument?

- Should constructors of abstract base classes always be marked as `private` to emphasize thier inability to be instantiated?

- Why should destructors be marked `virtual` and constructors not?