Autumn CS193D Practice Solution

Problem 1: Generic Algorithms

The manner in which rotate gets its work done is a little obscure, but this version minimizes the number of swaps while resisting the need to allocate temporary storage. A clearly more readable version of the rotate function reads something like the following:

```
template <class RandomAccessIter>
inline void rotate(RandomAccessIter first,
                   RandomAccessIter middle, RandomAccessIter last)
{
   vector<iterator_traits<RandomAccessIter>::value_type> front(first, middle);
   copy(middle, last, first);
   copy(temp.begin(), temp.end(), first + last - middle);
}
```

While this version is compact and clean, it constrains the range being rotated to be laid out sequentially in memory, or at the very least requires iterators that respond to a robust set of operations (in this case, `operator+` and `operator-`).

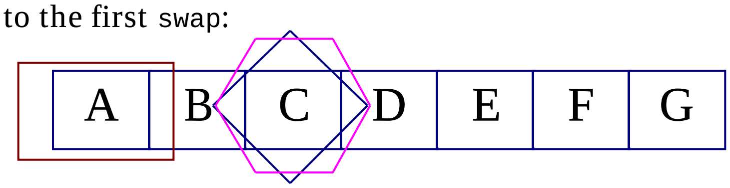
However, the purpose of generalizing rotate in the first place was so that it could apply to as broad a cross section of container classes as possible, not just those that have array-like semantics. The implementation above requires the iterators to be very flexible, but some of the STL containers (`slist` and `list` are the most relevant for this argument) provide iterators that respond to operator++ and operator--, but not operator+ or operator-.

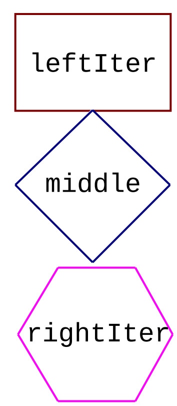
The version given in the exam is the actual Code Warrior Pro 5 implementation (and most likely the standard for all compilers—I'm sure they just lift it from other compilers, since the STL is open source.) Note that its iterators only need to respond to `operator++`, `operator*`, `operator==`, and `operator=`, and these operations are the standard that get supported for all general iterators, including those of `vector`, `slist`, `list`, and `deque`. While the algorithm is more complicated that it need be, the added complexity allows the implementation to do everything in place—that is, no addition storage is required beyond that of a single temp variable (inside a swap routine.) Imagine the memory overhead of the vector version if a range of 20000 elements were rotated around its midpoint.

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

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;
   }
}
```

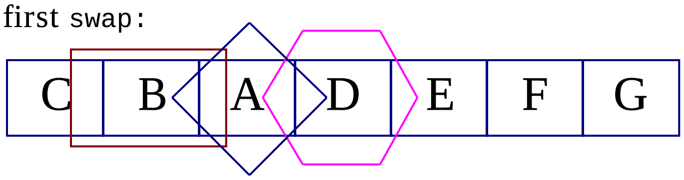
- As far as the trace is concerned, it'll certainly help to see a few intermediate drawing, so I'll present you with a few pictures that you should have caught on your way to the fourth iteration. The values of `first` and `last` never change, and are obvious based on the call. `leftIter`, `rightIter`, and even `middle` change throughout, so a little shape legend (with color if you look at the PDF) is provided on the right. Happy colors.

- Just prior to the first `swap`:

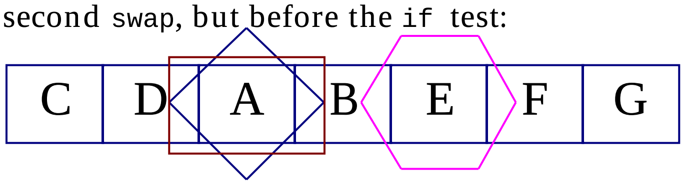




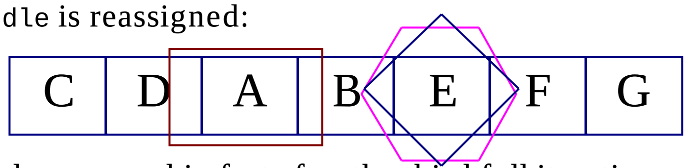
- Just after the first `swap`:



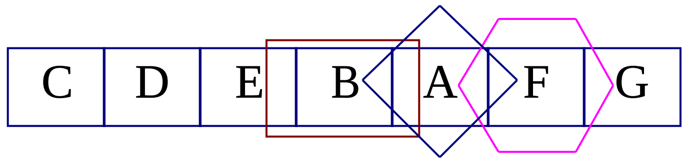
- Just after the second `swap`, but before the `if` test:



- Just after `middle` is reassigned:



- After the third `swap`, and in fact after the third full iteration.



Effectively, the AB sequence is moved as a unit toward the back of the sequence until pressed against the right end.

- 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`.

  When the first of the two ranges being exchanged is wider than the second, `rightIter` will advance to match last before `leftIter` will advance to match `middle`. Effectively, `rightIter` needs to wrap around to the front so that the swaps of the rotation can continue uninterrupted. However, the element at position `first` was previously swapped with that at position `middle`, so `rightIter` needs to be set to `middle` instead. That's where the element initially at position `first` now resides.

- 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`)

    Provided the ranges to be swapped are delimited by three iterators as in the global STL rotate algorithm, the rotation can be implemented through a series of constant-time pointer operations. All next and previous pointers requiring an update are accessible from the three iterators expressed as parameters.

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

Iterators know what they point to, but they have no way of knowing what container type they point into. There is no way for the rotate algorithm, given three iterators, to learn if the elements being referenced are contained in a list, so there's no sense in trying to specialize the algorithm based on a container type that the iterators just can't see.

Problem 2: Revisiting the Polynomial Template (31 points)

This problem was my favorite on the exam. The coding was likely straightforward for most of you, but the short answers and the lessons them are really the take-home points here.

First things first, let's get right down to the coding. The implementation of the `operator*=` method was fairly straightforward, particularly in light of the case I wasn't holding you accountable for multiplication by zero as a special case. My expectation was that you do brute force iteration, either via a traditional for loop, or the less traditional but more C++'ey one using iterators.

```
template <class Arith>
const Polynomial<Arith>& Polynomial<Arith>::operator*=(const Arith& factor)
{
   if (factor == Arith(0)) coeffs.clear();  // optional
   for (int power = 0; power < coeffs.size(); power++)
      coeffs[power] *= factor;

   return *this;
}

template <class Arith>
const Polynomial<Arith>& Polynomial<Arith>::operator*=(const Arith& factor)
{
   if (factor == Arith(0)) coeffs.clear();  // optional
   for (vector<Arith>::iterator iter = coeffs.begin();
        iter != coeffs.size(); ++iter)
      *iter *= factor;

   return *this;
}
```

For those of you who just happen to know your STL, you could map a `multiplies<Arith>` functional object using `transform`, where the second argument to the function call operator is always the specified `factor`. You bind the factor to the second argument using the `bind2nd` function adaptor, which renders the binary function call operator of `multiplies<Arith>` to be a unary function that applies the scaling by `factor` to every element between the delimiting iterators passed to `transform`. Note that the destination sequence can be that of another container, but we want an in place transformation, so we pass `coeffs.begin()` as `transform's` third argument.

```
template <class Arith>
const Polynomial<Arith>& Polynomial<Arith>::operator*=(const Arith& factor)
{
   if (factor == Arith(0)) coeffs.clear();  // optional
   transform(coeffs.begin(), coeffs.end(),
             coeffs.begin(), bind2nd(multiplies<Arith>(), factor));
   return *this;
}
```

And then there was that operator+= method, which isn't really any more complicated than the operator*=, except that self-multiplication was against a constant, whereas self-addition is against a polynomial. My expectation, again, was the use of a standard for loop around C-like semantics, though for loop around an iterator will certainly get a smiley face, and the use of transform and function objects will get a huge smiley face and a little note in my spreadsheet.

```
template <class Arith>
const Polynomial<Arith>&
Polynomial<Arith>::operator+=(const Polynomial<Arith>& rhs)
{
   int power;
   for (power = 0; power < min(coeffs.size(), rhs.coeffs.size()); power++)
      coeffs[power] += rhs.coeffs[power];

   if (power < rhs.coefficients.size())
      coeffs.insert(coeffs.end(),
                    rhs.coeffs.begin() + power,
                    rhs.coeffs.end());

   return *this;
}
```

Man oh boy oh man did this operator rock my world. The problem wasn't quite as easy as it first looks, for while it technically is a running sum, you needed an efficient yet general way to accurately compute what the successive integral powers of val were. An explicit for loop over the coefficients was acceptable, and in fact the way I expected everyone to write this.

```
template <class Arith>
const Arith Polynomial<Arith>::operator()(const Arith& val) const
{
   Arith result;     // assume it defaults to additive identity of 0
   Arith runningPower = 1;
   for (int power = 0; power < coeffs.size(); power++) {
      result += coeffs[power] * runningPower;
      runningPower *= val;  // next time it'll be val^(power + 1);
   }

   return result;
}
```

We could have included some shortcuts when val equaled zero, or when the polynomial itself was zero (presumably it's coeffs vector should be empty), but this captures the general algorithm and still works for both those special cases.

Kudos to anyone who wrote his or her own function object and used accumulate to do this. The only reason I knew to do this is because I love and abuse the accumulate template to death anyway. A proper understanding of how accumulate works is all it takes to appreciate the implementation below. Note that the function call operator almost replicates the body of the for loop above. The only reason it needs to be a function object instead of a function pointer is so that `val` can be maintained as function state.

```
template <class Arith>
struct PolynomialEval
{
   PolynomialEval(const Arith& val) : base(val), power(1) {}
   Arith operator()(Arith resultSoFar, const Arith& next)
   {
      resultSoFar += (next * power);
      power *= base;
      return resultSoFar;
   }

   const Arith base;
   Arith power;
};

template <class Arith>
const Arith Polynomial<Arith>::operator()(const Arith& val) const
{
   return accumulate(coeffs.begin(), coeffs.end(),
                     0, PolynomialEval<Arith>(val));
}
```

i. Which of the three methods you just 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?

   `operator*=` can't be expanded for a `string` coefficient, since `operator*=` is not part of the `string` class (so `coeffs[power] *= factor` would choke `gcc`). `operator+=` isn't a problem, since `string::operator+=` is supported and defined to mean self-concatenation. The function call operator couldn't do it, because binary `operator*` isn't defined for the `string` class. So operator+= would compile and even run, whereas operator*= and operator() just can't expand for a string, because the notion of multiplication isn't supported.

   There's no way to prevent a declaration, because a declaration calls and therefore only explands code for the specific constructor. Methods are expanded on an as-needed basis, so if only a constructor (which either initializes or clones a vector) and operator+= are used, everything can compile and run just fine.

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

   We could, but we do so only when attempting to combine automatic conversion and symmetry. Here, the need to convert an `Arith` to a `Polynomial<Arith>` isn't exactly huge; at least that's the impression you must take from my decision to make it a method. In general, we only use `friend` functions over instance methods when the function really buys us something we can't get in method form. That's just not the case here.

iii. 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?

There's nothing intrinsically wrong with a template within a template. The relevant portion of the `Polynomial` class could have been updated so that the `operator==` accepts all types of `Polynomial` instantiations, not just that type receiving the message.

```
template <class Arith>
class Polynomial {

   public:

      bool operator==(const Polynomial<Arith>& rhs) const;
};
```

could have been

```
template <class Arith>
class Polynomial {

   public:

      template <class AnotherArith>
      bool operator==(const Polynomial<AnotherArith>& rhs) const;
};
```

where the implementation is precisely the same:

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

Note that `Arith` and `AnotherArith` can be bound to the same type without a problem, so code that worked before works now. However, when `Arith` and `AnotherArith` are different types, `Polynomial<Arith>` and `Polynomial<AnotherArith>` are different types as well—to the compiler, they're as distinct from one another and `int`s and `multimap`s. That being the case, a method of the `Polynomial<Arith>` class isn't permitted to access the `coeffs vector` of the `Polynomial<AnotherArith> rhs`: `coeffs` is `Polynomial<AnotherArith>`'s `private` data, and a `Polynomial<Arith>` can't touch it. The two classes are automatically `friends` even though both were generated from the same template, because at some point the compiler forgets they were ever templates at all. There's no general way to express that all version of a template class are `friends` of each other, so there's no way to access the `coeffs vector` without some accessor methods being included.

This was a very subtle problem, so I wouldn't be crushed if your answer came out to be different.

iv. 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.

Very simple answer here. Introduce a `friend` version of `operator<<` specific to `ohtmlfstreams`. This way, the insertion of a `Polynomial<Arith>` into an `ohtmlfstream` prompts the invocation and eventual execution of that specific functionality—functionality which can include code to generate the `<sub>` and `</sub>` tags around the exponents.

Problem 3: Gustav Holst and You (16 points)

The orion class is abstract, and therefore no bona fide instance of the orion class could exists at runtime. Any attempt to construct just a plain old orion will be categorically rejected by the compiler, since even the compiler knows that it's an incomplete class. Not quite as obvious, at least at first glance, is that the `signus` class is also abstract, because it inherits the pure `virtual neptune` method without implementing it. `vulpecula` implements both the `neptune` and `jupiter` method, so it can be instantiated. Same goes for `gemini`, which inherits `jupiter` from `signus` and clears the pure `virtual` of a `neptune`.

The consensus: `constellation`, even if statically of type `const orion *`, can only point to a `vulpecula` or a `gemini`. The other two types couldn't be instantiated in the first place.

So what does explore do in each of these two cases? Of course, it depends whether constellation addresses a `vulpecula` or a `gemini` object.

When `constellation` points to a `vulpecula`, we see the following:

```
orion::uranus
orion::mars
vulpecula::neptune
orion::neptune
orion::uranus
```

When `constellation` points to a `gemini`, we see:

```
orion::uranus
orion::mars
signus::neptune
gemini::saturn
gemini::mars
signus::uranus
```

Problem 4: Understanding Inheritance (28 points)

i. 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?

   Well, the compile-time type of each is specified with the declaration type pf the reference or the pointer, not by the declaration type of the orignal object. When a virtual message is sent to a pointer or reference, the vtable attached to the object at construction time provides a pointer to the most specific version that can apply. The runtime binding of the method **emulates** the runtime determination of the dynamic type.

ii. Should constructors of abstract base classes always be marked as `private` to emphasize their inability to be instantiated?

   No, they should be marked as anything but `private`. Private constructors are off limits to everyone, including subclasses. If the base class is truly abstract, no standalone instance of the class can be instantiated regardless of the access specifiers marking its constructors. The damage comes when subclasses try to define their own constructors. Subclass constructors, in need to call some superclass constructor, are at a loss, because all of them are `private`, and even subclasses need to respect such privacy. Bummer for the subclasses.

iii. Why should destructors be marked `virtual` and constructors not?

   Destructors should be `virtual` so that the most specialized cleanup routine can be called on behalf of a deletion. Constructors can't be `virtual`, because objects are constructed according to the declaration type of the object—and that typename is ultra-clear about what constructor should be called.

iv. Why can't methods be pure (i.e. abstract, `= 0`) without being `virtual`?

   The omission of virtual implied compile-time determination of which version of a method to call. If runtime lookup isn't an option for the invocation of an abstract method, then there's really no point in having placed it in the parent class in the first place.