

COMP322 - Introduction to C++

Lecture 11 - Templates and defining our own iterators

Dan Pomerantz

School of Computer Science

26 March 2012

Announcements

- ▶ Assignment four posted. Due final day of classes.
- ▶ Quiz 2 next week (possibly full period possibly not)
- ▶ No class week after (cancelled due to Easter Monday and McGill being closed).

Defining our own iterators.

- ▶ Suppose we have defined a type that stores a bunch of data using a more complex type than simply one vector.
- ▶ Or suppose we have a vector but we want to iterate over it in an unusual order, or skip certain values.
- ▶ We want to be able to use the algorithms defined in `algorithms.h` to work with our data.
- ▶ All of these algorithms require iterators passed as input.
- ▶ So we need to be able to define an iterator to our collection.

Those weird types.....

- ▶ Remember the following code?

```
for (vector<int>::iterator current = v.begin();  
     current != v.end();  
     current++)
```

- ▶ `vector<int>::iterator` is the *type* of the iterator.
- ▶ It is a specialized type that has operators such as `==` , `=` , `!=`, `++`, `*` defined on it.
- ▶ In the case of a vector, we are given a type that has `+` as well as `--` and `-` defined on it.
- ▶ Recall that which operations are and aren't defined on an iterator type determines whether we call it a forward, backwards, random access, read only, write only iterator.

Defining an iterator type

- ▶ If we want our class to be usable with the algorithms in `algorithms.h`, we need to define an iterator type that works on it.
- ▶ Suppose we have a class that stores a `vector<int>`. We want to define three iterators: One which traverses over ALL values, one which traverses just the positive values, and one which traverses the negative values.
- ▶ For traversing over ALL values, it is easy: we can just call the vector `begin()` and `end()`
- ▶ But for traversing over positive/negative we need to define our own iterator.

What operations should we support?

- ▶ We need to determine what kind of iterator we should support.
- ▶ Good rule of thumb: Don't provide more features than is necessary AND fast.
- ▶ For example: random access is SLOW in this case because we need to move an arbitrary amount of steps. This means that sort would be very slow.
- ▶ Remember that sort is a $O(n\log(n))$ method to begin with. But that assumes random access.
- ▶ In our case, addition means finding n positive or negative values past the current place.
- ▶ This takes $O(n)$ time possibly, meaning sorting actually takes $O(n^2\log(n))$

Defining ++

- ▶ Defining ++ (or --) does not have this problem.
- ▶ Someone will use it to traverse over the entire container one at a time.
- ▶ They expect this would be linear with respect to the total number of elements.

Defining our new type

- ▶ We define a new type `PosOrNegVectorIterator`
- ▶ This type will have defined on it all operations necessary to traverse a `Vector` by choosing the right values.
- ▶ What does this type need to store to do its computations.

Storage for our type:

- ▶ Our type needs to store three things:
- ▶ A current iterator for the spot of the Vector we are currently at.
- ▶ An end iterator (one past the end) for the Vector we are traversing.
- ▶ A bool representing whether the `PosOrNegVectorIterator` should choose all positive or all negative values. If we wanted to make this iterator more general, we could take as input a boolean function pointer or an object (like the algorithm functions do) which specified the selection mechanism.

Start of our type

- ▶ Our type needs to store three things:
- ▶ A current iterator for the spot of the Vector we are currently at.
- ▶ An end iterator (one past the end) for the Vector we are traversing.
- ▶ A bool representing whether the `PosOrNegVectorIterator` should choose all positive or all negative values. If we wanted to make this iterator more general, we could take as input a boolean function pointer or an object (like the algorithm functions do) which specified the selection mechanism.

Start of our type

```
class PosOrNegVectorIterator
{
private :
    vector<int>::iterator current;
    vector<int>::iterator end;
    bool isPositive;
};
```

Initializing our iterator type

- ▶ Next we need to add a constructor. The constructor should take as input these three things:

```
PosOrNegVectorIterator(vector<int>::iterator start,  
                        vector<int>::iterator endV, bool positive)  
{  
    current = start;  
    end = endV;  
    isPositive = positive;  
}
```

- ▶ This will allow us to do something like
`PosOrNegVectorIterator positive(v.begin(),
v.end(), true)` inside a method such as
`positiveBegin()` in our class

Overloading operators

- ▶ Next we need to overload the necessary operators. For starters, let's overload the assignment operator.

```
PosOrNegVectorIterator& operator=(const  
    PosOrNegVectorIterator& other)  
{  
    current = other.current;  
    end = other.end;  
    isPositive = other.isPositive;  
}
```

- ▶ This will allow us to do something like
`PosOrNegVectorIterator current =
collection.positiveBegin()`

Overloading operators

- ▶ `==` and `!=` are fairly straightforward to do although the syntax is a bit messy. What about `++`. Note that there are actually two `++` (pre and post fix)

```
PosOrNegVectorIterator& operator++()  
//same header except put operator++(int) for post-fix  
{  
    current++;  
    int factor = isPositive ? 1 : -1;  
    while ((*current) * factor < 0 && current != end)  
    {  
        current++;  
    }  
  
    return *this;  
}
```

- ▶ This will allow us to update our iterator.

Dereference operator

- ▶ Another operator we need to define is the `*` operator. This is actually surprisingly easy.

```
int& operator*()
{
    return(*current);
}
```

- ▶ This will allow us to update our iterator.

Using this new type

- ▶ To use this new type now, we need to write a `positiveBegin` and `negativeBegin` method inside our class. Suppose our vector of data is called `data`

```
PosOrNegIterator positiveBegin() {  
    return PosOrNegIterator(data.begin(), data.end(), true);  
}
```

```
PosOrNegIterator positiveEnd() {  
    //think of what the value of the iterator would be when we  
    //reach the end.  
    return PosOrNegIterator(data.end(), data.end(), true);  
}
```

Good exercise

- ▶ Define a vector class yourself as if it weren't present.
- ▶ You'll need to write methods such as `push_back`, `begin()`, and `end()` which involves defining your own iterator type.
- ▶ You'll also need to be able to make your type take as "input" any other type. Which brings us to.....

C++ and abstraction

- ▶ Ideally, we want to express any non-trivial concept exactly once.
- ▶ But consider a simple stack class:

```
class Stack {  
private:  
    float *storage;  
    int max;  
    int top;  
public:  
    float pop();  
    void push(float);  
};
```

- ▶ Our interface specifies the data type stored in the stack.
- ▶ What happens when we want a stack of ints? Or pointers? Or some other object?

C++ and abstraction

- ▶ The same idea might apply to algorithms that are not naturally part of a class, such as a generic sorting function.
- ▶ What we'd like is a way to express the idea of an algorithm or data structure independently from the specific type it is to use.
- ▶ We could view the type of the stored object as one of the parameters of the function or class, and automatically apply the code for each case.
- ▶ This is exactly what C++ *templates* do!

Class templates

- ▶ Templates can apply to classes or functions.
- ▶ We saw in assignment two how this was applied to functions. (Could use multiple kinds of iterators)
- ▶ We can use templates to declare our stack class:

```
template <class T> class Stack {  
private:  
    T *storage;  
    int max;  
    int top;  
public:  
    Stack(int n = 100);  
    ~Stack();  
    T pop();  
    void push(T);  
};
```

Using a templated class

- ▶ We can use the template to create a stack of string objects:

```
int main() {  
    Stack<string> sstack;  
    sstack.push("world");  
    sstack.push("hello");  
    cout << sstack.pop(); // Print hello  
}
```

Defining template members

- It's easiest to define member functions in the class:

```
template <class T> class Stack {  
private:  
    T *storage;  
    int max;  
    int top;  
public:  
    Stack(int n = 100) {  
        storage = new T[n];  
        max = n;  
        top = -1;  
    }  
    ~Stack() {  
        delete [] storage;  
    }  
    T pop() {  
        if (top >= 0) {  
            return storage[top--];  
        }  
    }  
    // ...  
};
```

Defining template members

- ▶ Alternatively, member functions can be defined outside the class. This adds a bit of extra boilerplate to each definition:

```
template <class T> Stack<T>::Stack(int n) {  
    storage = new T[n];  
    max = n;  
    top = -1;  
}  
  
template <class T> Stack<T>::~~Stack() {  
    delete [] storage;  
}  
  
template <class T> void Stack<T>::push(T v) {  
    if (top < max - 1) {  
        storage[++top] = v;  
    }  
}
```

Template parameters

- ▶ While the keyword `class` is used, the parameter can be any name that is defined as a type.
- ▶ Parameters may also include integer constants of another type, or another template.

```
// One or more type parameters  
template <class T, class U> class C { /* ... */ };  
// An integer and type parameter  
template <int X, class Y> class D { /* ... */ };  
// A type parameter may define the type of another parameter  
template <class T, T def> class E { /* ... */ };  
// Pass a template as an argument  
template <class B, template<class> class C> class F {  
    C<B> inst1;  
    C<B *> inst2;  
    // ...  
};
```

- ▶ Parameter names do *not* have to be a single letter, but this is a common idiom.

Type equivalence

- ▶ A class template may create many distinct types:

```
int main() {  
    Stack<int> is;  
    Stack<float> fs;  
  
    // Stack<int> and Stack<float> are not assignment compatible!  
    is = fs; // Error!  
}
```

- ▶ However, if the template arguments are effectively identical, the types are compatible:

```
typedef unsigned char uchar_t;  
Stack<uchar_t> us1;  
Stack<unsigned char> us2;  
  
us1 = us2; // OK!  
  
SomeTemplate<int, 10> t1;  
SomeTemplate<int, 20-10> t2;  
  
t2 = t1; // OK, constant expressions equivalent
```

Class template instantiation

- ▶ Actual use of a template is sometimes referred to as template *instantiation*.
- ▶ Template code for a specific set of parameters is generated on demand:

```
int main() {  
    Stack<int> istack; // Generate code for integer stack  
    Stack<string> sstack; // Generate code for string stack  
    // ...  
};
```

- ▶ No code is generated for unused template parameter choices.
- ▶ This has important implications for libraries, and error checking.

Type parameter validity

- ▶ Any type may be passed as a template parameter, but it has to support the operations assumed by the template:

```
template <class B> class C {  
    B x;  
    B y;  
public:  
    B f() { return x + y; } // Addition must be defined on 2 Bs!  
};
```

```
template <class D> class E {  
    D x;  
public:  
    void update(int n) {  
        x.g(n); // D must include a member function "g"  
    }  
};
```

- ▶ Some errors can be caught only when the template is instantiated!

Notes about class templates

- ▶ Classes are generated from templates as requested.
- ▶ Template expansion occurs at compile time.
- ▶ Each generated class has its own copy of any static data.

Function templates

- ▶ We can define global functions using templates as well:

```
template <class T> void sort(vector<T> &v) { // definition
    const size_t n = v.size();
    for (int gap = n / 2; 0 < gap; gap /= 2)
        // ...
        if (v[j+gap] < v[j]) { // swap
            T temp = v[j];
            v[j] = v[j+gap];
            v[j+gap] = temp;
        } else break;
}
```

- ▶ The type of the arguments determines the version that is instantiated and called:

```
template <class T> void sort(vector<T> &);

void f(vector<int> &vi, vector<string> &vs) {
    sort(vi); // sort(vector<int> &);
    sort(vs); // sort(vector<string> &);
}
```

Function template arguments

- ▶ The choice of function template may be deduced from the parameters:

```
template <class T, int max> T& lookup(Buffer<T, max> & b,  
                                     const char *p);
```

```
class Record {  
    const char v[12];  
    // ...  
};
```

```
Record & f(Buffer<Record, 128> &buf, const char *p) {  
    return lookup(buf, p); // T is Record & max is 128  
}
```

- ▶ However, if the template argument can't be deduced, we need to provide it explicitly:

```
template<class T> T *create(); // Create a T  
  
void f() {  
    int *p = create<int>(); // function, template argument 'int'  
}
```

Source code issues

- ▶ By default, the full template definition must be accessible from any compilation unit (source file) that uses it.
- ▶ Often, this means the entire template definition is placed in a “.h” file.
- ▶ This may expose the implementation, or require extra information to be included during compilation.

Source code issues - export

- ▶ Alternatively, we can mark the template explicitly for export:

```
// min.h  
template <class T> T min(T, T);
```

```
// min.cpp  
export template <class T> T min(T x, T y) {  
    return (x < y) ? x : y;  
}
```

```
// client.cpp  
#include "min.h"  
  
// use min() as needed
```

- ▶ However, export is not implemented in many compilers.

Template specialization

- ▶ Each time you instantiate a templated class of a different type, a new version is created.
- ▶ The result can get large and complicated if we have to instantiate the template for many different types.
- ▶ Template specialization exists to minimize this.

Specialization example

- ▶ Consider a generic Vector class, which can any number of objects of any type:

```
template <class T> class Vector {  
    T *v;  
    int length;  
public:  
    Vector();  
    explicit Vector(int);  
    T & operator[](int i);  
    // ...  
};
```

- ▶ We can specialize this template for a specific type:

```
template <> class Vector<bool> {  
    // code for boolean bit vectors  
};
```

- ▶ We remove any parameters with fixed values from the parameter list.

Typename keyword

- ▶ Historically, C++ re-uses the `class` keyword to declare template parameters which may be any type.
- ▶ Arguably, this is confusing.
- ▶ More recent implementations have added the `typename` keyword to address this confusion:

```
template <typename T> class A {  
    T *data;  
    int sz;  
public:  
    /* ... */  
}
```

Templates and inheritance

- Inheritance is not preserved across templates:

```
template <typename T> class A { /* ... */ };  
class B { /* ... */ };  
class C : public B { /* ... */ };  
  
int main() {  
    B b;  
    C c;  
    A<B> ab;  
    A<C> ac;  
    b = c;    // legal, as B is derived from C  
    ab = ac;  // error!  
}
```

Deriving templates from templates

- ▶ We can derive a class template from another template.
- ▶ Normally the template parameter will be used as the parameter of the base class:

```
template<class T> class A { /* ... */ };  
template<class T> class B: public A<T> { /* ... */ };
```

- ▶ A range of situations are possible:

```
template <class T> class C { /* ... */ };  
template <class T> class D : public C< D<T> > { /* ... */ };
```

- ▶ Or we could inherit from two different template classes:

```
template <class T> class E { /* ... */ };  
template <class T> class F { /* ... */ };  
template <class T, class X> class G : public E<T>, public F<X> {  
};
```

Member templates

- ▶ A class or class template can contain templates as members:

```
template <typename T> class A {  
    // ...  
public:  
    template <typename X> A(X &arg);  
    // ...  
};
```

- ▶ This syntax would allow us to construct an A from an object of an arbitrary type - although presumably a type with some well-known set of operations.