NUMERICAL RECIPES
Webnote No. 1, Rev. 1

Using Other Vector and Matrix Libraries

Suppose you have a vector class `MyVec` that you want to use instead of `NRvector`. If `MyVec` has functionality and syntax identical to `NRvector`, then simple redefinition of `NRvector` by a preprocessor macro will do (as is the case for substituting the `vector<>` class). But suppose this is not the case. The goal, then, is to have automatic type conversions when a `MyVec` is passed to a *Numerical Recipes* function (and an `NRvector` is expected), and when an `NRvector` is returned by a Recipe function (and a `MyVec` is expected). The C++ language supplies us with the necessary tools: a *constructor conversion* to make an `NRvector` out of a `MyVec`, and an *operator conversion* to make a `MyVec` out of an `NRvector`.

The connection with the discussion on `const` correctness, above, is that automatic type conversion in C++ always results in `const` objects. In the alternative (one-line) form for `operator[]` discussed above, all invocations of `operator[]` will enforce constness of the object (though not, now, on its data). Although giving up the extra `const` checking is regrettable, this nevertheless turns out to be the best route to allowing transparent use of other matrix/vector libraries, as we now explain.

The first key idea is to redefine `NRvector` as a "wrapper class" for `MyVec`, that is, an `NRvector` "holds-a" `MyVec`. The second key idea is to have the wrapper class hold both a `MyVec` *and* a reference to a `MyVec`. Then when we want to make an `NRvector` out of a `MyVec`, we can use the constructor initializer list to point the reference to the existing `MyVec`. No copying of data takes place at all.

The wrapper classes must provide all the functions of the default classes introduced earlier, and we shall describe them in the same order. So here is the start of the `NRvector` wrapper class:

```
template<class T>
class NRvector {
    private:
        MyVec<T> myvec;
        MyVec<T> &myref;
    public:
        NRvector<T>() : myvec(), myref(myvec) {}
            ...
```

The private variables are only a `MyVec` and a reference to one. The private variable `myvec` is used solely when creating a new `NRvector`. The default constructor just invokes the default constructor for a `MyVec`, and then points the reference accordingly. Similarly, the remaining constructors simply invoke the corresponding `MyVec` constructors:

```
    explicit NRvector<T>(const int n) : myvec(n), myref(myvec) {}
    NRvector<T>(int n, const T &a) : myvec(n,a), myref(myvec) {}
    NRvector<T>(int n, const T *a) : myvec(n,a), myref(myvec) {}
```

Note we are assuming that the syntax of the MyVec class member functions is the same as that of the NRvector class, but it's easy to take care of different syntax. For example, some vector classes expect arguments in the opposite order: myvec(a,n) instead of myvec(n,a).

Next comes the conversion constructor. It makes a special NRvector that points to MyVec's data, taking care of MyVec actual arguments passed as NRvector formal arguments in function calls:

```
NRvector<T>(MyVec<T> &rhs) : myref(rhs) {}
```

Since all the other functions in NRvector access the object only through myref, there is no need to initialize myvec.

The copy constructor and assignment operator simply call the MyVec copy constructor and assignment operator:

```
NRvector(const NRvector<T> &rhs) : myvec(rhs.myref), myref(myvec) {}
inline NRvector& operator=(const NRvector &rhs)
   { myref=rhs.myref; return *this;}
```

We assume here that the MyVec functions do the sensible thing of making a "deep" copy, that is, they copy the data, not just the reference. If the copy is "shallow," that is, only the reference, then one needs to copy the actual data. If the library's function for doing this is called copy, then for example the copy constructor would be replaced by the following:

```
NRvector(const NRvector<T> &rhs) : myvec(rhs.myref.size()), myref(myvec)
   {copy(rhs.myref,myref);}
```

and similarly for the assignment operator.

In the same way, functions like size(), resize() and assign() are implemented by calling the corresponding MyVec functions:

```
inline int size() const {return myref.size();}
```

We pointed out earlier that for the subscript operator, only the form that guarantees constness of the container is allowed. We want automatic type conversion, and we get that only for objects passed by const reference. Accordingly, we cannot guarantee constness of the data (see the discussion in the previous subsection):

```
inline T & operator[](const int i) const {return myref[i];}
```

Next comes the conversion operator from NRvector to MyVec, which handles NRvector function return types when used in MyVec expressions:

```
inline operator MyVec<T>() const {return myref;}
```

Finally, the destructor is trivial: the MyVec destructor takes care of destroying the data.

```
~NRvector() {}
```

A wrapper class for NRmatrix follows exactly the same form as for NRvector. The only thing to watch out for is to make sure that the return type for operator[] is whatever a MyMat object returns for a single [] dereference. Then expressions like a[i][j] will work correctly.

Using the above guide, you should be able to write a wrapper class to interface between the Numerical Recipes functions and classes written using some other matrix/vector library. Typically, the overhead for using a wrapper class is small, less than 10% compared with using the library directly. Popular public domain libraries include the Template Numerical Toolkit, or TNT [1], the Matrix Template Library, or MTL [2], the uBLAS library (part of the Boost libraries) [3], and the Blitz++ library. [4] Because these and similar libraries evolve with time, we have not provided implementations of the wrapper classes. However, any reader who writes such a class is welcome to send it to us to be posted on our web site for others to use.

One final instruction: the type declarations in the file nr3.h also need to be changed to the "const protects the container, not the contents" convention for passing arguments by reference. *All* the typedef declarations, whether _I, _O, _IO or plain, must have const.

A note for C++ aficionados: You can also implement the interface to other matrix/vector libraries by making `NRvector` a derived class of your vector class. However, this is not nearly as elegant as the wrapper class. In particular, it depends on the implementations inside of your vector class, while the wrapper class uses only the public interface and semantics of your vector class.

CITED REFERENCES AND FURTHER READING:

Pozo, R., *Template Numerical Toolkit*, `http://math.nist.gov/tnt`.[1]

Lumsdaine, A., and Siek, J. 1998, *The Matrix Template Library*, `http://www.lsc.nd.edu/research/mtl`.[2]

Information on uBLAS at `http://www.boost.org/libs/numeric/ublas`.[3]

Information on Blitz++ at `http://www.oonumerics.org/blitz`.[4]