RC 20192 (09/19/95)
Computer Science

IBM Research Report

Scientific and Engineering C++: Two Faces of Polymorphism

John J. Barton, Lee R. Nackman

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Scientific and Engineering C++: Two Faces of Polymorphism

John J. Barton Lee R. Nackman

September 29, 1995

In this column we compare two forms of polymorphism side-by-side for a problem that either form can tackle. The template form uses common names in similar classes; the virtual function form uses common function prototypes in base classes. The problem requires neither accessing elements of collections through base class pointers—a forte of virtual functions—nor does it have many small objects that need efficient random access—a forte of templates. The choice between virtual function and template polymorphism being non-obvious, it is instructive to compare these two forms of polymorphism on the one problem.

The problem is to generalize the LAPACK wrapper classes we've discussed in our last two columns [1, 2]. From a software design point of view, our wrapper classes are usability coating on a high performance core provided by LAPACK's FORTRAN code. We won't discuss the core here; we want to focus on the software, not the math. Overall the problem we tackle has two input objects, a matrix of equation coefficients A and a matrix of equation results b. The output is a matrix of solution values, x, where $Ax = b$. The method of solution is to factor A into matrices that are easily inverted, e.g., $A = LU$, then to solve the factored equations, e.g., $x = U^{-1}L^{-1}b$.

To obtain an easy-to-use library without compromising performance we must solve two problems. First, we need to support in-place factorization—the storage for L and U must overwrite the storage of A for the case of $A = LU$—while simultaneously checking that the storage is not accessed as if it still held A. Second, we need to support LAPACK's many special storage formats for the matrix A because they allow faster and more accurate equation solutions.

We solved the first problem in our preceding column with a smart-reference class something like this:

```
template<class EltT>
class LapackRect {
public:
  // Array behavior not shown...
  FactoredLapackRect<EltT> factor();
private:
  ConcreteFortranArray2d<EltT>* ap;
};
```

The class acts like a 2D array, but each operation checks the validity of the pointer `ap`. The `factor()` member function hands off the storage to which `ap` points to the class `FactoredLapackRect<EltT>` and invalidates `ap`. Since we do all of the serious computations under the covers, `LapackRect<EltT>` adds safety checks and memory management without compromising performance.

The Template Face of Polymorphism. To generalize the smart-reference class to other kinds of storage we can simply add a template argument to `LapackRect<EltT>` for the type of the storage. However, we are going to go a bit further and take out the `factor()` member function so that we get an abstraction independent of matrix factoring:

```
template<class SharedStorageT, class EltT>
class ArrayInSharedStorage {
public:
   // A pointer-management class with Array behavior.

  ArrayInSharedStorage (int nrows, int ncols) :
     ap(new SharedStorageT(nrows, ncols)) {}
  ArrayInSharedStorage (
```

```
    const ArrayInSharedStorage<SharedStorageT, EltT>& a);
  void operator=(
    const ArrayInSharedStorage<SharedStorageT, EltT>&);
  ~ArrayInSharedStorage() { delete ap; }

  // Array behavior
  EltT& operator()(int i, int j)
      { if (!ap) error(); return (*ap)(i, j); }
 int nrows() const
      { if (!ap) error(); return ap->nrows(); }
  int ncols() const
      { if (!ap) error(); return ap->ncols(); }
  // ...

  // Pointer management behavior
  SharedStorageT* release();
private:
  void error() const;

  SharedStorageT* ap;
};
```

This class is a 2D array implemented by forwarding to a pointer; the array elements are stored in the format given by the `SharedStorageT` template argument. The storage is released and the pointer invalidated when `release()` is called:

```
template<class SharedStorageT, class T>
SharedStorageT*
ArrayInSharedStorage<SharedStorageT, T>::release() {
   if (!ap) error();
   SharedStorageT* p = ap;
   ap = 0;
   return p;
}
```

We can use this template anytime we need a shareable 2D array with checked operations.

Here, we use the template to perform in-place factorization. The factored matrix object is not a matrix, just a collection of factors. Its only interesting behavior is a `solve()` member function; it will look something like this for simple rectangular storage:

```
template<class SharedStorageT, class EltT>
class FactoredInPlace {};

class FactoredInPlace<
            ConcreteFortranArray2d<float>, float>
   {
public:
   typedef ConcreteFortranArray2d<float> SharedStorageT;
   FactoredInPlace(SharedStorageT* mp) : ap(mp),
      pivots(min(ap->nrows(), ap->ncols())) {
      // Call LAPACK factoring routine here
   }

   FactoredInPlace(
     const FactoredInPlace<SharedStorageT, float>& a)
       : ap(new SharedStorageT(*a.ap)), pivots(a.pivots) {}

   ~FactoredInPlace() { delete ap; }

   ConcreteFortranArray2d<float>&
        solve(ConcreteFortranArray2d<float>& b) const;
private:
   SharedStorageT*                ap;
   ConcreteFortranArray1d<int> pivots;

   // Prohibit assignment
```

```
   void operator=(
     const FactoredInPlace<SharedStorageT, float>&);
};
```

Here `ConcreteFortranArray2d<EltT>` is a column-major (i.e., FORTRAN storage layout) 2d array having elements of type `T` and with no virtual functions. The constructor takes a pointer to the matrix storage (presumed to have been released from control of the unfactored matrix) and calls the appropriate LAPACK subroutine. We'll ignore details like pivots since we have covered them in previous columns.

(One day, compilers will support support the partial template specializations described in section 14.6 of the draft ANSI/ISO C++ standard When that day comes, we'll be able to write a class that specify the storage type, but leaves the element type as a template argument, like this:

```
template<class EltT>  // New in ANSI.
class FactoredInPlace<
        ConcreteFortranArray2d<EltT>, EltT>
   { ... }
```

Neat!)

Now we have to back up and fill in the functionality we lost when we removed the `factor()` member in moving from `LapackRect` to `ArrayInSharedStorage`. We'll use a global "interconnection" function to link the abstractions of sharable storage and in-place factoring:

```
template<class SharedStorageT, class EltT>
FactoredInPlace<SharedStorageT, EltT>
factor(ArrayInSharedStorage<
                        SharedStorageT, EltT>& a) {
   // Connects ArrayInSharedStorage and
   // FactoredInPlace to allow factor(a).solve(b);
   return a.release();
}
```

This technique of connecting classes by global functions has much to recommend it. It decouples `ArrayInSharedStorage` from matrix factoring and, conversely, let's us use `FactoredInPlace` with other sources of array storage. A `factor()` member function would couple the two classes.

We should pause for a moment to talk about efficiency. Suppose we write the following code:

```
typedef ConcreteFortranArray2d<float> Array2d;
ArrayInSharedStorage<Array2d, float>  a(100,100);
// ... fill a ...
FactoredInPlace<Array2d, float> factored_a = factor(a);
```

The last statement potentially invokes `FactoredInPlace`'s copy constructor twice, once during the return from `factor()` and once during the initialization of `factored_a`. These "unnecessary" copies are unacceptable if the matrix is large. Fortunately, many compilers eliminate such copy constructor invocations. If your compiler doesn't, or if you want to be portable, use a reference-counted smart pointer for `ap` instead of a built-in pointer (see for example [3], Sec. 14.5 and references therein). That way, copy constructor invocations don't copy the data and memory management is still handled correctly.

Finally, we can write a driver routine for solving $Ax = b$:

```
template<class ShareableArray2d, class EltT>
ConcreteFortranArray2d<EltT>&
xForAxEqb(ShareableArray2d& a,
  ConcreteFortranArray2d<EltT>& b) {
   return factor(a).solve(b);
}
```

This code using matrices is easy to write, applies to many different storage formats, and adds considerable saftey to the LAPACK core with essentially no performance degradation.

The Virtual Face of Polymorphism. Now let's rewind to the `LapackRect<EltT>` class:

```
template<class EltT>
class LapackRect {
public:
  // Array behavior not shown...
  FactoredLapackRect<EltT> factor();
private:
  ConcreteFortranArray2d<EltT>* ap;
};
```

and take a different view. Instead of replacing the concrete array pointer with a template argument, let's abstract the checks for validity of storage up into a base class that clients of the matrices will work with. In practice, we'll use a layer of three base classes. The first is independent of element type to reduce template expansion:

```
class ValidityCheckedArrayBase {
public:
  ValidityCheckedArrayBase() : _valid(true) {}
  virtual ~ValidityCheckedArrayBase() {}

  int nrows() const {
    if (!isValid()) error();
    return doNrows();
  }

  int ncols() const {
    if (!isValid()) error();
    return doNcols();
  }
protected:
  bool isValid() const { return _valid; }
  void setInvalid()    { _valid = false; }
  void error() const;   // Throws exception
private:
  // Prohibit assignment through interface
  void operator=(const ValidityCheckedArrayBase&);
private:
  virtual int doNrows() const = 0;
  virtual int doNcols() const = 0;

  bool _valid;
};
```

Notice that we have a private interface being called by the non-virtual functions of the base class. For example, clients can only call `doNrows()` by calling `nrows()`. This guarantees that derived classes cannot forget to apply validity checks: users can only get to the derived class implementations by calling the base class checking functions. For those of you who are into "design patterns," this coding style is an example of Gamma et al.'s *template method*[1] design pattern [4]. Barton and Nackman [3], Section 10.3, and Meyers [5], Item 37, also discuss the implications of non-virtual functions in public base classes.

The second layer of the base class is element type specific; it is parallel to `ArrayInSharedStorage` in that it provides a checked array:

```
template<class EltT>
class ValidityCheckedArray
   : public ValidityCheckedArrayBase {
public:
  // Array behavior
  EltT& operator()(int i, int j) {
    if (!isValid()) error();
    return doSubscript(i, j);
  }
  // ...
private:
  virtual EltT& doSubscript(int, int) = 0;
};
```

[1] This use of "template" does not refer to the C++ meaning of the word.

The third and final layer adds factoring:

```
template<class EltT>
class ValidityCheckedFactorableArray
   : public ValidityCheckedArray<EltT> {
public:
  // Matrix behavior
  Solvable<EltT>* factor() {
    if (!isValid()) error();
    setInvalid(); // Factoring invalidates unfactored form
    return doFactor();
  }
private:
  virtual Solvable<EltT>* doFactor() = 0;
};
```

The return type for `factor()` is a pointer to an object derived from this interface base class:

```
template<class EltT>
class Solvable {
public:
  virtual
    ConcreteFortranArray2d<EltT>&
        solve(ConcreteFortranArray2d<EltT>&) const = 0;
  virtual ~Solvable() {}
private:
  // Prohibit assignment
  void operator=(const Solvable<EltT>&);
};
```

Thus the interface to a `Factorable` array gives us an interface to an object with a `solve()` member function.
The bottom layer of the validity checking base classes becomes the base for a new and improved `LapackRect` class:

```
template<class EltT>
class LapackRect
   : public ValidityCheckedFactorableArray<EltT> {
public:
  LapackRect(int nrows, int ncols);
  LapackRect(const LapackRect<EltT>& a);
  LapackRect<EltT>& operator=(const LapackRect<EltT>&);

  virtual ~LapackRect() { delete ap; }
private:
  // Implementations of interface-specified functions
  virtual
    EltT& doSubscript(int i,int j) { return (*ap)(i, j); }
  virtual
    int   doNrows() const          { return ap->nrows(); }
  virtual
    int   doNcols() const          { return ap->ncols(); }
  virtual FactoredLapackRect<EltT>* doFactor();

private:
  ConcreteFortranArray2d<EltT>* ap;
};
```

This class does no checking, it simply forwards each array 2d function to the member `ap` thus implementing the private interface of the base classes.
Of course, the `doFactor()` member does not forward to `ap` since factoring is the capability that we are adding to the 2d arrays in the first place. The `doFactor()` member creates a factored matrix matching the storage format for `LapackRect<EltT>`:

```
template<class EltT>
FactoredLapackRect<EltT>* LapackRect<EltT>::doFactor() {
    ConcreteFortranArray2d<EltT>* p = ap;
    ap = 0;
    return new FactoredLapackRect<EltT>(p);
}
```

The FactoredLapackRect<EltT> constructor, given a pointer to the underlying matrix storage, overwrites the storage with factored-matrix components:

```
template<class EltT>
class FactoredLapackRect
   : public Solvable<EltT> {
public:
   // Factor argument, overwriting it with the factors
  FactoredLapackRect(ConcreteFortranArray2d<EltT>* ap);

   // solve calls LAPACK and overwrites b with x.
   virtual ConcreteFortranArray2d<EltT>&
      solve(ConcreteFortranArray2d<EltT>& b) const;
   virtual ~FactoredLapackRect() { delete fmat; }

private:
  ConcreteFortranArray2d<EltT>*  fmat;
  ConcreteFortranArray1d<int>    pivots;

  // Prohibit copying.
  FactoredLapackRect(const FactoredLapackRect<EltT>&);
};
```

This class has the Solvable<EltT> interface and implements solve() by calling a LAPACK routine, passing it the factored matrix storage together with the array b of right-hand sides.

To implement factoring/solving for other matrix storage formats, we derived two classes, one like LapackRect<EltT> for the initial storage of A and one like FactoredLapackRect<EltT> for the storage of the factored form and its solution. These classes can now be used in code that is matrix-type independent:

```
template<class EltT>
ConcreteFortranArray2d<EltT>&
xForAxEqb(ValidityCheckedFactorableArray<EltT>& a,
              ConcreteFortranArray2d<EltT>& b) {
   Solvable<EltT>* factored_a = a.factor();
   factored_a->solve(b); // Solution is in b
   delete factored_a;
   return b;
}
```

This code calls through virtual function interfaces to factor and solve the system of equations.

Using virtual function interfaces has an important implication. In the template-based code, the global function factor() matches an unfactored matrix form to the corresponding factored form via the storage format template argument. In the virtual function-based code, ValidityCheckedFactorableArray<EltT>'s clients cannot tell which storage format is being used. This implies that the virtual function-based system of classes must connect factored and unfactored forms via a virtual function—doFactor()—defined by derived classes like LapackRect<EltT> and FactoredLapackRect<EltT>.

Interfaced Arrays. We might consider yet a different approach, if we started from a system of arrays that already have virtual function interfaces. For example, suppose we have 2D arrays with this interface:

```
template<class EltT>
class Array2dInterface {
public:
  virtual EltT& operator()(int i, int j) = 0;
  virtual int   nrows() const = 0;
  virtual int   ncols() const = 0;
  //..
};
```

A more "natural" object-oriented approach to this whole problem would use base-class composition [3, Section 10.7] to extend Array2dInterface<EltT> with factor():

```
template<class EltT>
class FactorableArray2dInterface
   : public virtual Array2dInterface<EltT> {
public:
 // Caller controls lifetime of returned obj.
 virtual Solvable2dInterface<EltT>& factor() = 0;
};
```

Then the array implementations could be extended to make them "factorable," something like this:

```
template<class EltT>
class FactorableFortranArray2d
   : public virtual FactorableArray2dInterface<EltT>,
     private FortranArray2d<EltT> {
public:
  // ctors, assignments, and factor
};
```

However, this approach does not allow us to get both safety and in-place factorization because we only get an interface to the object returned by `factor()`, which precludes access to the underlying storage.

Summary. C++ programmers often have to choose between template and virtual function polymorphism. Sometimes the choice is obvious, as when implementing concrete container classes where speed is paramount; other times, the choice is equally obvious in the other direction, as when manipulating lists of objects with similar behavior. The difficulty comes in the middle, when the choice is not obvious. Our LAPACK wrapper classes fall in this middle ground.

At first thought, you might think numerical computing's speed requirements would preclude using virtual functions at all in matrix computations. It is certainly true that frequent virtual function calls for, say, subscripting would be disasterous. The key word here is "frequent": In most matrix computations, the vast majority of the work is done by "kernel" routines such as those provided by the BLAS (Basic Linear Algebra Subroutines); indeed most computer vendors devote considerable effort to providing fast BLAS implementations. For maximum speed, these routines must exploit subtle details of both the matrix representations and the particular machine. Indeed, the BLAS are often written in assembler.

But our situation is different: once matrices are initialized, almost all of the computation is done by calls to LAPACK (and thence to BLAS). Matrices are typically initialized where created and therefore the code is written directly against a particular type. In many cases this allows the compiler to replace virtual function calls with inline code. Regardless, the initialization is typically a tiny fraction of the total computation. Thus, for our application it is reasonable based on performance consideration to use virtual function interfaces.

Working with virtual function interfaces means dealing with memory management since working with interfaces means working with pointers (or references). For example, `doFactor()` must return a pointer to the interface of the factored form, requiring allocation on the heap; this in turn means that the client must manage the factored form's lifetime. A reference-counted pointer could handle this for us, but makes the interface more complex for the client. (Smart pointers used in `FactoredInPlace` are an internal detail not visible to the client.)

Code size is another consideration: client code written against virtual function interfaces need only be generated once; function templates result in separate generated code for each instantiation of the template. Whether this replication is significant depends on how much client code there is.

Type-safety simply falls out of the template version and the memory management headache accompaning any use of virtual function interfaces need not be faced. On the other hand, heavy template use with today's compilers often means long compile or link times, obscure error messages, or working around unimplemented language features. With compiler vendors now working to implement the ANSI C++ draft and with emerging tools for C++ we are moving toward a time when compiler pragmatics won't be as much of a consideration as they are today.

Corrigendum. We blew it. Steve Clamage was kind enough to send us email saying that he had read our last column [2] "with interest." Then he went on to tell us why our code is wrong. Steve is right.

We wrote

```
LapackRect<float> a(100, 100);
FactoredLapackRect<float> factored_a = a.factor();
```

Initializing `factored_a` implicitly invokes `LapackRect<EltT>`'s copy constructor, which is declared private. Now, the standard says that a compiler is free to eliminate such copy constructor invocations. But it also says that the compiler must check access rights even if it doesn't invoke the copy constructor. The two compilers we tried, including the one we used for our last column, didn't check access. Our code in this column does not use private copy constructors for this reason. Thanks, Steve.

References

[1] John J. Barton and Lee R. Nackman. Scientific and engineering C++: Wrapping LAPACK in objects. C++*Report*, 7(5):50–53, June 1995.

[2] John J. Barton and Lee R. Nackman. Scientific and engineering C++: Efficiency and indefinite types with runtime failures. C++ *Report*, 7(7):56–57, 60, September 1995.

[3] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley, Reading, MA, 1994.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[5] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, Reading, MA, 1992.