RC 20126 (07/10/95)
Computer Science

IBM Research Report

Scientific and Engineering C++: Efficiency and Indefinite Types with Runtime Failures

John J. Barton, Lee R. Nackman

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

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

Scientific and Engineering C++: Efficiency and Indefinite Types with Runtime Failures

John J. Barton Lee R. Nackman

July 12, 1995

Efficiency use versus complexity: this is the daily battle in building large systems. Carefully designed libraries are a potent weapon in this battle. C++ libraries can unify concepts, hide optimizations, and add compile time type-checking. Designing a library with all of these features is a challenge.

One approach good quality library design is to began with a high-quality subroutine library and wrap it in objects. An excellent example for scientific and engineering work is the FORTRAN subroutine library, LAPACK [1]. LAPACK efficiently solves linear equations like

$$A\mathbf{x} = \mathbf{b},$$

where A is the coefficient matrix, $\mathbf{b}$ is the right-hand side, and $\mathbf{x}$ is the sought-for solution. Most of the LAPACK system works by factoring the matrix A into products of simpiler matrices. The LAPACK library has unifying concepts and optimizations that library users need not understand to use, but its concepts are not expressed in the code and it supports very little compile-time checking.

In our previous column [2] we wrapped LAPACK with C++ objects. While LAPACK uses a few built-in FORTRAN data types and more than 250 different (but carefully named!) function calls, we created a system of C++ matrix types, each with a few functions. Our C++ system added conciseness of expression and type-safety to LAPACK. Both benefits resulted from exploiting the C++ type system: each mathematical concept in LAPACK was represented by a distinct C++ type.

Simple LAPACK wrappers Let's review our work from last time. Here is native LAPACK code transliterated into C-like C++:

```
float A[100][100];  // Matrix
float B[100][1];    // Right hand side
int   IPIV[100];    // Pivots
int   INFO;         // Return code

// Code to fill A and B goes here ...

// Solve AX = B
sgetrf(100, 100, &A[0][0], 100, IPIV, INFO);
sgetrs(100, 100, &A[0][0], 100, IPIV,
                     &B[0][0], 100, INFO);
```

Immediately before `sgetrf()` is called, `A` is conceptually a mathematical object, an invertible matrix. Concretely, `A` is an array of 10,000 numbers. Immediately after the `sgetrf()` call, `A`, together with `IPIV`, is a factored matrix, consisting conceptually of the product of an upper triangular matrix, a permutation matrix, and a lower triangular matrix. Concretely, `A` is still 10,000 numbers.

Now look at the equivalent code using the C++ wrapper objects we developed last time:

```
LapackRect<float> a(100,100);
// ... fill a ...
FactoredLapackRect<float> factored_a = a.factor();
LapackRect<float> b(100, 1);
// ... fill b with right hand side  ...
factored_a.solve(b); // Solution is in b
```

We used the C++ type system to express directly the relevant concepts: LapackRect<float> is the unfactored form and FactoredLapackRect<float> is the factored form. The code is quite clear on this. Moreover, if we try to factor the factored form, the compiler would tell us that there is no factor() member function for FactoredLapackRect<float>. When the compiler diagnoses a type error like this it is catching a logic error for us. This is precisely the safety that we hoped to gain by exploiting types.

So far so good. But notice that the FORTRAN version uses only one A matrix. The C++ version uses two different objects, a and factored_a. The concrete representations of those two objects are roughly like this:

```
template<class T>
class LapackRect :
    public ConcreteFortranArray2d<T> { // LAPACK A, M, & N
public:
    LapackRect(int nrows, int ncols);
    FactoredLapackRect<T> factor();
    // ...
};

template<class T>
class FactoredLapackRect {
public:
    LapackRect<T>& solve(LapackRect<T>&);
private:
    friend FactoredLapackRect<T> LapackRect<T>::factor();
    FactoredLapackRect(int nrows, int ncols);

    ConcreteFortranArray2d<T>   fmat;     // LAPACK "A"
    ConcreteFortranArray1d<int> pivots;   // LAPACK "IPIV"
};
```

A ConcreteFortranArray2d<T> is a FORTRAN-compatible 2d array of T elements. Both a and factored_a contain such an array. For LapackRect<T>, the 2d storage is in the base class; for FactoredLapackRect<T>, the 2d storage is in the member variable fmat. Together our nice C++ version uses at least twice as much space for matrix storage as the FORTRAN version. In our effort to use C++ types to gain safety, we've thrown out an important optimization that LAPACK provides: in-place factorization.

LAPACK optimizes space by factoring matrices in place, replacing the contents of the input matrix with the factored form. This also avoids copying the contents of the input matrix in the common case that its values are no longer needed. Our first challenge is to add this space efficiency of LAPACK while we retain the type-safety of the wrapper; our second challenge is to ponder why this optimization is trivial in FORTRAN but requires extra work in C++.

Indefinite Types Our initial wrapper design didn't accurately capture the LAPACK model. In the LAPACK model, the array of numbers, A, is interpreted in two different ways. This array of numbers is neither a factored nor an unfactored matrix, it's just an array of numbers. Then, depending on the state of the computation, this one object is interpreted as either an unfactored matrix or (part of) a factored matrix. In the FORTRAN LAPACK code, this interpretation is done solely by the programmer. We want this dual interpretation to capture the space savings, but we want it to be explicit in our C++ code to retain the expressiveness and type-safety.

To achieve this we'll add one level of indirection – a pointer – to our two wrappers. Instead of containing our 2d array directly as a base class subobject or member variable, we refer to the 2d array indirectly via a pointer to a ConcreteFortranArray2d<T> object. Then we can share the same storage between two different objects, the unfactored matrix object and the factored one.

Each wrapper object can be designed to allow only the proper operations as we did in the simple wrapper case. LapackRect<T> and FactoredLapackRect<T> still serve as interfaces to the underlying array of numbers. But in addition they have to cooperate to share this array.

Here is how the new version could look. The unfactored matrix is represented by this class:

```
template<class T>
class LapackRect {
public:
  LapackRect(int nrows, int ncols);
 ~LapackRect();
```

Equivalent to book fig 15.2, but in a standard notation e.g. OMT or Booch.

Figure 1: The FORTRAN array of numbers is represented by an instance of `ConcreteFortranArray2d<T>`. Before factoring, this instance is a component of `LapackRect<T>`. Member functions that implement array behavior are implemented by forwarding to the `ConcreteFortranArray2d<T>` instance.

```
  // Array behavior
  T& operator()(int, int);           // Subscripting
  LapackRect(const LapackRect<T>&);  // Array copy
                                     // Array assign
  LapackRect<T>& operator=(const LapackRect<T>&);
  int nrows() const;                 // # rows
  int ncols() const;                 // # columns
  // ...

  FactoredLapackRect<T> factor();
private:
  ConcreteFortranArray2d<T>* ap;   // LAPACK "A"

  void check() const {
    if (!ap) throw "FactoredLapackRect not in matrix state";
  }
};
```

The ap member points to a `ConcreteFortranArray2d<T>` that holds the numbers. This pointer also acts as a flag: a zero-valued pointer means that the storage is under control of another object. (Using pointers as flags is such a natural C and C++ idiom that we sometimes forget that FORTRAN programmers have to learn). The private member function `check()` tests this pointer-flag and signals an error. By using this function rather than direct tests of the member, we can change our error handling procedure in one place.

The underlying 2d array is allocated by the unfactored matrix constructor:

```
template<class T>
LapackRect<T>::LapackRect(int nrows, int ncols) :
  ap(new ConcreteFortranArray2d<T>(nrows, ncols)) {}
```

The resulting structure is shown in Figure 1

When `factor()` is called on the unfactored matrix, control of the `ConcreteFortranArray2d<T>` is handed off to the newly created `FactoredLapackRect<T>`:

```
template<class T>
FactoredLapackRect<T> LapackRect<T>::factor() {
  check();
  ConcreteFortranArray2d<T>* p = ap;
  ap = 0;
  return FactoredLapackRect<T>(p);
}
```

The ap pointer is zeroed to indicate that the unfactored form is no longer in a valid state (we'll see below that the `FactoredLapackRect<T>` constructor factors the matrix). Each member function that provides array-like behavior calls `check()` before using ap. For example, here is the subscripting member function:

```
template<class T>
T& LapackRect<T>::operator()(int i, int j) {
  check();
  return (*ap)(i, j);
}
```

With the version of `check()` we gave above, an exception string will be thrown if we access the unfactored matrix after the factoring.

The factored form is represented by this class:

Equivalent to book fig 15.3 but in standard notation.

Figure 2: The underlying array becomes a controlled component of a FactoredLapackRect<T> and LapackRect<T> is in invalid state.

```
template<class T>
class FactoredLapackRect {
public:
   LapackRect<T>& solve(LapackRect<T>& b);
  ~FactoredLapackRect();
private:
  friend FactoredLapackRect<T> LapackRect<T>::factor();
  FactoredLapackRect(ConcreteFortranArray2d<T>*);

  ConcreteFortranArray2d<T>*  fmat;
  ConcreteFortranArray1d<int> pivots;   // LAPACK "IPIV"

  // Prohibit these.
  void operator=(const FactoredLapackRect<T>&);
  FactoredLapackRect(const FactoredLapackRect<T>&);
};
```

The factored form has no array behavior and its constructor is private; a factored form can only be created by factoring an unfactored form.

When control of the underlying array is handed to the FactoredLapackRect<T> constructor, the matrix is factored and it then becomes possible to solve the system of equations:

```
template<class T>
FactoredLapackRect<T>::
FactoredLapackRect(ConcreteFortranArray2d<T>* ap) :
  fmat(ap), pivots(min(ap->nrows(), ap->ncols())) {
  // Call LAPACK factoring routine ...
}
```

The constructor initializes fmat to take control of the underlying array of numbers, initializes pivots to the appropriate size, and then calls the appropriate LAPACK factoring subroutine. The structure after factoring is illustrated in Figure 2.

This implementation allows in-place factorization, an important LAPACK optimization, while retaining full type checking. As in the LAPACK FORTRAN code, we cannot use the unfactored array after we factor it. However, unlike the FORTRAN version, uses of the unfactored array after factoring do not give bizarre results. Instead, such access will immediately and predictably stop program processing. Thus we have implemented in-place factorization with a run-time type-check.

Type-checking So why did we need to do more work to add in-place factorization to our C++ classes while the FORTRAN-like C++ code gets to use in-place factorization for "free"? There are two kinds of answers.

First, consider how we obtained type-safety in our wrapper classes. We did it by restricting the kinds of operations that we allow on the wrapped storage: we allow factor() on unfactored matrices but not on factored ones; we allow solve() on factored matrices but not on unfactored ones. These restrictions help us avoid the error of using an unfactored matrix as a factored one and vice-versa, but that dual-use is exactly what happens in the in-place factorization. Thus to retain the type-checking, we have to add runtime transfer of storage from one object to another.

Second, consider what it means to have an object-based design. It means that certain bits — the object's state — will be interpreted by certain rules — the object's member functions. In place factorization means that certain bits — the array of equation coefficients — will be interpreted by two different rules — the rules of unfactored and factored matrices. The only way to reconcile an object-based design with in-place factorization is to pass the bits between two objects and simulataneously change the way both objects behave. Thus to work in an object-based design, we have to add the runtime transfer of bits from one object to another.

C++ allows us to work with as much or as little type-checking as we choose to implement. We can work with built-in floating point 2d arrays or we can build a programmer defined datatype that supports our specific application requirements and adds significant checks to our program logic. Programmer defined datatypes are particularly valuable in larger and more complex systems and it pays to understand how they can be optimized.

Summary We have improved the space efficiency of our C++ classes for large matrix problems. Notice that these changes were fully absorbed in our library code: the code that uses our wrapper classes has not changed. This success of encapsulation is a minor victory during library design, but as a project matures it becomes a dominant force. Without encapsulation we cannot afford to improve efficiency in a mature system if it means changing the data-structures used to implement the system's algorithms.

LAPACK encapsulates the algorithmic complexity but as it stands we must decipher its encoded function names, long argument lists, and we must ensure that we store and access values in the 2d arrays properly. The class library wrapper adds uniform function names, compact intuitive argument lists and typed interfaces to protect against errors. The additional complexity of the library reflects the complexity of the information being conveyed to the compiler. Once we have created this library, factoring matrices becomes simpiler, less prone to error, and we still can achieve good space and runtime performance.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

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