C++ Classes for **R** Objects

Douglas M. Bates [*]   **Saikat DebRoy**[†]

Abstract

When creating the R Matrix package, which provides access to the Fortran Lapack and BLAS3 routines, we patterned the functions after the corresponding S-PLUS library but chose a completely different implementation. We based the R implementation on the `lapack++` classes that provide `C++` wrappers for the Lapack code.

There are several advantages to using `C++` instead of `C` for code that will be called from R. In particular it is possible to encapsulate some of the S-language class structure in the `C++` classes. There are also disadvantages to using `C++`. We discuss these for the particular example of the Matrix package and also describe a more general approach of reflecting basic R object types in `C++` classes.

1   Using `C++` in the Matrix package for R

Numerical linear algebra is a cornerstone of any numerical or statistical computing system, such as S. The different implementations of S — S-PLUS and R — use extended versions of the Fortran code in the Linpack[2] and Eispack[4, 3] packages for numerical linear algebra. S-PLUS versions 3.4 and later have a separate Matrix library based on Lapack[1], which is an extended and enhanced Fortran package combining capabilities of both Linpack and Eispack. Linpack and Lapack both use the Basic Linear Algebra Subroutines (BLAS) for the compute-intensive calculations but Linpack only uses the level-1 BLAS for vector-vector operations while Lapack uses levels 1, 2, and 3 for vector-vector, vector-matrix, and matrix-matrix operations.

Recent work on automatically tuned linear algebra subroutines (Atlas[5]) has made the use of Lapack even more desirable. Not only is Lapack more robust and more complete than Linpack but, when combined with Atlas, it can be considerably faster.

We decided to create a Matrix package for R patterned after the corresponding S-PLUS library in terms of the S-language function calls. The actual interface from R to the Lapack Fortran code is quite different from the S-PLUS library. The R Matrix package uses the `.Call` interface to create `lapack++` objects which, in turn, access the Fortran code.

By using the `lapack++` code as the intermediate step we can provide an object-oriented numerical linear algebra system in the compiled code. There are also certain technical advantages in calling the Lapack code from `C++` interface routines rather than from C.

1.1 Reasons for using `C++` intermediate routines

The structure of Lapack, described in [1] (available as `http://www.netlib.org/lapack/lug/lapack_lug.html`), is like an object-oriented system, even though it is written in Fortran. It provides several classes of matrices and several actions that can be performed on matrices. The names of Lapack subroutines are generally constructed from a storage type, a matrix class, and an operation. For example, `dgetrf` is a subroutine that creates a triangular factorization of a double precision, general matrix.

By using `C++` instead of `C` for the wrapper functions to access Lapack from R, we can preserve some of the object-orientation in the R library as we call the Fortran routines.

The `C++` language provides classes and methods, inheritance of classes, polymorphism in calls, and automatic destructors that get called when an identifier goes out of scope. All of these properties can be very helpful in writing code that interfaces with R.

1.1.1 Classes and abstract classes

The original `lapack++` code provided classes for several types of matrices and a few methods for each of these classes. We extended the original `lapack++` classes with abstract classes `LaMatrix`, `LaMatDouble`, and `LaMatComplex` to provide a hierarchy of classes and a wider selection of methods. All the matrix and vector classes in our extended `lapack++` inherit from `LaMatrix`. All the double precision matrix and vector classes inherit from `LaMatDouble` and all the complex matrix and vector classes inherit from `LaMatComplex`.

With `C++` we can create a pointer to an object from the abstract class. A call to a virtual method accessed through that pointer will call the correct method for the actual class. For example, the entire `C++` source for the function that is called by the R function `norm.Matrix` is

```
SEXP R_LapackPP_norm(SEXP a, SEXP which)
```

```
{
    LaMatDouble *aa = 0;
    try {
        if (!isString(which))
        error("R_LapackPP_norm : which should be of mode character");
        aa = asLaMatrix(a);
        SEXP val =
        PROTECT(ScalarReal(aa->norm(CHAR(STRING_ELT(which, 0))[0])));
        setAttrib(val, R_ClassSymbol, ScalarString(mkChar("norm")));
        delete aa;
        UNPROTECT(1);
        return val;
    } catch(LaException xcp) {
        delete aa;
        error(xcp.what());
        return R_NilValue;        // to keep -Wall happy
    }
}
```

The `try/catch` block is standard for such interface routines. It allows us to throw exceptions when error conditions are detected within the `lapack++` functions and methods. The alternative, returning error codes and checking or propagating them at each call, quickly becomes tedious. The main part of the code is, naturally, in the `try` block. The `catch` block just does some cleanup then reports the error using R's `error` function.

In the `try` block, after some error checking on the `which` argument, the `C++` code passes the R object `a` (in `C/C++` code R objects have the type `SEXP`) to the function `asLaMatrix` that returns a pointer to an object in one of the actual classes inheriting from `LaMatrix`. The actual class is determined by examining the S-language class attribute of `a`. The `norm` method for that class is invoked (`aa->norm`) and the value is converted to an `SEXP`, which is returned as the value of `R_LapackPP_norm` with the S-language class `"norm"`. The appropriate `norm` method for any `LaMatrix` class is determined automatically.

This example shows some of the power of the `C++` classes. The virtual method `norm` behaves like a generic function in S.

1.1.2 Use of more sophisticated declarations

The header files for `C++` classes often contain a substantial portion of the implementation code for the classes because the use of the `inline` directive for trivial methods is encouraged. Many of the methods in the `lapack++` classes are inlined. An inlined method or function is similar to a preprocessor macro in `C` except that the inlined functions and methods and calls to them are subject to syntax checking by the compiler.

Arguments to `C++` methods and functions can include the `const` keyword. This usage is encouraged because the compiler will declare an error if the argument is

used in such a way that it could change the value of an object passed by a pointer or a reference.

The header files are also the place where polymorphic methods or functions are declared. It is the compiler's responsibility to check the types of the arguments and ensure that the correct function or method body is used.

1.1.3 Use of references

In `C` arguments are passed to functions by value. If an argument is to be modified it must be passed as a pointer. Because Fortran uses call by reference all arguments, even compile-time constants, must be passed as pointers when calling a Fortran subroutine from `C`. It is not difficult to do this but it can be irritating to be required to create a local variable and assign it a constant value solely so that a pointer to that variable can be passed as an argument. There is also the danger of accidently changing the value of such a "constant" within the Fortran subroutine.

In `C++` one can avoid these extra steps because the prototype of a function can declare that some arguments are to be passed as references. If such an argument declaration also includes the `const` keyword then a compile-time constant, or the result of a function evaluation, can be used as the actual argument.

We have created a set of `C++` declarations for all the subroutines and functions in the double precision version of Lapack. These declarations, contained in the file `lapackd.h`, ensure that

- All scalar arguments are passed as references.

- All vectors or matrices (in the Fortran sense — these are both stored as single-dimensional arrays in `C++`) are passed as pointers.

- Any arguments that will be unchanged upon return from the Fortran subprogram include the `const` keyword.

For example, the arguments `M`, `N`, and `LDA` are not altered by the subroutine `DGETRF`, which is declared as

```
SUBROUTINE DGETRF( M, N, A, LDA, IPIV, INFO )
         INTEGER        INFO, IPIV( * ), LDA, M, N
         DOUBLE         PRECISION A( LDA, * )
```

The corresponding declaration in `lapackd.h` is

```
void F77_NAME(dgetrf)(const int& m, const int& n, double* a,
                      const int& lda, int* ipiv, int& info);
```

In the constructor for an `LaLUFactorDouble` object that takes a `LaGenMatDouble` object as an argument, this subroutine is called as

```
F77_CALL(dgetrf)(A.size(0),A.size(1),&A(0,0),A.gdim(0),&piv(0),info);
```

The only arguments being passed as pointers are those passed to `a` and to `ipiv`. The arguments passed to `m`, `n`, and `lda` are the results of method calls on the object `A`. This call looks more "natural" than a call from a `C` function, where all the arguments must be passed as pointers.

1.2 Reasons not to use `C++` intermediate classes

In addition to showing advantages of `C++` for intermediate code, the `R_LapackPP_norm` example also shows some of the disadvantages of `C++`. Many of these disadvantages are the result of `C++` being designed as an extension of `C`. In particular, it was not designed to have a run-time environment, such as that the environment for Java, that provides garbage collection. In `R_LapackPP_norm` it is necessary to create a pointer (or a reference) to an object of the abstract class `LaMatrix` before a virtual function can be invoked. Because storage is allocated when the object is created by `asLaMatrix`, the `delete` operator must be invoked on it before the function returns. If this is not done the destructor will never be called and there will be a memory leak.

In some ways the single most difficult aspect of designing numerical linear algebra code in `C++` is deciding when destructors should be invoked to recover the memory allocated on temporary objects. Those who are familiar with Java, for example, find this tedious.

There is a certain amount of overhead in using the object-orientation of `C++` from R. The package author must construct a function like `asLaMatrix` that transforms an R object, passed as an `SEXP`, into a `C++` object of the desired class. Although not shown here, we also created `asSEXP` methods for the various `lapack++` classes to do the reverse transformation. These methods are used are used to create the function values to be returned through `.Call`. (A function called from R through `.Call` must return an `SEXP`.)

From an organizational point of view, the use of `C++` requires yet another compiler and yet another set of libraries against which the executable must be linked and for which shared objects must be available at run time. For a system like R that is used on many different platforms, creating autoconf scripts or similar system-building tools that can work with three different languages (`C`, Fortran and `C++`) across all these platforms is a formidable undertaking.

Another aspect of `C++` that creates organizational difficulties is the heavy dependence on header files. The definition of classes, methods, and functions must be available at compile time to other packages whose code uses these classes. The current installation process for an R package does not preserve the header files so cross-dependencies between packages at this level are problematic.

2 General `C++` classes for use with R

Although the organizational issues described in the previous section are not trivial, they are the sorts of problems that can be solved by R-core members and packaged as part of the R source distribution. The issues of transformation from R objects to `C++` objects and back and of memory management need to be addressed by each package developer. To alleviate some of these problems we created a set of `C++` classes that mirror basic R objects while using the R memory allocator. With these classes it is possible to write `C++` code that maps `SEXP` objects to `C++` objects and that uses the R garbage collection mechanism.

2.1 Objected-oriented representation of **R** Objects

There are two fundamental types of internal R objects — vectors (LOGICAL, IN-TEGER, REAL, COMPLEX, VECTOR, STRING) and lists (LIST, LANG). For some of the vector objects (INTEGER, REAL, COMPLEX) arithmetic operations are meaningful and these are considered separately.

We created `C++` classes for these objects that are very similar to the `C++` STL container classes. A simple function using these classes is

```
VectorSExpn f(RealSExpn x)
{
    VectorSExpn val(3);
    if (x.size() > 0) {
        val[0] = x;
        val[1] = x*2 > 1;
    }
    val[2] = StringSExpn(2);
    val[2][0] = "a";
    val[2][1] = "b";
    return val;
}
```

To call this function through `.Call` from R, we must use a `C` wrapper function to create the `RealSExpn` from a `SEXP` and to get a `SEXP` from the return value —

```
SEXP fwrapper(SEXP arg)
{
    return f(RealSExpn::create(arg)).getSExp();
}
```

The argument `x` to `f` is an R `numeric` object of unknown size and the return value is an R `list` of three elements. In `C++`, the R object classes are named according to their internal `C` API names — VECTOR and STRING become `VectorSExpn` and `StringSExpn` respectively.

Note that the operators like `*` and `>` are overloaded for the R classes and are vectorized. These, strictly speaking, are not part of the R API but operations like these are sufficiently common that it is useful to do operator overloading. The `C++` classes for R objects are similar to some built-in and STL classes. For example, a `StringVector` object looks very much like a `vector<string>` object. A `RealSExpn` on the other hand, has properties of both a `vector<double>` object and a `valarray<double>` objects.

2.2 Design issues

One of the fundamental design issues for these `C++` classes is deciding how closely they should resemble the classes at the S-language level. While this is highly de-batable, we believe that `C++` has its own conventions and they should be followed

where possible. Users familiar with the `C++` STL should find the R interface easy to understand.

At the same time, one should be able to access the R API as much as possible. We should look to the R API for guidance in those cases where it has no equivalent in `C++` — although some of the functions and macros in the R API can become methods for appropriate classes.

2.3 Current state of the implementation

The way the `C++` classes are implemented now, computation of `val[1]` in the above example is equivalent to

```
RealSExpn temp = x*2;
val[1] = temp > 1;
```

The computation of the temporary can be avoided and the whole computation can be done in a single loop in a more sophisticated implementation.

The other thing to be noted is that the temporaries created in this process are still collected by the R garbage collector. The declared objects (like `x` or `val`) are preserved across garbage collection by protecting them internally in the implementation. The `SEXP` corresponding to a variable is unprotected once it goes out of scope.

3 Conclusions

Although it is not the ideal programming language by any means, `C++` does have the advantage of meshing with the existing `C` and `Fortran` code in R while providing object-orientation. We found this helpful when creating the Matrix package because we could use the `lapack++` classes.

We carried this approach further by creating `C++` classes for the internal representations of fundamental R objects. We started working on the implementation to make writing `C++` code easier for ourselves and have implemented those parts of the R API that we use often. Other parts, such as the R LANG objects are unimplemented — because we are not sure what is the best way to approach them. For example, it is entirely possible to implement R functions with `C++` function objects. Whether that is desirable and useful is not clear.

References

[1] E. Anderson, Z. Bai, C. Bischoff, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, 2nd ed.* SIAM, 1994. 1, 2

[2] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *Linpack Users' Guide.* SIAM, Philadelphia, 1979. 1

[3] B.S. Garbow, J.M Boyle, J.J. Dongarra, and C.B. Moler. *Matrix Eigensystem Routines - EISPACK Guide Extension.* Springer-Verlag, New York, 1977. 1

[4] B.T. Smith, J.M. Boyle, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler. *Matrix Eigensystem Routines - EISPACK Guide.* Springer-Verlag, New York, 1974. 1

[5] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the atlas project. Available as `http://www.netlib.org/lapack/lawns/lawn147.ps`, 19 September 2000. 2