Scientific and Engineering C++: What's That Template Argument About?

John J. Barton Lee R. Nackman

February 28, 1994

Technical programmers often work with models of reality populated with many identical objects. Two tools important for their work are arrays and templates. Arrays store the many objects and templates allow efficient, adaptable code to be written to manipulate the objects. In this issue we use templates to write array classes. The story is about arrays; the message is about templates.

In our preceding column (Dec. '93), we argued that there is no kitchen-sink array class that can meet the spectrum of performance, compatibility, and flexibility requirements that arise in scientific and engineering computing. As we support more function in a general purpose array class, we lose performance.

Instead of loading all kinds of function into one array class, a coordinated system of array classes is needed. Even better would be a customizable system of array classes that was already coordinated. Then we could crank out the kinds we want today: Fortran-compatible and C++-compatible arrays, arrays with size fixed at creation time and arrays with variable size, arrays with and without virtual function interfaces, and so on. And later we could come back and get new custom arrays suited to new special purposes.

This column outlines a plan for building a coordinated family of multidimensional array class templates. Along the way, we study the plan itself.

Collection Parameterization. The archetypal use of templates is for *collection parameterization*, to allow reuse of one piece of source code to create multiple collections. For example, collection parameterization can be used to create new array classes like this:

```
template<class T>
class Array2d {
public:
  typedef T EltT;
  Array2d(int, int);
  int shape(int) const;
  EltT operator()(int, int) const;
  // ...
};
```

Then we can create an `Array2d<double>` and, if we have, say, a class called `Spinor`, we can create an `Array2d<Spinor>`.

Function templates can use collection parameterization to work on the elements of every collection generated from a template. Here is an example from our preceding column:

```
template<class T>
T runningSum(const Array2d<T>& a) {
  T sum(0);
  int m = a.shape(0);
  int n = a.shape(1);
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      sum += a(i,j);
    }
  }
  return sum;
}
```

A function generated from this function template sums the elements of the array passed to it. Collection parameterization supports reuse: we can write one piece of source code, `Array2d<T>` or `runningSum()`, and use it for `double` objects or `Spinor` objects and so on.

Service Parameterization. Notice that the template `Array2d` will work for most template argument values: we can expect to make an array of `Spinor` objects even though we don't have a clue what they might represent. The preceding `runningSum()` function template makes more demands on the type parameter: the element type must support `+=` for example. Another version of `runningSum()`, also from our last column, makes even more demands on the template parameter:

```
template<class AnArray2d>
AnArray2d::EltT runningSum(const AnArray2d& a) {
  AnArray2d::EltT sum(0);
  int m = a.shape(0);
  int n = a.shape(1);
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      sum += a(i, j);
    }
  }
  return sum;
}
```

When this `runningSum()` is called, as in the code

```
Array2d<int> a(2, 3);
// ...
int sum = runningSum(a);
```

the compiler generates a function (assuming no non-template `runningSum()` function that exactly matches the argument type exists) from the template, substituting the argument's type for the type parameter `AnArray2d`.

Unlike the `T` template parameter in `Array2d<T>`, only certain array types will work as matches for `AnArray2d`. The array type must supply a nested type `EltT`, a member function named `shape` taking an `int` argument, and a function-call operator taking two `int` arguments. These are all used directly in the code for `runningSum()`. Also, the nested type must have a constructor that can accept the integer `0` as its argument, a copy constructor, and a `+=` operator. Without these, the function generated from the template won't compile.

These requirements boil down to certain names being supplied like `EltT` or `shape`. Some of the names must have particular properties. For example, `shape` must be a function that can be called with an `int` argument. Let's call all these requirements "services": function templates like the second version of `runningSum()` will only work if their template parameter provides these services. To emphasize the tighter coupling of such templates to their template argument, we'll call this *service parameterization*. Since `runningSum()` uses services supplied by the type substituted for `AnArray2d`, we can say that it is a *client* of the `AnArray2d` parameter.

Our focus on the names needed and supplied dominates when we begin to use templates to generate the requirements of service parameters. So `runningSum()` needs `shape()` for every `AnArray`? No problem: we'll supply a `shape()` in a system of array classes built with templates.

Class Templates as Clients. As we have discussed, many kinds of arrays are necessary for scientific and engineering programming. A large family of arrays can be built from a pointer to the first element plus a function of subscripts that gives the offset to other elements. For example, arrays stored with column-major layout, like FORTRAN arrays, arrays stored with row-major layout like C++ arrays, packed symmetric triangular arrays, and reduced-dimension projections of these arrays can all be implemented this way. See Figure 1.

Our system combines collection parameterization for the element type with service parameterization for the computation of the offset, and the whole system is designed to supply the services needed by array-using clients like `runningSum()`. The central class template, `ConcreteArray2d`, provides the pointer to the first element, but it gets its offset computation from the template argument called `Subscriptor`:

```
template<class Subscriptor, class T>
class ConcreteArray2d :
   private Subscriptor {
public:
   typedef T EltT;

   // Expose Subscriptor's 'shape' members
   Subscriptor::dim;
   Subscriptor::shape;
   Subscriptor::numElts;

   // Element access via subscripts
   const T& operator()(int s0, int s1) const;
         T& operator()(int s0, int s1);
protected:
   ConcreteArray2d(const Subscriptor& s, T* p) :
     Subscriptor(s), datap(p) {}
  ~ConcreteArray2d() { delete [] datap; }

   T* datap;
private:
   // Prohibit copy and assignment
   typedef ConcreteArray2d<Subscriptor, T> rhstype;
   ConcreteArray2d(const rhstype&);
   void operator=(const rhstype&);
};
```

The familiar parts are the parameterized pointer to the elements, `datap`, and the subscripting functions returning a reference to an element. We protect the constructor to require that specific kinds of arrays be defined by derivation from this template. The derived classes are responsible for allocating memory to hold the elements and for providing appropriate copy constructors and assignment operators. We shall come to an example shortly. First let's look at the unfamiliar part, the `Subscriptor` base class.

The `Subscriptor` template argument is a class that supplies members that know the shape of the array and one member, `offset()`, that provides an appropriate offset computation. `ConcreteArray2d<Subscriptor, T>` is derived from this class to inherit these members; the derivation is private because we are inheriting implementation and don't want to expose that fact. The three access declarations make the corresponding functions public in the derived class; `offset()` remains private.

The element access functions can be written using the offset computation service provided by `Subscriptor`:

```
template<class Subscriptor, class T>
T& ConcreteArray2d<Subscriptor, T>::
operator()(int s0, int s1) {
  return datap[offset(s0, s1)];
}

template<class Subscriptor, class T>
const T& ConcreteArray2d<Subscriptor, T>::
operator()(int s0, int s1) const {
  return datap[offset(s0, s1)];
}
```

`Subscriptor` is not at all like `T`; it's a variable service on which `ConcreteArray2d` relies.

Services for Class Templates. To implement these `Subscriptor` classes, we start by recognizing that one simple implementation for many such classes will share a common base of two integers giving the array shape; in fact, for `ndim`-dimensional arrays this will be `ndim` integers, so we define a template class parameterized by dimensionality:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \qquad \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{0,1} & a_{1,1} & a_{1,2} \\ a_{0,2} & a_{1,2} & a_{2,2} \end{bmatrix}$$

Array Symmetric Array

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$

Row-major storage layout

$a_{0,0}$	$a_{1,0}$	$a_{2,0}$	$a_{0,1}$	$a_{1,1}$	$a_{2,1}$	$a_{0,2}$	$a_{1,2}$	$a_{2,2}$

Column-major storage layout

$a_{0,0}$	$a_{0,1}$	$a_{1,1}$	$a_{0,2}$	$a_{1,2}$	$a_{2,2}$

LAPACK symmetric packed storage layout

Figure 1: Example array storage layouts. The array elements indicated on in top left array can be stored in continguous memory locations using, for example, either row-major or column major formats. The symmetric array on the top right could be stored in the packed storage layout is used by LAPACK [1] using just upper triangle of a symmetric matrix.

```
template<int ndim>
class ConcreteArrayShapeBase {
public:
  int shape(int dim) const { return _shape[dim]; }
  int dim()          const { return ndim; }
  int numElts()      const;
protected:
  int _shape[ndim];
  ConcreteArrayShapeBase() {}
};
```

The constructor is protected because this class should only be used as a base class, with the derived classes responsible for setting the shape:

```
template<int ndim>
class ConcreteArrayShape {
  // Dummy definition: supply a specialization
  // for each value of ndim.
};

class ConcreteArrayShape<2> :
  public ConcreteArrayShapeBase<2> {
public:
  ConcreteArrayShape(int s0, int s1) {
    _shape[0] = s0;
    _shape[1] = s1;
  }
};

class ConcreteArrayShape<3> :
  public ConcreteArrayShapeBase<3> {
public:
  ConcreteArrayShape(int s0, int s1, int s2) {
    _shape[0] = s0;
    _shape[1] = s1;
    _shape[2] = s2;
  }
};
```

We supply a template specialization for each dimensionality, so that the constructor can be given the appropriate number of arguments. (There is no way to say that the constructor should take `ndim int` arguments.)

Our `Subscriptor` services can now build on this base. For example, FORTRAN-compatible arrays might use this class:

```
class ColumnMajorSubscriptor2d :
  public ConcreteArrayShape<2> {
public:
  ColumnMajorSubscriptor2d(int s0, int s1) :
    ConcreteArrayShape<2>(s0, s1) {
  }
protected:
  int offset(int i, int j) const {
     return i + j * shape(0);
  }
};
```

Then we can create a FORTRAN-compatible array by supplying this service to the core array class template:

```
template<class T>
class ConcreteFortranArray2d : public
  ConcreteArray2d<ColumnMajorSubscriptor2d, T> {
public:
  ConcreteFortranArray2d(int s0, int s1);

  // Copy constructors and assignment operators
  // ...
};
```

A `RowMajorSubScriptor2d` would have a different `offset()` calculation that arranges the elements in row-major order:

```
class RowMajorSubscriptor2d :
  public ConcreteArrayShape<2> {
public:
  RowMajorSubscriptor2d(int s0, int s1) :
```

```
    ConcreteArrayShape<2>(s0, s1) {
  }
protected:
  int offset(int i, int j) const {
     return j + i * shape(1);
  }
};
```

We can then supply it to the `ConcreteArray2d` template to obtain a two-dimensional array with C- and C++-compatible storage layout:

```
template<class T>
class ConcreteCArray2d : public
  ConcreteArray2d<RowMajorSubscriptor2d, T> {
public:
  ConcreteCArray2d(int s0, int s1);

  // Copy constructors and assignment operators
  // ...
};
```

This pattern is easily extended to various packed storage layouts. For example, the LAPACK Fortran subroutine library for linear algebra [1] stores the upper triangle of a symmetric square matrix as shown in Figure 1. The corresponding `Subscriptor` can be written like this:

```
class SymPackedSubscriptor2d {
public:
  SymPackedSubscriptor2d(int s0) : n(s0) {}
  int shape(int) const { return n; }
  int dim()      const { return 2; }
  int numElts() const { return (n * (n+1)) / 2; }
protected:
  int offset(int i, int j) const {
    if (i <= j) return i + (j*(j+1)) / 2;
    else        return j + (i*(i+1)) / 2;
  }
private:
  int n;   // n x n matrix
};
```

Unlike the previous two cases, this `Subscriptor` is not derived from `ConcreteArrayShape<2>`; the properties of symmetric matrices make a simpler representation desirable. Nevertheless, `SymPackedSubscriptor2d` supplies the same names and functions as the other subscriptor classes, and therefore meets the needs of a `FortranSymPackedArray2d` class template:

```
template<class T>
class FortranSymPackedArray2d : public
  ConcreteArray2d<SymPackedSubscriptor2d, T> {
public:
  FortranSymPackedArray2d(int s0);

  // Copy constructors and assignment operators
  // ...
};
```

We have omitted some of the details necessary for practically useful arrays, but every detail we add to `ConcreteArray2d` will apply to all of the arrays we derive from it. Moreover, the pattern is simple to extend to higher dimensions, to arrays with subscripts starting at one, as in FORTRAN, to arrays with subscript range determined when an array is created, and to other storage layouts.

Class Template Servers. `ConcreteArray2d<Subscriptor, T>` is a client of the `Subscriptor` parameter and different `Subscriptor` types yield different arrays. But when we start using these arrays, they can become servers, providing just the names we need for function templates like our second `runningSum()`, the one that has a service parameter of `AnArray`. For example, if we create a `ConcreteFortranArray2d<double>` we can apply `runningSum()` to it like this:

```
  ConcreteFortranArray2d<int> a(2, 3);
  // Set array elements ...
  cout << runningSum(a) << endl;
```

We can do the same for `FortranSymPackedArray2d<T>` and so on because the whole group of arrays derive from our `ConcreteArray2d<Subscriptor, T>` and that template supplies the services that `runningSum()` needs.

Can we put our finger on the exact difference between a collection parameter like `T` above and a service parameter like `Subscriptor`? After all, the first version of `runningSum()` that we showed, the one parameterized by `T`, relies on services in `T` like `+=`. It's a matter of degree, but if we think about a template argument like a macro argument we call it a collection parameter; otherwise we call it a service parameter. The names aren't important, but getting beyond viewing templates as fancy macros is important.

Imagine trying to describe one of our array classes via macro expansion. When the `ConcreteFortranArray2d<double>` class is generated from the corresponding class template, it causes the `ConcreteArray2d<ColumnMajorSubScriptor2d, double>` to be generated. This, in turn, causes the `ColumnMajorSubScriptor2d` class to be needed. It requires `ConcreteArrayShape<2>`, which is supplied by template specialization instead of generation. Already we can see trouble: Adopting a macro-expansion view of templates leads to a complex view of this system, even before we imagine the expansion of the class member functions.

A view based on common names supplied by service classes and used by client classes or functions treats the relations between parts of a system of templates abstractly, avoiding the details that make the macro expansion view cumbersome. We view `ConcreteFortranArray2d<double>` as an array class that provides `EltT`, `shape()`, and `operator()()` to `runningSum()`. The array class happens to get these services from its base class, `ConcreteArray2d<Subscriptor, T>`. And, if we wish to look, the base class gets `shape()` from the `Subscriptor`We can delve as shallow or as deep as we wish, relating each level to the next in the natural abstraction for that level.

Two major features are missing in our array implementation: projections and iterators that visit every element. As these are added, the client/server parameter view becomes more essential to understand how the system works. For example, how shall we express projections? The offset between elements of the rows of a column-major 2D array will differ from the offset between elements of the rows for a row-major 2D array. Selecting the offset computation for the projection of a 2D array must be a service provided by the `Subscriptor`.Thus the natural way to talk about the first parameter for the template `ConcreteArray2d` is as a class that will provide `offset()` service. We have used this approach in [2] to implement the features missing from the simplified example presented here. Stroustrup [3, Section 8.7.1] has also used the same technique to supply a memory allocator to a container.

Static Polymorphism. We have argued that templates provide a means to supply services to a client by exploiting name commonality. How does this compare with using virtual functions? The big leverage in object-oriented programming comes from polymorphism: one piece of code being able to work with more than one type of object. Virtual functions—C++'s support for polymorphism—are very efficient compared to the support in other languages, but many scientific and engineering programs simply can't tolerate the space and time overheads of virtual functions, particularly since current compilers don't inline calls to virtual functions.

Templates do not have these overheads and yet they offer some of the benefits of polymorphism. For example, the function template `runningSum()` is one piece of code that works with many types of arrays; the class template `ConcreteArray2d` has members that work with many different subscriptors, each implementing a storage layout.

Of course this polymorphism is different than polymorphism provided by virtual functions. But the differences are smaller than the templates-as-macros view leads you to believe. As we have discussed here, client/server templates work together by expecting and offering agreed-upon names for types and functions. Virtual function systems do the same, with client functions expecting the virtual functions in a base class and derived classes supplying these services.

The central difference between virtual function polymorphism and template polymorphism is time: the piece of code that exploits virtual function polymorphism is compiled and the binding to multiple objects happens at runtime, while the piece of code that exploits template polymorphism is source code and the binding happens at compile time. For this reason, we call the polymorphism captured by templates *static polymorphism.*

With this in mind, we can predict when template polymorphism will not work as well as virtual functions: whenever we try to mix a bunch of different object types at runtime. The classic example is a list of objects held by base class pointers: we can't do this with templates. But conventional science and engineering programs don't often mix a bunch of object types at runtime. Much more often we transform object values—add a scalar to a vector—or pass values on to other object types—factor a matrix. So we can get many of the advantages of polymorphism and still generate fast code by using static polymorphism in templates. The balancing cost is increased compile time, larger code size, and more frequent need to recompile as your program evolves.

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++: An Introduction with Advanced Techniques and Examples. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[3] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, second edition, 1991.