Scientific and Engineering C++: Is Const Worth a Thousand Words?

John J. Barton Lee R. Nackman

May 2, 1994

C++ let's you define new types and it checks that all uses of an object are consistent with its type. But there's no free lunch: C++ can only check the use of new types against the declarations you provide. More specific declarations yield more effective type checking at the cost of more declarations, more complexity, and, perhaps, more compile time. The value of extra help from the compiler must be balanced against the extra declaration complexity required to tell the compiler how to help us.

If we balance the value of `const` against the extra work to declare `const` variables and member functions, most programmers will opt for the extra work. Programmers know that `const` is not a guarentee of read-only access (there is always cast-away `const`), but they use it anyway because `const` documents the actions of functions and protects against some silly errors. A `const` reference argument avoids copies of arguments that will be logically unchanged by a function.

However, built-in `const` references are not enough. As we build systems of objects, some objects will end up refering to the insides of other objects. The most obvious examples are structured components of large objects: substrings of strings, submatrices of matrices, subgraphs of graphs, and so on. These component objects need not be just built-in references, but programmer-defined classes. Marking these component objects `const` does not specify the nature of the objects they reference. In the same way that C++ pointers can be `const` or not while pointing to objects that are `const` or not, a non-`const` string can have `const` or non-`const` substrings. Thus programmer-defined classes with referencelike behavior faced an added challenge when `const` is introduced.

In this column we explore the impact of `const` on one kind of programmer-defined reference, the projections of arrays. Projections are fundamental to array manipulation: They let us get at the planes of a three-dimensional array, the rows or columns of a two-dimensional array, or the elements of a one-dimensional array. Except for the one-dimensional case, projections both refer to a subset of an array's elements and themselves act like arrays. Since they are used extensively (cf. [1]), projections must be compact, efficient, and easy to use.

The type checking question arises when we consider projections of `const` arrays. Suppose we have a two-dimensional array class, say `FArray2d<T>` (with the F standing for FORTRAN-compatible), and we want to provide `Row<T>` and `Column<T>` projection objects representing rows and columns respectively. We'll see later that we can either ignore the whole issue of `const`, forcing clients to forego the compiler's help in checking for unintentional updates of arrays, or that we need both a `const` and a non-`const` version of each kind of projection. So the trade-off is this: enhanced checking versus the complexity of twice as many projection classes.

In previous columns we have discussed array design issues, but we skirted around the issue of projections and other subarrays so we could avoid this nasty issue. Time to roll up our sleeves and dig in.

Projection Basics. Projections are subarrays obtained (conceptually) by holding one of the original array's subscripts constant while the others vary. To keep things simple, we'll focus on two-dimensional arrays with contiguous, column-major storage layout (i.e., FORTRAN-compatible layout). There is nothing magic about this choice

Here's our two-dimensional array template:

```
template<class T>
class FArray2d {
public:
  FArray2d(int nrows, int ncols);
  // ... other constructors
 ~FArray2d();

  const T& operator()(int, int) const;
        T& operator()(int, int);

  int shape(int d) const;
  // other array functions and operations
private:
  T*  _p;
  int _shape[2];
};
```

A corresponding projection object might look like this:

```
template<class T>
class Column {
public:
   Column(const FArray2d<T>& a, int col) :
      _shape(a.shape(0)),
      _p(&a(0, col)) {}
```

```
   const T&
     operator()(int i) const { return _p[i];  }
   int shape(int)      const { return _shape; }
   // other array functions
private:
   const int _shape;
   const T* const _p;
};
```

When constructed, the projection object squirrels away a pointer, `_p`, to the elements of the specified column for implementing subscripted element access and the number of rows, `_shape`, for implementing a shape query.

Const this to a const that. Having dashed off a projection class we jump to using it in a function. Suppose we choose to write a function called `runningSum()` that just adds up the elements of a two-dimensional array by looping over columns and calling a companion `runningSum()` function for columns:

```
template<class T>
T runningSum(const Column<T>& a) {
   T sum(0);
   for (int i = 0; i < a.shape(0); i++) {
      sum += a(i);
   }
   return sum;
}

template<class T>
T runningSum(const FArray2d<T>& a){
   T sum(0);
   for (int i = 0; i < a.shape(1); i++) {
      sum += runningSum( Column<T>(a, i) );
   }
   return sum;
}
```

This works just fine.

But now let's write a function that modifies columns of an array, say a column scaling:

```
template<class T>
void colScale(Column<T>& c, T scaleFactor) {
   for (int i = 0; i < c.shape(0); i++) {
      c(i) /= scaleFactor;
   }
}
template<class T>
void colScale(FArray2d<T>& a, T scaleFactor) {
   for (int i = 0; i < a.shape(1); i++) {
      colScale( Column<T>(a, i), scaleFactor );
   }
}
```

This code looks ok, but it won't compile. Our compiler gives two error messages. First it objects to initializing a non-`const` reference, namely the first argument of the one-dimensional version of `colScale()`, with a temporary object (cf. Section 8.4.3 of [2]). We can fix this error by changing to a `const` reference argument. Such a `const` would say that the `Column<T>` object itself can't be altered; it would say nothing about the elements of the underlying array. In other words, the `Column<T>` object behaves something like a pointer: a pointer itself can be `const` or, independently, the object to which it points can be `const`. Any of the four combinations are possible. File this observation away for a moment.

The compiler's second objection is to the `/=` operation, which attempts to modify an element through the `const` reference returned by `Column<T>`'s element access function. This function can't return a non-`const` reference to the element because `Column<T>` holds a pointer to `const` elements. We could cast away the `const`-ness, but this would eliminate all checking: a client could accidentally and silently modify a `const` array via a projection.

It looks like we're stuck with three bad choices:

- Change `Column<T>` to provide non-`const` access to the elements of the underlying array. This choice implies that we have to give up on writing functions, like `runningSum()`, that accept `const` reference arguments.

- Go the other way and make `Column<T>` read-only, as we've written it. With this choice, projections aren't generally useful. We've written our code this way and we can't even write a column scaling function.

- Cast away `const` inside of `Column<T>` so that it works with both `const` and non-`const` arrays. This choice dispenses with checking.

Before we give up, let's come back to our earlier observation that, like C++ pointers, a `Column<T>` could be `const` or not and it could refer to objects that are `const` or not. Unlike pointers, we have no need to alter a `Column<T>` object after we initialize it. This is reference-like behavior, with some `Column<T>` objects that can refer to columns of `const` arrays and some that can refer to columns of non-`const` arrays.

Built-in references work with both `const` and non-`const` objects, so we can use them as a pattern for our design. Consider:

```
int        i   = 5;
int&       ir  = i;
const int  ci  = 4;
const int& cir = ci;
```

If `i` models an array, `ir` models a projection; if `ci` models a `const` array, `cir` models a projection of a `const` array. There are two different reference types here: one `int&`, a reference to an `int`, and one `const int&`, a reference to a `const int`.

Based on this model, we plan two column classes, one for columns of `const` arrays and one for columns of non-`const` arrays. Call these `ConstColumn<T>` and `Column<T>`, respectively. Again, look at the built-in reference model:

```
int          i      = 5;
int&         ir     = i;
const int& cir_ir = ir;
```

A const reference can be initialized from a non-const reference, or, turned around, a non-const reference can be converted to a const reference. This makes sense because a const reference provides more restrictive access to an object than does a non-const reference. Thus, the built-in model suggests that we need to be able to convert a Column<T> to a ConstColumn<T>. This too makes sense: If we have a non-const column, we should be able to pass it to a function, like runningSum(), that expects a const column.

Finally, let's look at copying and assignment. Copying a reference means initializing one reference from another. No object is copied. So, when we initialized cir_ir from ir, the int object to which ir referred was not copied. Coding

```
int j = 10;
int k = 11;
const int& cjr = j;
int& kr = k;
kr = cjr;
```

causes the value of j to be copied into k; kr itself is not modified. There is no assignment of references.

Referencelike column objects. It's time to take another crack at writing column objects, this time using built-in references as our model. Here is the non-const version:

```
template<class T> class ConstColumn;

template<class T>
class Column {
public:
   Column(FArray2d<T>& a, int col) :
      _shape(a.shape(0)),
      _p(&a(0, col)) {}

   T& operator()(int i) const { return _p[i];  }
   int shape(int)       const { return _shape; }

   operator ConstColumn<T>() const;

   // other array functions
private:
   const int _shape;
   T* const  _p;
};
```

The constructor takes a non-const array reference argument, a signal that this projection may write into elements of the array. Sure enough, the operator()() function provides non-const reference to an array element when we subscript into the column's elements. With the conversion operator we can use an Column<T> where a ConstColumn<T> is expected; see below.

The operator()() might seem odd: it is a const member function but it returns a non-const reference to the selected element. Think back to the built-in reference model: the reference itself always acts like it is const even when it refers to a non-const object. Likewise, a Column<T> can be treated as const even when selected elements of the array it references are altered.

Neither a copy constructor nor an assignment operator appeared in our class, even though we are using pointer member data. The C++-generated copy constructor does what we need: Copying a Column<T> copies the count and pointer, not the array elements. C++ doesn't generate an assignment operator for Column<T> because both data members are const and therefore can't be altered. Again, this is the behavior we want.

The const version is much the same as the non-const version, with const everywhere the array or its contents appear:

```
template<class T>
class ConstColumn {
public:
   ConstColumn(const FArray2d<T>& a, int col) :
      _shape(a.shape(0)),
      _p(&a(0, col)) {}

   const T&
     operator()(int i) const { return _p[i];  }
   int shape(int)      const { return _shape; }
   // other array functions
private:
   ConstColumn(const T* p, int shape) :
      _shape(shape), _p(p) {}
   friend
     Column<T>::operator ConstColumn<T>() const;
private:
   const int      _shape;
   const T* const _p;
};
```

One new item is the second and private constructor; it is used by Column<T>'s conversion operator:

```
template<class T>
Column<T>::operator ConstColumn<T>() const {
  return ConstColumn<T>(_p, _shape);
}
```

Unlike built-in references this conversion involves runtime work—copying the count and the pointer—not just compile-time checking.

Now we can get back to the running-sum and column-scaling functions that started this whole discussion. Here they are, modified to work with our new classes:

```
template<class T>
T runningSum(const ConstColumn<T>& a) {
   T sum(0);
   for (int i = 0; i < a.shape(0); i++) {
      sum += a(i);
   }
   return sum;
```

```
}

template<class T>
T runningSum(const FArray2d<T>& a){
   T sum(0);
   for (int i = 0; i < a.shape(1); i++) {
      sum += runningSum( ConstColumn<T>(a, i) );
   }
   return sum;
}

template<class T>
void colScale(const Column<T>& c, T scaleFactor) {
   for (int i = 0; i < c.shape(0); i++) {
      c(i) /= scaleFactor;
   }
}
template<class T>
void colScale(FArray2d<T>& a, T scaleFactor) {
   for (int i = 0; i < a.shape(1); i++) {
      colScale( Column<T>(a, i), scaleFactor );
   }
}
```

Notice that the column arguments are always declared as `const` references to column objects, whether we intend to modify the elements of the columns or not. This looks a bit odd, but makes sense when you think about it: the column objects, like built-in references, are not modified; only elements to which they refer are modified. (With built-in references, there is no need to say "`const`" to keep the reference from being altered because references can't be altered.)

On the other hand, the array arguments are declared as `const` references when the array elements are not to be modified (`runningSum()`) and as non-`const` references when they are to be modified (`colScale()`). Again this makes sense: the `const` refers to the array itself, not to a reference.

The pair of `Column<T>` and and `ConstColumn<T>` illustrate the pattern of dual classes that support consistent `const` types when programmer-defined referencelike objects are created. But as array projections they are limited. Let's do a bit more.

Beyond columns. We've been concentrating on the column references and how they fit into the C++ type system, but haven't thought much about how the column references fit together with the arrays. It seems wrong to treat columns and rows separately: Our `Column<T>` project classes are not enough. At very least we need row projection as well, but a `Row<T>` class will not express the common behavior of row and column projections. For example, we shouldn't need separate code for column scaling and row scaling. This is easy to fix by generalizing our column references to become projection references.

Our column references assume a column-major array storage layout: the distance between elements of a column is one. More general projection references require storing a stride—the distance between elements—in the references:

```
template<class T>
class Proj {
public:
   static Proj<T> makeRow(FArray2d<T>& a, int row) {
     return Proj<T>(&a(row, 0),
                      a.shape(1), a.shape(1));
   }
   static Proj<T> makeCol(FArray2d<T>& a, int col) {
     return Proj<T>(&a(0, col), a.shape(0), 1);
   }

   T& operator()(int i) const
                         { return _p[i*_stride]; }
   int shape(int) const { return _shape;        }

   operator ConstProj<T>() const;
   // other array functions
private:
   Proj(T* p, int shape, int stride) :
     _shape(shape), _stride(stride), _p(p) {}
private:
   const int _shape;
   const int _stride;
   T* const  _p;
};
```

Rows and columns are distinguished only in construction, by calls to the appropriate static "make" function. These functions call the constructor with appropriate pointer, shape, and stride information; since they are the only way to make `Proj<T>` objects, the constructor is `private`. A `const` version of the class would be analogous.

In a revised array class we can provide both `row()` and `col()` members:

```
template<class T>
class FArray2d {
public:
  FArray2d(int nrows, int ncols);
  // ... other constructors
 ~FArray2d();

  const T& operator()(int, int) const;
        T& operator()(int, int);

  int shape(int d) const;

  ConstProj<T> row(int i) const;
  Proj<T>      row(int i);
  ConstProj<T> col(int i) const;
  Proj<T>      col(int i);
  // other array functions and operations
private:
  T*  _p;
  int _shape[2];
};
```

The `row()` and `col()` member definitions all call the correponding "make" function. Here is one of them:

```
template<class T>
Proj<T> FArray2d<T>::row(int i) {
  return Proj<T>::makeRow(*this, i);
}
```

Now we can rewrite the client functions with a simpiler syntax for taking projections. Here's one example:

```
template<class T>
T runningSum(const FArray2d<T>& a){
   T sum(0);
   for (int i = 0; i < a.shape(1); i++) {
      sum += runningSum( a.col(i) );
   }
   return sum;
}
```

We also can deal with rows and columns uniformly:

```
template<class T>
void scale1d(const Proj<T>& c, T scaleFactor) {
   for (int i = 0; i < c.shape(0); i++) {
      c(i) /= scaleFactor;
   }
}
```

Summary. We now have array projections for FORTRAN-compatible arrays that are convenient to use, efficient, and consistent with C++'s type checking. Many extensions are needed to have a consistent family of multi-dimensional array classes. Projections should behave like lower-dimensional arrays and should work with arrays with various storage layouts. Chapter 13 of [3] uses the subscriptor approach described in our last column (May '94) to develop code covering row-major, column-major, and some packed storage layouts. Other kinds of array references, like diagonals and block sub-arrays, are also useful.

As an example of programmer-defined referencelike objects, projections illustrate the issues added by `const` types. Once we understood the pattern, the cost of `const` was two projection classes instead of one. The same pattern can be applied to substrings, submatrices, subgraphs and so on. Other cases of programmer defined reference-like objects are not so easy to recognize as such, but any interior pointer or reference problem may face this issue.

This leads us back to the title of this column: is `const` worth the extra work? We think so. We get the important checks provided by `const` objects, especially `const` reference arguments to functions. But, even more important, our programmer-defined references—and that's what array projections are—fit into C++'s type system. In this case, we think that the declaration complexity is repaid by the checking gained and by the consistency with C++.

References

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD, second edition, 1989.

[2] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

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