This is a pre-publication draft of the column I wrote for the February 1996 issue of the *C++ Report*. "Pre-publication" means this is what I sent to the *Report*, but it may not be exactly the same as what appeared in print, because the *Report* and I often make small changes after I submit the "final" draft for a column. Comments? Feel free to send me mail: smeyers@aristeia.com.

# Interface Types Revisited (and More)

Okay, okay, I take it back! Geez, they put those hot buttons in the oddest places.

In my September column, I argued against the use of unsigned types in class interfaces. I said the use of unsigned types decreases the ability of class implementers to detect client errors that are not uncommon. I also wrote that unsigned types, when used as return values, themselves lead to likely client errors.

As an example, I considered the following template:

```
template<class T>
class Array {
public:
  Array(unsigned int size);
  ...
  unsigned int length() const;
};
```

I criticized the constructor, because I claimed there was no way to detect when `size` was initialized with a negative number, and I disliked `length`'s return value, because I said it was likely the caller would store the return value in an `int` and thereby run the risk of overflow. I suggested using signed `int`s in both cases, thus trading a bit of representation for increased error-checking ability (in the constructor) and reduced error introduction (for the `length` function).

I didn't expect this advice to be particularly controversial, but I was mistaken. I received an unusual amount of mail, and the writers were about equally split between people thanking me for publishing what they've believed for years and people patiently pointing out why they thought I was a dufus. After all was said and done, I concluded that I was indeed a dufus, but at least I wasn't an *obvious* dufus.

An ideal implementation of the `Array` constructor would declare its parameter as an `unsigned int` (because negative values make no sense), but would be able to reliably verify within the constructor that no negative value was actually passed. One writer proposed this solution:

```
template<class T>
Array<T>::Array(unsigned int size)
{
  if (static_cast<int>(size) < 0)
```

```
    do something appropriate for a negative
    size value;

  ...

}
```

The thinking here is reasonable enough. Assuming an `int` was passed in, cast `size` back to the `int` it used to be, then see if that `int` was negative. Alas, this code doesn't necessarily do what we want it to, because the draft C++ standard says that if an `unsigned int` with a value greater than `MAX_INT` (an implementation-defined value found in `<limits.h>`) is cast to an `int`, the resulting value is implementation-defined. In other words, if `size` is initialized with a negative value, there is no guarantee that `static_cast<int>(size)` will yield the number originally passed in. Thus this cast-based solution won't work.

There is a way to do what we want to do, however, and it was brought to my attention by both Dan Saks and Bear Giles. The trick is to avoid the cast from an unsigned to a signed `int`. Instead, compare `size` against `MAX_INT`. If `size` is greater, the value with which `size` was initialized must have been negative. In other words, implement the check inside `Array`'s constructor like this:

```
template<class T>
Array<T>::Array(unsigned int size)
{
  if (size > MAX_INT)
    do something appropriate for a negative
    size value;

  ...

}
```

This should be portable. Of course, this also limits the maximum value that can be passed into the `Array` constructor to `MAX_INT`, but the solution I proposed in my September column (declaring `size` to be of type `int`) suffers from that drawback, too.

Now that I've been shown how to check for negative values when a function's parameter is unsigned, I believe that is the best way to declare function parameters when negative values make no sense. But it's not perfect. Dawn Koffman summarized the downside in her mail:

> If you're using an unsigned, how do you *communicate* to the caller that they can only use the lower half of the numeric range of the unsigned? As I see it, using an unsigned communicates that negative values are inappropriate (which is what you want to communicate), but it also communicates that the value can have twice the magnitude in the positive direction (which in this case you *don't* want to communicate). In practice, miscommunicating about the upper half of the range for long unsigned values is unlikely to be an issue, but I'd be concerned about the miscommunication when it comes to short unsigned values.
>
> I think in general what makes programming hard is that we're forced to use the same language to communicate with compil-

ers *and* with the humans who'll be using and maintaining our code. It's very difficult to communicate so that we're completely understood by both of these very different systems (who have very different jobs to do), at least given the kinds of languages we have these days.

Hmmm, anybody for a new language?

Once we've decided to use unsigned parameter types but to accept only values in the range [0, `MAX_INT`], we can safely start to return unsigned values from other functions, because the draft standard guarantees us that it's portable to convert an `unsigned int` to an `int` provided the value of the unsigned int is within the range of the `int`. We can thus have `Array::length` return an `unsigned int` (to emphasize that it will never return a negative value), but still be assured that all is well if — as is likely — the caller treats the return value as an `int`.

I am grateful to all who wrote, and especially to Dan and Bear for sharing their solution to the unsigned-values-in-interfaces problem.

**Arrays and Polymorphism**

With that behind us, let us move on to a different topic, that of using arrays polymorphically. One of the most important features of inheritance is that you can manipulate derived class objects through pointers and references to base class objects. Such pointers and references are said to behave *polymorphically* — as if they had multiple types. C++ also allows you to manipulate *arrays* of derived class objects through base class pointers and references. This is no feature at all, because it almost never works the way you want it to.

For example, suppose you have a class `BST` (for binary search tree objects) and a second class, `BalancedBST`, that inherits from `BST`:

```
class BST { ... };

class BalancedBST: public BST { ... };
```

In a real program, such classes would be templates, but that's unimportant here, and adding all the template syntax just makes things harder to read. For this discussion, we'll assume `BST` and `BalancedBST` objects contain only `int`s.

Consider a function to print out the contents of each `BST` in an array of `BST`s:

```
void printBSTArray( ostream& s,
                    const BST array[],
                    int numElements)
{
  for (int i = 0; i < numElements; ++i) {
    s << array[i];       // this assumes an
  }                      // operator<< is defined
}                        // for BST objects
```

This will work fine when you pass it an array of `BST` objects:

```
BST BSTArray[10];

...
```

```
printBSTArray(cout, BSTArray, 10);// works fine
```

Consider, however, what will happen when you pass `printBSTArray` an array of `BalancedBST` objects:

```
BalancedBST bBSTArray[10];

...

printBSTArray(cout, bBSTArray, 10);
                                  // works fine?
```

Your compilers should accept this function call without complaint (though I know of one compiler that will erroneously reject it), but look again at the loop for which compilers must generate code:

```
for (int i = 0; i < numElements; ++i) {
  s << array[i];
}
```

Now, `array[i]` is really just shorthand for an expression involving pointer arithmetic: it stands for `*(array+i)`. We know that `array` is a pointer to the beginning of the array, but how far away from the memory location pointed to by `array` is the memory location pointed to by `array+i`? The distance between them is `i*sizeof(`*`an object in the array`*`)`, because there are `i` objects between `array[0]` and `array[i]`. In order for compilers to emit code that walks through the array correctly, they must be able to determine the size of the objects in the array. This is easy for them to do. The parameter `array` is declared to be of type array-of-`BST`, so each element of the array must be a `BST`, and the distance between `array` and `array+i` must be `i*sizeof(BST)`.

At least that's how your compilers look at it. If you've passed an array of `BalancedBST` objects to `printBSTArray`, your compilers are probably wrong. In that case, they'd assume each object in the array is the size of a `BST`, but each object would actually be the size of a `BalancedBST`. Derived classes usually have more data members than their base classes, so derived class objects are usually larger than base class objects. We thus expect a `BalancedBST` object to be larger than a `BST` object. If it is, the pointer arithmetic generated for `printBSTArray` will be wrong for arrays of `BalancedBST` objects, and there's no telling what will happen when `printBSTArray` is invoked on a `BalancedBST` array. Whatever does happen, it's a good bet it won't be pleasant.

The same problem arises in a different guise if you try to delete an array of derived class objects through a base class pointer. Here's one way you might innocently attempt to do it:

```
// delete an array, but first log a message
// about its deletion
void deleteArray(ostream& logStream, BST array[])
{
  logStream  << "Deleting array at address "
             << static_cast<void*>(array)
             << '\n';

  delete [] array;
}
```

```
// create a BalancedBST array
BalancedBST *balTreeArray = new BalancedBST[50];

...

// log its deletion
deleteArray(cout, balTreeArray);
```

You can't see it, but there's pointer arithmetic going on here, too. When an array is deleted, a destructor for each element of the array must be called. When compilers see this statement,

```
delete [] array;
```

they must generate code that does something like this:

```
// destruct the objects in array in the inverse
// order in which they were constructed
for ( int i = the number of elements in the array - 1;
      i >= 0;
      --i)
  {
    array[i].BST::~BST();     // call array[i]'s
  }                           // destructor
```

Just as this kind of loop failed to work when you wrote it, it will fail to work when your compilers write it, too. The language specification says the result of deleting an array of derived class objects through a base class pointer is undefined, but we know what that really means: execution of the code is almost certain to lead to grief. Polymorphism and pointer arithmetic simply don't mix. Array operations almost always involve pointer arithmetic, so arrays and polymorphism don't mix.

Note that you're unlikely to make the mistake of treating an array polymorphically if you avoid having a concrete class (like BalancedBST) inherit from another concrete class (such as BST). Designing your software so that concrete classes never inherit from one another has many benefits. For details, see my columns in the July-August 1994, November-December 1994, and January 1995 issues of this magazine. Alternatively, you may consult Item 33 of my new book, *More Effective C++* (Addison-Wesley, 1996). For information on this particular example (i.e., having a BalancedBST class inherit from a BST class) , see Martin Carroll's article on Invasive Inheritance in the October 1992 *C++ Report*. You may also consult section 3.4 of Carroll's and Margaret Ellis' *Designing and Coding Reusable C++* (Addison-Wesley, 1995).

**Acknowledgement**

My warning about deleting an array of derived class objects through a base class pointer is based on material in Dan Saks' "Gotchas" talk, which he's given at several conferences and trade shows.