N3810
Bjarne Stroustrup
2013-10-13

Alternatives for Array Extensions

Bjarne Stroustrup

The most important aspect of an "array alternative" is that it enables simple and efficient use of a run-time-specified amount of stack storage and provides an alternative to the traditional problems of "lost" size and "lost" type that make direct use of arrays error-prone.

Here, I will record some reflections based on the array/dynarray discussions at the Chicago meeting with some brief analysis of the alternatives as a start of a discussion. I consider a resolution urgently needed. We need arrays with run-time-specified bounds and safer access to such storage "yesterday."

VLAs and ARBs

The Array of Run-Time Bounds (ARBs, N3497 *Runtime-sized arrays with automatic storage duration* by Jens Maurer) is a great simplification of, and improvement over, C's Variable-Length Arrays (VLAs). First of all, an ARB is a conventional C-style array:

```
const int n1 = 99;
void f(int n2)
{
        int a1[n1];         // an array (traditional)
        int a2[n2];         // an array (ARB)
        int* p = a1;
        p = a2;             // a1 and a2 are of the same type
}
```

That is, a programmer can use an array without having to know whether the expression specifying the number of elements was a constant or a variable. In addition, ARBs do not imply separate syntax or new rules for declarations and argument passing.

However, being arrays, ARBs suffer the usual problems associated with their number of elements not being available unless the user specifically "remembers" it and the type used to access elements can implicitly change after the array name has decayed to a pointer. For example:

```
void draw_n(Shape*p, int n)     // poor code/interface
{
        p[n].draw();
}
```

```
void g(int n)
{
        Circle ac[n];
        Shape* p = ac;          // array decay and Derive* to Base* implicit conversion
        ac[7].draw();           // decay and subscripting (no range check)
        p[7].draw();            // disaster: wrong offsets (and no range check)
        draw_n(ac,n/2);         // ask for disaster
}
```

I assume that **Circle** is publicly derived from **Shape** and that **sizeof(Circle)>sizeof(Shape)** so that subscripting a **Shape*** give different offsets from subscripting a **Circle***. Yes, this is poor code, but it is free of casts and unions, compiles without warnings, and has potentially disastrous effects. Unfortunately, neither (pointer,count) interfaces nor arrays nor class hierarchies are uncommon. Stronger: (pointer,count) interfaces are common, array decay is fundamental to much C-style code, and implicit Derived* to Base* conversion essential for much object-oriented programming.

If we have only ARBs for stack storage, these problems are not addressed, (pointer,count) interfaces will become more common (and probably "baked into" ABIs), and we will have encouraged a lowering of the level of programming from the use of containers and algorithms to the use of arrays and pointers. This is a topic of major importance because the importance of stack storage (and any other storage that works well with concurrency and doesn't require synchronization for allocation and deallocation) is increasing.

ARB has an obvious weakness: we cannot ask an ARB for its size. Thus, we need "special wording" to make a range-for work for an ARB and we can't easily pass [**begin(a)**,**end(a)**) pairs to an algorithm. We need a simple and elegant way to place elements on the stack and pass some form of reference to another function (e.g., an algorithm). VLAs provide **sizeof()** to find the end of allocated memory, but I consider that seriously flawed – it reports the size in bytes (chars) rather than in number of elements.

```
void f(int n);
{
        Circle ac[n];
        draw_n(ac,sizeof(ac)/sizeof(*ac));
}
```

This may be familiar to many, but it is error-prone (not everyone uses **sizeof(ac)/sizeof(*ac)**), verbose, and if **draw_n()** expects a pointer into an array **Shape**s that call is still wrong. The "sizeof trick" is familiar to C programmers, but for people brought up with just about any other language it is seen as a weird flaw. We should not confuse "familiar" with "simple."

We introduced **std::array** to allow people to handle such problems for **T[n]** when **n** was a constant. We need something at least as elegant for **T[n]** when **n** is a variable. When we accepted ARBs that was the basis for our consensus – ARBs on their own would not have passed.

An class interface for stack memory

So, we need a higher-level, less error-prone interface to stack memory to complement ARBs for people who prefer not to deal directly with arrays. The crucial properties of such an interface is

- It will be on the stack (or equivalent) for a local object
- It does not implicitly "decay" to a pointer
- We can ask it for its number of elements (and/or its end).

This is essential and must not be lost as a design is adjusted to meet other/further needs or to address problems.

For example, there are architectures on which the amount of stack space is severely limited. Thus, there have been designs where the elements end up being allocated using **new** when the amount of stack is deemed insufficient. There are problems with such a design:

- Using **new**/**delete** leaves the user vulnerable to locking problems, fragmentation problems, and inefficiencies. This approach is not generally acceptable for embedded systems.
- Ideas of dynamically deciding how much stack is left for an array are impractical in the face of hardware and fundamental software restrictions

On the other hand, using a "side stack" for array elements can be acceptable (a "handle" is left on the call stack as the local object while the elements are elsewhere).

I assume that array overflow problems are handled identically for an array interface class and an ARB.

Dynarray

Aka C++ Dynamic Arrays described in N3532 - 2013-03-12 by Lawrence Crowl and Matt Austern was voted into the Working paper, but suffered national body (and other) objections for not guaranteeing stack allocation, not providing full allocator support, and/or suffering constructor ambiguities . N3532 explains the basic ideas:

> "Instead of adopting C variable-length arrays, we propose to define a new facility for arrays where the number of elements is bound at construction. We call these dynamic arrays, `dynarray`. In keeping with C++ practice, we wish to make `dynarray`s usable with more than just automatic variables. But to take advantage of the efficiency stack allocation, we wish to make `dynarray` optimizable when used as an automatic variable.
>
> Therefore, we propose to define `dynarray` so that compilers can recognize and implement construction and destruction directly, without appeal to any particular standard library implementation. However, to minimize the necessary burden on compilers, we propose that `dynarray` can be implemented as a pure library, although with lost optimization opportunity.
>
> We believe that the compilers can introduce the optimization without impact on source or binary compatiblity. There may be some change in code profiles and operator new calls as a result of that optimization, but such risks are common to compiler and library upgrades.

> Syntactically, our proposal follows the lead of `std::array` and `std::vector` containers. Semantically, our proposal follows the lead of built-in arrays. That is, we do not require more out of `std::dynarray` element types than we do of standard array element types."

My reading of the facts and of the opinions I have heard expressed is that the desire to support **dynarray** as a general container (e.g., allocator support) let to a larger than expected class, complicated optimizations, and distracted from its (IMO) primary role as an accessor of stack-allocated objects. As time passed, **dynarray** accreted aspects of "an ordinary container" (leaving only its fixed size after construction to distinguish it from **vector**), and implementations focused on that and failed to provide the crucial optimizations for local objects. That "crucial optimization" requires compiler support.

This led to a desire among some (notably me) for bringing **dynarray** back to basics.

Bs_array

In discussions, I have mentioned that what I thought we needed was a "basic stack-allocated array type that relates to an array with dynamic extent like **std::array** relates to an array with static extent." Here is a minimal outline:

```
template<class T>
class bs_array {            // bike shed issue: find a proper name for a basic stack allocated array
        using value_type = T;

        bs_array(int n);                        // n elements

        // default copy, no move, maybe copy to vector/other_containers

        T& operator[](int i);                   // ith element
        const T& operator(int i) const;         // ith element
        T& at(int i);                           // ith element, range check
        const T& at(int i) const;               // ith element, range check

        T* begin() { return a; }                // or define externally
        const T* begin() const { return a; }
        T* begin() { return a+n; }
        const T* begin() const { return a+n; }

        int size();
        T* data();
private:
        T a[n];   // for exposition only, a is stack allocated
};
```

The point is that a **bs_array** can be allocated on the stack only, does not implicitly "decay" to a pointer and does not forget its number of elements. You can safely pass a reference to a **bs_array**, you can run a range-for loop over it.

Like **dynarray**, implementing a **bs_array** requires "compiler magic" so it has to be a compiler-supported standard-library construct. In particular, I do not consider a (pointer,size) representation plus a **new**/**delete** implementation acceptable (in this, **bs_array** differs from **dynarray**). Use of the general free store implies the use of synchronization in a multithreaded environment and the possibility for fragmentation. Neither is acceptable in a range of likely application, including some high-performance computing and many embedded systems. A "side stack" for elements of a **bs_array** would be an acceptable implementation technique for high-performance and embedded systems. Its use would imply little overhead, would not consume much call stack, would imply no added locking, and would not cause fragmentation. I consider this common to all acceptable implementations. I imagine the use of a "side stack" to be an implementation detail, depending on implementation concerns, rather than programmer choice for an individual object.

Unlike **dynarray**, **bs_array** does not try to be a general container. In particular, it is not meant for member variables.

Quoting Matt Austern:

"the argument about **dynarray** that I found most convincing was Richard Smith's: what if a user writes

```
dynarray<int> a(10);
f(a);
// do something more with a
```

and the definition of f is

```
void f(dynarray<int>& a) {
   a.~dynarray<int>();
   new(a) dynarray<int>(50);
}
```

If `dynarray` is anything like a normal C++ class, this is clearly legal code. Conversely: if we want non-heroic compilers to be able to implement `bs_array` without dynamic allocation, we need to find some way of saying that code like this is illegal. (Maybe: you can't take the address of or form references to a `bs_array`?)"

In a reflector thread, Chandler Carruth provided a "nice" collection of examples that would lead to disaster with **dynarray** and any other container that tried to be both a general container and ensure stack allocation. I present(ed) **bs_array** as an alternative that doesn't suffer such problems.

I thought/think of **bs_array** as an experimental minimal class to supplement ARBs. Obviously, code like **f()** has to be avoided. I don't care much how that is done. The **bs_array** constructor is (obviously?) "magic," and I suppose the destructor must also be. Only their conventional use for a local variable are acceptable.

Explicit Arrays

In EWG at the Chicago meeting, the idea of a modifier to an array declaration controlling its decay was aired (in several variations). Consider:

```
void f(int n)
{
        explicit Circle ac[n];              // bike shed warning!
}
```

Here, the prefix **explicit** indicates that **ac** decays to an accessor object holding (**&ac[0],n**). Here, I will use the name **array_ref** for the accessor:

```
void g(array_ref<Circle> c) {
{
        for (auto& x : c)
                x.draw();
        for( auto p = c.begin(); p!=c.end(); ++p)
                p->draw();
        c[7].draw();
        Shape* p = &c[7];         // on your head be it
}

void f(int n)
{
        explicit Circle ac[n];
        Shape* p = ac;                      // error: explicit arrays do not decay to pointers
        Shape* q = &ac[0];                  // OK: pointers convert
        array_ref<Circle> r = ac;           // OK: explicit arrays converts to (decays to) array_refs

        g(ac);

        for (auto& x : ac)
                x.draw();
        for( auto p = ac.begin(); p!=ac.end(); ++p)
                p->draw();
}
```

Consider some obvious questions:

- Why **explicit**? In the EWG discussions several alternative notations were suggested, including a suffix **class** (because the array behaves a lot like a class) and a declarator operator to combine with **[]** as opposed to a "storage class specifier" applying to a declaration containing a **[]**. I am currently sticking with **explicit** because, like other uses of **explicit**, it inhibits a conversion.
- Why **array_ref**? Because that name is descriptive; it refers to an array – it has reference semantics. It seems that the ideal accessor type is very close to **array_ref**s that has been in use in various places.

- What about multidimensional arrays? I see no problem with multidimensional arrays as long as only one dimension is specified at run time. That's why I use the "storage class specifier" **explicit**, rather than some modification of the declarator operator **[]**.
- How does subscripting work? Since there is no array decay, the formal description of how **[]** works on an explicit array has to be a bit different, but the implementation can be identical to subscripting other arrays. The **array_ref** class needs a **[]** operator.

This solution differs from the **dynarray** and **bs_array** solutions by

- Having a cleaner separation between compiler and library concerns
- By not providing a container (or "almost container") class.

If you want a container for elements on the stack, you must build it out of **array_ref**s. For example:

```
void f(int n)
{
        explicit int a[n];
        bs_array ba {a};
        dynarray da {a};
        // …
}
```

I fear that the notational overhead of having to use two definitions would be unpopular, and a barrier to use of the containers as (opposed to **array_ref**).

Explicit arrays are most useful on the stack, but they don't need to be just for local objects. We could generalize the idea to allow **explicit** for every array to get namespace and member explicit arrays (in which case explicit would probably have to be a suffix). I suspect such a generalization to be "mission creep." I suspect that most uses will not be of elements that may or may not be on the stack. Rather, the programmer wants to assure that elements are on a stack or on stack-like storage if the machine architecture is not suitable for large arrays on the call stack.

Naturally **explicit** would apply both to ARBs and arrays with constant bounds:

```
const int n1 = 99;

void f(int n2)
{
        int a1[n1];         // an array (traditional)
        int a2[n2];         // an array (ARB)
        int* p = a1;
        p = a2;             // a1 and a2 are of the same type

        explicit int a3[n1];        // an explicit (traditional) array
        explicit int a4[n2];        // an explicit array (ARB)
        array_ref<int> q = a3;
        q = a4;                     // a3 and a4 are of the same type
                                    // (if we allow assignment to array_ref)

        p = q;    // error: a1 and a3 are of different types
        q = p;    // error
    }
```

Array Constructors

In a message to the –ext reflector, J. Daniel Garcia suggested a solution based on allocation of an array member in the scope in which its class object is created:

> What we need is:
> a) Allow an ARB data member of unspecified size.
> b) Allow to set the size of the ARB in construction.

```
template <class T>
class bs_array {
public:
  bs_array(int n) : v[n]{}, sz{n} {}
  // ...

private:
  T v[];           // Alternate syntax could be T[] v;
  int sz;
};
```

I see this as a generalization of ARBs. We get:

```
class Array {
public:
        Array(int s) : sz{s}, elem[s]{}
        {
                // ...
        }
        int size() { return sz; }
        // ...
private:
        int sz;
        double elem[];
};

void f(int n)
{
        Array as {n};                   // elements on stack
        Array* p = new Array{n};        // elements on heap
        Static Array sas {n};           // elements in static storage
}
```

As for **explicit** arrays, generalizing to heap and static storage may or may not be desirable. The **elem[s]{** element initializer **}** syntax seems fine and is unambiguous. The **T elem[];** syntax may clash with C's empty array bounds, so Daniel's **T[] elem;** alternative may be preferable.

An array constructor allocates its array elements in the memory of its surrounding scope. If that scope must be a stack scope, we have something roughly equivalent to the "explicit array" solution.

If this approach is preferred, I we can define **dynarray** or **bs_array** using array constructors. Now instead of the **dynarray** class being "magic," it is the constructors and destructor that are. Like for "explicit arrays," the separation between library and language concerns seems cleaner.

Daveed suggests that the mapping to underlying storage should be described entirely within the constructor declaration; not the data members nor the definition. E.g.:

```
struct MyArray {
  MyArray(int n) double storage[n];
  int size() const { return s; }
private:
  MyPtr<double> p;
  int s;
};

MyArray::MyArray(int n): s(n), p(storage) {}
```

This cleanly separates the implementation from the interface. The role of the "**`double storage[n]`**" is to provide sufficient information in the constructor declaration for an implementation to allocate sufficient memory with seeing the constructor definition.

I like the idea of specifying "all magic" in one place, but I can't say that I find the syntax intuitive or attractive. It could be seriously messy for a class with many such arrays. Maybe we could simplify by requiring the constructors to be defined inline if we have an ARB member:

```
struct MyArray {
  MyArray(int n, int m) :ad{n}, ac{m} {}
  int size() const { return s; }
private:
  double ad[];
  int ac[];
};
```

We already have "special rules" for constructors of classes with **const** or reference members. I like this notation for its simplicity. One problem would be that changing a bound from a constant to a variable would force inline definition, but that seems minor compared to adding new syntax.

Final Thoughts (for now)

So, which solution do I like best? "Array constructors" by a small margin over "explicit arrays" provided we can have the/a simple syntax for "array constructors." For "explicit arrays" I would need more use cases to be sure. Furthermore, we need to decide which standard container(s) – if any – we would supply with array constructors (**dynarray**? **bs_array**?). How we could build a container out of explicit arrays? I suspect we should *not* expand the scope of a solution from stack storage to members, free store, and static memory.