

## Re: Teaching new tricks to an old dog (C++ -->Ada)

*Source:* [http://coding.derkeiler.com/Archive/C\\_CPP/comp.lang.cpp/2005-03/3290.html](http://coding.derkeiler.com/Archive/C_CPP/comp.lang.cpp/2005-03/3290.html)

---

*From:* Georg Bauhaus ([bauhaus\\_at\\_futureapps.de](mailto:bauhaus_at_futureapps.de))

*Date:* 03/22/05

Date: Tue, 22 Mar 2005 23:00:09 +0100

Ioannis Vranos wrote:

> *Georg Bauhaus wrote:*

>

>> *Hm. Searching a \*map\* of entries at numeric keys is different*

>> *from scanning an array of values and counting occurrences. What*

>> *are you trying to do here?*

>

>

>

> *Just using the appropriate available container. :-)*

Nope.

>> *The std::vector is missing an instantiation argument which adds*

>> *the guarantee that no index value is outside the range*

>> *-800\_000..12\_000\_000;*

>

>

>

> *vector provides method at() that performs range checking.*

I think we've been through this. Again, this is not the point.

> *Also if you*

> *want a vector that has signed integer subscripts or even floating point,*

We want a vector type indexed by values between M and N only \*and\* we want the compiler + tools to help us make data structures in accord with these ranges. We want it to take advantage of what it can learn from the type system which allows index range checking \_at compile time\_.

>> *(How do you make a subrange of double, which is missing from*

>> *your example.)*

>

>

>

>

> *Do you mean like this?*

No. I mean a double subtype whose values range from N.M to N'.M'.

I think you are missing the point here. As someone else said this is not about functions operating on containers.

> *[STL for constructs] These are bullet-proof code approaches.*

Unfortunately, this is not relevant in this discussion, which is not just about algorithms.

>> *Imagine an array shared between a number of threads. The program's*

>> *task is to count the number of occurrences of a particular value*

>> *in the array. Examples:*

>> *1) A shop has 10 unique doors (use an enum).*

> *We can use whatever I like, perhaps strings. What is wrong with "Door 1"*

> *etc? :-)*

I see the smiley but this is what might be wrong with your approach: a map indexed by strings is not the same as an array indexed by ten and only ten numbers. (Actually the numbers are in a type which includes only 10 numbers and their operations.)

Imagine a Matrix. String indexing of matrix element seems possible but you'd rather have proper arrays. There is something wrong with ("Row 5", "Column 1") indexing unless you have very much time and a lot of RAM.

>> *For each door 4 states*

>> *can be measured: open/closed, van/no van at the door.*

>

>

>

> *OK, this sounds easy. What do you think? Since you want an array:*

> *bool operator() (const Door &arg)*

Nice. However, operator() doesn't return one out of four states, it returns a Boolean, but o.K.

(If the door is closed and there is a van, you still want some action to take place. Likewise for the other cases. I know you can write this, no need as this is not the point, see below)

> {

> *return open == arg.IsOpen() && van == arg.IsVan();*

> }

> *vector<Door> doors(10);*

Here you can see one point that you might want to demonstrate:  
The compiler won't tell you that there is something wrong  
with

```
doors[10].SetOpen().SetNoVan();
```

Worse, the program won't tell you either. This shows the missing  
link between vector indexing and the base type system in your  
approach. You could use

```
doors.at(10).SetOpen().SetNoVan();
```

and handle the exception `_at` at run time.

In Ada, the compiler will tell you: "index value 10 is no good!"  
because the array "doors" can only be indexed by values effectively  
between 0 .. 9. These and only these are the values of the type  
enumerating the ten doors, and only these are allowed as index  
values `x` in expressions `doors(x)`.

No exception handling, no `.at()` needed when you listen to your  
compiler and fix the indexing error before you deliver a program.  
You get this for free as a result of the language's type handling  
at compile time.

```
>> 2) A 5-player team, each team is identified by number drawn from a fixed
>> set of team numbers. An array (an array, not some other data
>> structure) measures the number of players from each team present in
>> a room. Count the number of odd-team players in a room.
>
>
>
> This is easy too. I do not get your point. This can be done with arrays
> as also with other containers. Why should we be restricted to one type
> of container?
```

There is a reason that arrays still exist. One of the reasons  
should be obvious when `comp.realtime` is on the recipient list.  
Again, imagine a wave file manipulation process.  
A map indexed by strings is probably not the recommended container  
when you need fast matrix computations. In fact, a map might not be  
suitable at all irrespective of its key type, when r/w should be in  $O(1)$ .

- Given an enum, and
  - given a language that allows the enum as a basis for the construction  
of an array type in the type system (not using some run time computation  
method, like those you have shown here, IINM)
  - given that the compiler can use its knowledge of the enum
    - + when it sees an array type based on the enum
    - + when it sees an array
    - + when it sees an array indexed by a statically known enum value
    - + etc.,
- you have

comp.lang.c++. Re: Teaching new tricks to an old dog (C++ -->Ada)

- (a) useful names for objects in your problem domain, checked at compile-time
- (b) a conceptual link between the enum (naming the single items) and a container `_type_` (containing these items); you cannot use anything but these named numbers for indexing
- (c) the fastest possible access, for both reading and writing, possibly checked at compile time
- (d) etc.

The STL descriptions provide further reasoning why there can be restrictions on the uses of specific containers in specific situations, viz.  $O(f(n))$ .

>> *I hope these examples illustrate some points. They are not meant to trigger a discussion as to whether an array is the best data structure for everything. (Note that it might be necessary to read values from the array/Vector using random access in  $O(1)$ , and to store and replace values in  $O(1)$ , another reason to use an array.)*

What if a compiler or other tool can show that in the following expression (pseudo notation)

```
array_variable.at_index [n + m] <- f(x)
```

does not need an index range check on the lhs? (Again, yes, it is possible to write correct programs. The question is, does one notation + compilation system have advantages when compared to another? What is the price to pay?)

> *It is not difficult to write a container getting a signed integer as a subscript and have range checking.*

Which is not the point.

Georg