When is a vector not a vector? When it's a container. **Kevlin Henney** examines the use of containers that look like vectors, which can be used to conserve memory when elements in a vector all have the same value

Bound and checked

Kevlin Henney is an independent software development consultant and trainer. He can be reached at **www.curbralan.com**

The Standard Template Library (STL) has settled the question of what a resizable array class template should look like, although its choice of name is a little questionable: vector has few of the properties one would identify with a mathematical vector. Given the obvious naming of a linked list as list, I believe C++ novices and experts alike would have appreciated a more obvious and direct name for a resizable array, such as array or dynarray – the name for the pre-STL array class in the draft standard C++ library.

No matter. We know the name now. More significantly, the STL has brought with it an expectation and definition of what a container interface should look like. The semantics and appearance of subscripting offer one of the simplest examples. It follows, with a couple of notable exceptions, a simple form: the use of a non-negative index on operator[] to return a reference to an element, the const-ness of which reflects the const-ness of the source container. Here is a slightly simplified view of the relevant features of std::vector:

```
namespace std
{
   template<typename value_type>
   class vector
   {
   public:
      value_type &operator[](std::size_t);
      const value_type &operator[](std::size_t) const;
      ...
   private:
      ...
      std::size_t length;
      value_type  *elements;
   };
}
```

The only technical omission from this fragment is the defaulted allocator template parameter. It is a feature of little practical use, whose sole purpose appears to be to obfuscate and mislead. Standard containers also offer a number of typedefs in their public interface. Many of these have little practical use; in the previous code fragment the underlying types are clearer. std::size_t is the standard C type for size quantities, but wrapped in namespace std for more standard C++ digestion.

It is interesting to explore the design decisions that were and were not taken in establishing subscripting semantics. There is little doubt that indexing should start from 0 and be valid up to and including size() - 1, but what happens when the index is out of bounds? What are the quality of failure options?

Undefined behaviour

This is perhaps the simplest option and it means what it means. Out-of-range accesses are not something that should be present in a correct program and therefore their behaviour is not defined. A valid interpretation of this absence of requirement is that no overhead should be expended in checking the correctness. This minimal requirement is opted for by the standard and is potentially the most efficient in execution, the simplest to implement and the most like the native arrays on which they are modelled. The following definition, shown as if it were defined inline within the class definition, would be typical of many standard library implementations:

```
value_type &operator[](std::size_t index)
{
   return elements[index];
}
```

Should you access outside the legal range of constructed objects in data, you are on your own. Your program may crash immediately. Then again, it might limp on for a bit, only to stumble and fall elsewhere... or the problem may never surface as a recognised bug. If you're lucky it will crash immediately.

Debug checking

The next simplest option is to include checks in a debug version. The standard assert macro offers a suitable route, aborting the program on failure:

FACTS AT A GLANCE

- **The standard C++ library generally requires no index checking on containers, preferring efficiency to security.**
- **Other index checking models are possible and can be implemented.**
- **Out-of-range access can also be interpreted to be in range, either by remapping or by growth.**
- **A just-in-time instantiation policy can be used as the basis of a bounded sparse sequence implementation.**

```
value_type &operator[](std::size_t index)
{
    assert(index < length);
    return elements[index];
}
```

Thanks to a little preprocessor magic, such checks can – and should – be omitted in the release version. For those who care, no performance penalty is paid in a release version. For those with different concerns, the correctness constraint is visibly documented in the code as well as in any debug messages generated.

Debug checking is a valid implementation of undefined behaviour and some libraries have an option that enables such checks, such as the STLport library[1]. The availability of such an option is great not just for testing and debugging: it is also useful for those scaling the C++ learning curve – the moral equivalent of stabilisers on a bicycle.

Note that although the indexing type is an unsigned type, the assertion will also catch attempts to use negative subscripts. A negative number converted to an unsigned quantity will become a very large positive number, well beyond the valid subscript range for the vector.

The STL has brought with it an expectation and definition of what a container interface should look like. The semantics and appearance of subscripting offer one of the simplest examples

Runtime enforcement

Although technically a logic error – something that should not occur in a fully tested program – there is the view that bounds should always be checked regardless of the runtime version. In such cases, mapping out-of-bound access to a call to abort – as is the case with assert – is perhaps not the most constructive approach. Throwing an exception is the only meaningful enforcement:

```
value_type &operator[](std::size_t index)
{
    if(index >= length)
        throw out_of_range("operator[] invalid index");
    return elements[index];
}
```

The std::out_of_range exception class is found in the <stdexcept> header and derives from std::logic_error.

Note that throwing an exception is also a viable implementation of undefined behaviour: if no behaviour is defined, any behaviour will do. However, except in teaching, this is not always a reasonable quality of failure to impose on users without informing them first.

The exception-throwing option should not be confused with the debug checking option either. It makes little design sense to have exception throwing as a release-versus-debug option, because exception handling is either part of the fine-grained architecture or it is not. Any halfway house is often a ramshackle, tumbledown affair, rather than a crisp design that makes the best of all options. Exception-safe code will not write itself if you didn't know that exception safety was a requirement[2].

Similarly, a design should be unambiguous and its use obvious. The standard vector attempts to please all of the people all of the time by supporting undefined behaviour for operator[] and bounds-checked exception throwing for the at member. Alas, the appeal of this design is no deeper than veneer. operator[] is the natural choice for subscripting. It requires a conscious effort to use, implying that the programmer is wilfully planning to access out of bounds. If this is the case, it is a simple matter to fix the code before the error is made, which means operator[] can then be used safely. The need for having operator[] and at disappears in a puff of logic and good programming practice.

Index remapping

Where the debug checking and runtime enforcement models remap undefined behaviour to defined but exceptional behaviour, an index remapping takes some of the subscript range outside the 0 to size() - 1 range and resolves it to somewhere valid.

An approach that is both creative and practical – although it breaks with C's array model and C++'s iterator model – is to accept signed indexing, treating negatives as accesses counted down from the end, so that -1 accesses the last element, -2 the second from last and so on.

```
value_type &operator[](std::ptrdiff_t index)
{
    return elements[index + (index > 0 ? 0 : length)];
}
```

This convention is used in the Ruby and Python languages, but perhaps strays a little too far from received wisdom on operator overloading – "when in doubt, do as the ints do"[3] – to be considered an appropriate default for C++ classes. It also addresses only the range -size() to size() - 1.

A less alternative behaviour would be to treat a vector as a circular data structure, supporting wraparound indexing. Indexing off the end just continues again from the beginning:

```
value_type &operator[](std::size_t index)
{
    return elements[index % length];
}
```

The only failure mode this remapping exhibits is when the vector is empty. By default, a division by zero is undefined behaviour, which on some platforms is mapped to a signal. You can select one of the other out-of-bound access models described so far to address this case.

Magic memory

And then there is always "magic memory". If you access past the end of the vector, it automatically grows to that size:

```
value_type &operator[](std::size_t index)
{
    if(index >= length)
        resize(index + 1);
    return elements[index];
}

const value_type &operator[](std::size_t index) const
{
    static const value_type defaulted;
    return index < length ? data[index] : defaulted;
}
```

For this design to work, the value_type used must support default construction.

Some languages and libraries have features that are equivalent to or similar to this just-in-time growth model. For instance,

subscripting a non-existent element in an associative array in Awk will spontaneously cause the creation of that element, and likewise std::map from the STL.

However, there is the legitimate claim that this magical behaviour could be considered subtle and error-prone for a sequential container. A genuinely incorrect index, such as the arbitrary value in an uninitialised variable, could lead to a resize attempt on the order of gigabytes. With any luck this would fail outright. If not, you might hear your machine paging madly as it attempts, diligently and exhaustively, to default construct each and every element in the range. There is also the stark difference between the const and non-const versions of operator[]. That degree of asymmetry may be a little too much to stomach.

If you don't wish to make a big feature out of remapping, a change to the non-const operator[] allows out-of-bound access to be mapped to legal objects rather than to arbitrary locations in memory:

```
value_type &operator[](std::size_t index)
{
   static value_type defaulted;
   return index < length ? data[index] : defaulted;
}
```

This solution provides a scribble space for subscripting, more of a graffiti object than a constant object. The intent is not to provide a useful value –that will change every time the out-of-range value is assigned to – but a legal location that steers the program clear of undefined behaviour. Note that non-const statics should be avoided in multi-threaded environments. Allocating a per-object spare element for use as scribble space would patch this.

Sparse and lazy

Before we throw the baby out with the bath water, there are a couple of bright ideas lurking in the overgrowth model. Consider a large vector in which most of the elements have the same value: how can you work around the need for redundant storage?

Consider a sparse representation, where the common filler value is specified and the differences are stored. The std::map container is an associative container that offers a way of holding a value against a key, without any requirement that keys are continuous. We can use the unsigned index to key elements with values different from the filler. Although we can give such a container an STL-conforming public interface, it cannot be considered a pure drop-in substitute for a vector because it no longer supports constant-time random-access indexing:

```
template<typename value_type>
class sparse_vector
{
public:
   explicit sparse_vector(
      const value_type &filler = value_type())
    : length(0), filler(filler)
   {
   }
   bool empty() const
   {
      return length == 0;
   }
   std::size_t size() const
   {
      return length;
   }
   void clear()
   {
      elements.clear();
      length = 0;
   }
   const value_type &operator[](std::size_t index) const
   {
      map_type::const_iterator found = elements.find(index);
      return found == elements.end() ? filler : *found;
   }
   const value_type &front() const
   {
      return (*this)[0];
   }
   const value_type &back() const
   {
      return (*this)[length - 1];
   }
   ...
private:
   typedef std::map<size_t, value_type> map_type;
   map_type   elements;
   size_t     length;
   value_type filler;
};
```

In the simple case of const subscripting, it is a matter of discovering whether there is a stored corresponding element to return, returning filler if not. The find member function of std::map searches in logarithmic time, which is better than the linear time of the global std::find algorithm but worse than the constant-time random access of std::vector. The virtual length of the sparse_vector is stored explicitly. It could also be used to implement some form of bounds checking, as explored earlier in this article.

Note that throwing an exception is also a viable implementation of undefined behaviour: if no behaviour is defined, any behaviour will do

What about non-const subscripting? If you access an element that is being stored explicitly, then no problem, but accessing something that isn't there calls for something like spontaneous creation:

```
template<typename value_type>
class sparse_vector
{
public:
   ...
   value_type &operator[](size_t index)
   {
      return elements.insert(
         std::make_pair(index, filler)).first->second;
   }
   value_type &front()
   {
      return (*this)[0];
```

```
   }
   value_type &back()
   {
      return (*this)[length - 1];
   }
   void push_back(const value_type &new_back)
   {
      if(new_back != filler)
         elements.insert(std::make_pair(length, new_back));
      ++length;
   }
   void pop_back()
   {
      elements.erase(length - 1);
      --length;
   }
   ...
};
```

The magical incantation in operator[] needs a little deciphering:

- std::map works in terms of std::pair objects, holding the key and mapped value together as a pair.
- The insert function takes a std::pair, which can be composed from the intended index and the filler using std::make_pair, and returns a result that holds an iterator to the corresponding inserted location.
- But it doesn't just return an iterator: it returns a std::pair with the iterator as the first value and a bool as the second value. The second value is true if the new mapping was inserted and false if the key already existed.
- However, we don't care if a new mapping was inserted with filler as its value or if the key was already there, so we just access the first element of the resulting std::pair, the iterator.
- In using the iterator, we need to remember that the iterator for a std::map dereferences to a std::pair holding the key, first, and mapped value, second. We are interested only in the looked-up value, so we access second.

A sparse_vector that isn't sparse

The end accessors, front and back, fall into place more easily once you have operator[] sorted. The erase function used in pop_back will only erase an element at a key if it exists, doing nothing otherwise. And, true to the lazy nature of the sparse_vector data structure, push_back performs a physical insertion only if the value to be appended is different from the filler value. This adds the requirement that the value_type parameter should be equality comparable. From the primitives of pop_back and push_back you can construct a suitable resize member function, should you wish to make sparse_vector more std::vector-like.

You may notice that the sparse_vector is not perfectly sparse. It cannot be 100% effective in optimising away storage for elements that have the same value as filler, because not all uses of the non-const operator[] are for assignment, and assignment to elements cannot be intercepted easily:

```
sparse_vector<int> data(-1);
data.push_back(-1);
data.push_back(-1);
...
data[0] = -1;
std::cout << data[1] << std::endl;
```

After the assignment, the initial element is stored explicitly but has the same value as filler, because an lvalue to a real object was required on the left-hand side of the assignment. Similarly, the next - element is stored explicitly, despite never being the target of assignment. The call to operator[] is made independently of how its result is to be used, even if a non-const result is used in a const fashion.

We have little direct control over how std::vector manages its memory and the same can be said of sparse_vector. It is possible to extend the implementation of sparse_vector to intermittently rout out values being stored surplus to requirements. However, this would complicate the implementation and raise questions about the evenness and efficiency of sparse_vector's performance. Taking a cue from std::vector's own storage management mechanisms, we can institute a simple manual interface for sparse_vector users to use, for those who truly care about minimising storage:

```
template<typename value_type>
class sparse_vector
{
public:
   ...
   size_t capacity() const
   {
      return elements.size();
   }
   void conserve()
   {
      map_type::iterator at = elements.begin();
      while(at != elements.end())
         if(at->second == filler)
            elements.erase(at++);
         else
            ++at;
   }
   ...
};
```

Using composition

Where std::vector's capacity function indicates, in conjunction with size, how much spare storage is left before reallocation, sparse_vector's capacity function indicates how much capacity is actually used in storing elements. Where std::vector's reserve function puts aside extra space in advance of usage, spare_vector's conserve function reclaims redundant element storage from usage. The gotcha to avoid in conserve's while loop – and the reason why it is not a fully fledged for loop – is that erasing the element referred to by an iterator invalidates that iterator, which means it cannot be used for any purpose. For std::map it is a simple matter of using a postfix increment to ensure that the iterator is moved on somewhere valid.

The power of libraries lies in composition. It is what you can build with a component that makes a component useful, not its public billing. Part of this is to understand which design alternatives were available, which were taken and what to do if you don't like them. In the case of the STL, it proves easy to layer a sparse sequence container implementation on top of an associative container to support lazy indexing. ■

References

1. STLport, www.stlport.com
2. Kevlin Henney, "Making an Exception", *Application Development Advisor*, 5(4)
3. Scott Meyers, *More Effective C++*, Addison-Wesley, 1996