```
namespace accu
{
  template<typename type>
  bool operator==( const pointer<type> &lhs, const type *rhs ) throw()
  { return ::raw(lhs) == rhs; }

  template<typename type>
  bool operator==( const type *lhs, const pointer<type> &rhs ) throw()
  { return rhs == lhs; }

  template<typename type>
  bool operator!=( const pointer<type> &lhs, const type *rhs ) throw()
  { return !(lhs == rhs); }

  template<typename type>
  bool operator!=( const type *lhs, const pointer<type> &rhs ) throw()
  { return !(rhs == lhs); }
}
```

That's almost it for now. I'll just leave you with one final thought.

What you don't implement (eg ++ in pointer) can be as important as what you do.

1. [1] Scientific and Engineering C++, John J.Barton & Lee R.Nackman, Addison Wesley, ISBN 0-201-5393-6, Chapter 14 Pointer Classes, page 419
2. [2] C++ Draft Standard, CD2, 12.8 Copying class objects, Footnotes 104 107

Jon Jagger
jjagger@qatraining.com

STL Algorithms: Finding
By Francis Glassborow

The issue before last I wrote a brief survey of the resources the STL provides to support your need to sort a container of objects. Other things got in my way so I missed an article for the last issue. But I had not forgotten. I think that mastery of the STL as such and the underlying philosophy is important. Though a relative late comer in the process of standardising C++ I think that it is one of the most significant developments in the language. I would go so far as to state that anyone who has not mastered the STL has no right to either present courses on C++ nor to write books about it. I know that I have been highly critical of C++ in the past and probably will be again but the STL together with exception handling and namespaces are three vital elements that make modern C++ something special. The class concept did much to move us on from the procedural style of programming that characterises good C. The concept of component genericity, good encapsulation and proper management of problems take us from the classic C++ of the 1980's to what should be the C++ of the late 90's. Unfortunately it will be well into the next millennium before the majority understand this.

The much sought after silver bullet actually has nothing to do with a new programming language and little to do with a change in methodology. What is needed is for programmers to understand their tools and use them properly. With the level of instruction currently on offer that is a hopeless case.

OK, end of rant and on with the topic that logically follows sorting: searching.

C offered just one library mechanism for searching: ***bsearch*()**. You might guess that ***bsearch*** applies a binary search. There is nowhere in the ISO C Standard that places any such burden on the implementor. The

only limitation is that the elements of the array that match the required criterion shall come after all those that compare less than and before all those that compare greater than. In other words the array shall have been so sorted that a binary search would work. Of course most implementors will use a binary search because that would seem the obvious solution to the problem but their choice is a pure quality of implementation issue.

The STL algorithms place a far greater requirement on their implementors. In addition there are more options open to you.

Sequences & Containers

You need to recognise that among containers there is a sub-group of sequences. A sequence is a special form of container wherein it makes sense to speak of one element coming before another. Arrays, vectors, queues, lists are all examples of sequences. There are also containers like bags and sets where there is no ordering. In between we have things like maps where there may be an ordering but there does not have to be. Make sure that you understand that. The idea is not the same as the idea of being sorted. A non-sequence container cannot be sorted because the concept of order is alien to the concept (of course the underlying data-structure used to implement a bag will have some linear ordering in storage but that is a low-level implementation detail that has nothing to do with the concept of a bag.) It is implicit in a sequence container that it can be sorted, but it does not have to be. When we look at the algorithms we will need to ask ourselves if they require that the container is a) a sequence and b) sorted (by a criterion related to the search criterion). There is no point in looking for the first of something if it is not in a sequence, but it is perfectly reasonable to ask if an element is in a container even if it is a non-sequence container type.

So much for the general concept. However trying to handle the general concept of a non-sequence container would be rather daunting so the STL generally assumes that whether they are sorted or not, containers will be sequences. Even our sets and bags will allow us to iterate over all elements, addressing each once. So the important distinction will not be whether an STL container is a sequence but whether it has been ordered.

Unordered Sequences

Note that an unordered sequence does not mean that its order is meaningless. On the contrary, the order may be extremely important and not to be disturbed. If you doubt this consider what would happen if I re-ordered the sequence of symbols that make up this paragraph.

Of course these include inappropriately ordered ones. Basically you want to ask one of the following questions about such a sequence. Remember that writing code to answer a specific question for a specified container type can be trivial but the purpose of placing these operations into the STL is to ensure that we can change the container type and still have our code work.

[In the following ***vec*** is an instance of ***vector<int>***. and that ***ip*** is a ***vector<int>::iterator***. Unless stated otherwise the first two parameters of the STL find family of functions are iterators delineating the range of element to be checked. I have also omitted the ***std::*** prefix. In practice you would be well advised to retain this prefix because the names of many functions in the STL algorithms are obvious and so likely to have been used elsewhere in code.]

Where is the next instance of something?

To do this you use ***find*()**. The first two parameters are iterators that identify the sequence to be searched. (Remember that the STL always uses the rule of giving the iterator of the first element and the iterator for one after the last one.) The next parameter gives the value to be found. Note that this is a value (though it is a reference parameter) and so relies on there being a definition of ***operator*==()** available for the type of the elements of the sequence.

Example:
```
ip=find(vec.begin(), vec.end(), 7);
```

ip will be set to the first instance of 7 in the vector. If there isn't a 7 in ***vec*** then the iterator of the end boundary (***vec.end*()** in the example is returned – there isn't a general null-iterator so we have to make do and trust the programmer to check)

Where is the first instance of something that matches a specific rule that is provided as a predicate?

To do this you use ***find_if*()**. The first two parameters give the sequence to be searched. The third parameter is a predicate, that is either a function or a function object (instance of a class that includes an overload for ***operator*()**.) The predicate must take a single parameter of a type appropriate for the sequence and return a ***bool***.

Example:
```
ip = find_if(
        vec.begin(),
        vec.end(),
        bind2nd(less<int>, 12));
```

I will deal with the tools for creating predicates in another column. The above example uses two items from the STL to create a predicate that returns true if the element is less than 12. The result is that ****ip*** will be the first element of ***vec*** that is less than 12.

How many instances of a value occur in the container?

Simple; use ***count*()** and give it the required value as its third parameter.

Example:
```
int i = count(vec.begin(),vec.end(), 7);
```

will count the instances of 7 in ***vec*** and store the answer in ***instances***.

How many elements satisfying a specific rule occur in the container?

It is the purpose of ***count_if*()** to answer this question. Its third parameter will be a predicate that provides the rule.

Example:
```
int instances = count(
    vec.begin(),
    vec.end(),
    bind2nd(not_equal_to<int>, 41));
```

Will give you the number of elements of ***vec*** that are not equal to 41.

Where is the first instance of an element of one sequence in another?

The answer is provided by ***find_first_of*()** which takes four parameters. The first two are consistent with our convention in that they provide the sequence to be checked for a value. The remaining parameters are iterators that delineate the list of acceptable values. This is a very useful algorithm because it allows me to provide a 'list' of things any one of which will satisfy my requirement.

Where is the first instance of a consecutive pair of values in the container?

This question is answered by ***adjacent_find*()**. The third parameter is a predicate that takes two arguments of the type in the sequence.

Example:
```
ip = adjacent_find(
          vec.begin(),
          vec.end(),
          greater<int>());
```

will result in ***ip*** iterating the first element of ***vec*** that is greater than the next one.

Sub-sequences

As well as being able to search for and count elements that either match a given value or conform to a given rule, we can also consider the relationships between pairs of sequences. Sensibly we can check if two sequences match (contain the same values in the same order) with ***equal()*** (four parameters delineating the two sequences.)

If two sequences are not equal it makes sense to ask where is the first element that does not match. The answer to this is provided by ***mismatch()***.

We could also want to check a sequence to see if it contains a specified sub-sequence.

The STL provides us with two functions for this purpose. ***search()*** (with four iterator type parameters) returns the iterator of the first element of the first instance of the second sequence as a sub-sequence of the first one. ***find_end()*** returns an iterator to the last matching sub-sequence. The choice of function name for these algorithms leaves much to be desired. When the experts have spent hours thrashing out the details they have little time or energy left to work on name consistency. Sad, but we should be thankful for all they did rather than moaning about the way they fell short of perfection.

There is a third function concerning sub-sequences and that is ***search_n()***. It is far from obvious what this function does and I had to spend quite a time studying it before I understood it (I hope). What this function does is to search for consecutive repetitions within a sequence. The required number is given in the third argument of the function call. So:

```
ip= search_n(vec.begin(),vec.end(),5,3);
```

would set ***ip*** to the first element of ***vec*** that is the first of five consecutive threes.

The basic versions of each of the sub-sequence functions assumes that the comparison will be done strictly in terms of equality. However if you want to provide some other rule to determine what you mean by matching elements then you can provide it as a final extra argument. So if ***vec1*** is another ***vector*** of ***int*** then:

```
equal(
  vec.begin(),
  vec.end(),
  vec1.begin(),
  vec1.end(),
  less<int>())
```

returns ***true*** is every element of ***vec*** is less than the corresponding element of ***vec1*** and the two vectors are the same length, otherwise it returns ***false***. It might have been wiser to have called this function match but that is history.

Curiously there is no function that counts the number of instances of a sub-sequence within a sequence. You must also be careful when searching for a sub-sequence that is composed of consecutive identical sub-sequences (within the terms of what constitutes a match). It may be clear that searching for an exact match with the sequence ***1,2,1,2*** leaves the question of what to do with it finding ***1,2,1,2,1,2,1,2*** (two instances or three?) but when you start providing a rule via the extra parameter the potential for the unexpected increases.

Sorted Sequences

Searches on unsorted sequences (or sequences sorted by an inappropriate criterion) are inefficient because little can be done to improve on a straight linear search (there are a few improvements which have been developed for text searches but they are basically linear improvements). If you want to check that there are no instances of 64 in vector of a million ***int***s then that will take a thousand times as long as making the same check on a vector of a thousand ***int***s. Of course if what you are looking for is near the start of a container you will get a quick answer, but if not you will have to wait (or invest in a large array processor).

When you have an appropriately sorted sequence you have an opportunity to apply a binary search. That is a vast improvement as it works in a time proportional to the number of bits needed to represent the number of elements being searched. This means searching through a million items at worst takes about twice as long as searching a thousand.

The STL function you need is ***binary_search*()**. It takes either three or four arguments (the usual first two, followed by the value required and an optional predicate to define match).

When you have a sorted sequence you might be interested in a sub-sequence that meets certain boundary conditions. There are three functions that support this requirement.

lower_bound*()** returns an iterator to the first element that meets the requirement specified by its final argument(s). This function is an optimised version of ***find() (or ***find_if*()**) that

takes into account that the sequence is ordered.

***upper_bound*()** returns an iterator to the first element that fails to meet the requirement after one that has.

equal_range*()** returns a ***pair of iterators (***pair*** is an STL component) that delineate the sub-sequence that meets the requirement specified by the final argument(s).

Conclusion

The above is a rather skimpy survey of the features of STL that support the requirement to find or count elements that meet specific constraints. Rather than spend your time writing your own functions for such purposes you would be better to study those that are relevant to your needs as and when those occur. That way you will produce more maintainable code that will gain from the expertise that has been applied in producing implementations of STL. Of course you will need a good STL implementation to get the best advantage but even a poor one is likely to be better than the handcrafted code of all but the most expert.

Francis Glassborow
Francis@robinton.demon.co.uk

Whiteboard

Rational Value Comments By Graham Jones

I have some comments on the Harpist's 'Rational Values' articles. He asked for an algorithm to convert floating point numbers to fractional form. There is a good algorithm based on continued fractions which does this:

Given a real number z>0, define p[0]=0, q[0]=1, p[1]=1, q[1]=0, and x[1]=z.

Then for n>=2, recursively define a[n]=(int)x[n-1], p[n]=a[n]*p[n-1]+p[n-2], q[n]=a[n]*q[n-1]+q[n-2] and x[n]=1/(x[n-1]-a[n]). The sequence p[2]/q[2], p[3]/q[3], ... gives the best rational approximations to z. If z is rational, a[n] will equal x[n-1] for some n and z=p[n]/q[n].

However, I am very dubious about the usefulness of this. In fact it is clear that the Harpist and I have very different ideas about what a Rational class should look like. My idea of a Rational class is based on something that might actually be useful, and the question I asked myself was: why should anyone use it in preference to floating point numbers? The only possible advantage I can see is that calculations with rationals are exact. If you convert floating point numbers to rationals, this advantage is lost. I can see no point in providing such a conversion, and providing it as a constructor seems positively harmful, more or less guaranteeing that users will misuse the class, accidentally or otherwise.

I should also point out that while the continued fractions method provides a good approximation to π and many other values, the best approximation to 0.000007 is 0 using 16-bit numerator and denominator. If the numerator and denominator have limited ranges, rationals are not good for general purpose arithmetic.

For this and other reasons it seems to me that multi-length integer arithmetic would be essential for a useful Rational class. Once this is done, the conversion from floating point can be done exactly, by a completely different method - frexp() and modf() from math.h point the way. (Even then, I think that providing the conversion as a constructor is a bad idea.) The Harpist talks about multi-length integers as something that could be added later, but much of the code would have to be rewritten and it would seem better to me to start off by writing a BigInteger class.

Graham Jones