Kevlin Henney explores the relevance of his sparse_vector class template to simplifying selection statements using lookup tables, and then takes things further by encapsulating an efficient flat-file structure for lookups

Look me up sometime

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

MY LAST ARTICLE[1] LOOKED AT CONTAINER indexing, arriving eventually at a class template, sparse_vector. This had a similar functional interface to a standard vector but a representation that was node-based rather than array-based. Rather than storing all elements in a sequence consecutively, a sparse_vector aims to store only elements that are not equal to a given default filler value.

What use could such a container be? Saving memory is one obvious application. If you have an indexable container with a large number of elements – the majority with the same value – some systems will benefit from a sparse representation that avoids holding redundant data instead of a potentially memory-hungry contiguous representation.

Another use is for simplifying logic, in particular the replacing of rambling and clumsy selection statements with a lookup-based approach. The following fragment is based on some code I saw recently:

```
std::string message;
switch(error)
{
case 0:
   message = "No error";
   break;
case 1:
   message = "Initialisation failed";
   break;
case 2:
   message = "Connection failed";
   break;
case 4:
   message = "Connection dropped";
   break;
case 12:
   message = "Unknown format";
   break;
case 27:
   message = "Memory exhaustion";
   break;
case 67:
   message = "Internal error";
   break;
default:
   message = "Unknown error";
   break;
}
```

Elegant? No. Wretched case-and-paste code seems to litter many systems, fragmenting program flow into a list of hardwired exceptional cases instead of… well, something that flows, clearly and smoothly. The antidote to most rambling switch statements or cascading if else if statements is to adopt a table-driven approach. This is more declarative, and as simple and practical as it is elegant. The only problem in this case is that the indices are not contiguous. If we were looking up the number of days in each month, a simple const array would suffice. But for a 16-bit error code whose bandwidth is mostly unused, a single array with 65,336 elements, most of which were initialised to "Unknown error", could be considered inept rather than adept programming.

Using the map container

This is where the standard map container can help out – hold the significant messages against their error numbers. This leads to a simple initialisation that can be kept separate from the error processing code:

```
std::map<error_code, std::string> messages;
messages[0]  = "No error";
messages[1]  = "Initialisation failed";
messages[2]  = "Connection failed";
messages[4]  = "Connection dropped";
messages[12] = "Unknown format";
messages[27] = "Memory exhaustion";
messages[67] = "Internal error";
```

The error processing code becomes a little more direct, but is still hampered a little by the catch-all for unknown error codes and some moderately serious syntax:

```
std::map<error_code, std::string>::iterator found =
   messages.find(error);
```

FACTS AT A GLANCE

- **Standard associative containers are node-based, sorted hierarchies of linked elements.**
- **Random access sequences can be used as associative containers when they are sorted.**
- **New container types can be written that offer different properties from the standard containers, but are built from and encapsulate them.**
- **The flat_set and flat_multiset class templates are useful non-standard commodity types.**

```
std::string message =
   found != messages.end() ? *found : "Unknown error";
```

This is where a sparse_vector can simplify things:

```
sparse_vector<std::string> messages("Unknown error");
messages.resize(65536);
messages[0]  = "No error";
messages[1]  = "Initialisation failed";
messages[2]  = "Connection failed";
messages[4]  = "Connection dropped";
messages[12] = "Unknown format";
messages[27] = "Memory exhaustion";
messages[67] = "Internal error";
```

Which reduces the original logic to the highly readable:

```
std::string message = messages[error];
```

Internally, a sparse_vector is based on a map, but all the special-case logic is encapsulated within it rather than polluting the application code. This highlights the fact that encapsulation is not simply about making data private and is certainly not about cluttering a class interface with dubious *setters* and *getters*. It is more genuinely concerned with reducing the number of hoops a class user has to jump through to get a job done.

Sorted

What about the other way around? How about a using a vector for content-based rather than position-based lookup? Why might such a need arise? Consider a large set of strings that is *read-mostly*. Insertions and removals are not common or are normally grouped together, and lookup is the most frequent action. For instance, a simple spelling dictionary is a set that requires fast lookup but does not require constant updating. This sounds like a job for std::set, which holds its elements in a sorted tree of individually allocated nodes. Both lookup and insertion take logarithmic time with respect to the number of elements in the set. For a dictionary, the majority of insertions take place during a single phase of the program:

```
std::ifstream in("words.txt");
std::istream_iterator<std::string> begin(in), end;
std::set<std::string> words(begin, end);
...
std::string word;
...
if(!words.count(word))
   ...
```

This model of usage means that in exchange for a lot of dynamic memory activity, most of the benefits of a hierarchically linked structure are never exercised. A common alternative is to flatten out the data structure and operate on it manually: populate a random access sequence, such as std::vector or std::deque, with elements, and use standard algorithmic function templates to sort and search it:

```
std::ifstream in("words.txt");
std::istream_iterator<std::string> begin(in), end;
std::vector<std::string> words(begin, end);
std::sort(words.begin(), words.end());
words.erase(
   std::unique(words.begin(), words.end()),
   words.end());
...
std::string word;
...
if(!std::binary_search(words.begin(), words.end(), word))
   ...
```

This is a little tedious because it exposes the workings of the data structure to the whole program. The absence of encapsulation means that logic will inevitably be duplicated and there is no simple way to safeguard the invariant – to be a sorted sequence of unique entries – of the data structure. Sorting is easy enough, but eliminating duplicates requires the user to remember that the iterator range is merely reorganised and nothing is actually removed from the container until the container is told explicitly (this is the same idiom that is used with std::remove[2]).

Flatland

A flattened data structure is useful, but it is crying out to be encapsulated. A flat_set class template that supports the same function interface as std::set, but with slightly different space and speed trade-offs, is the design goal. The easiest way to score this is first to solve a slightly simpler problem – that of flat_multiset. Like std::multiset it is sorted, but may contain duplicate elements.

As well as offering a useful facility, the implementation of flat_multiset is a useful exercise in working with the standard algorithms and the received wisdom on their effective use[3]. I won't describe or define all the operations for flat_multiset or flat_set – you can find guidance on what they should be and what is required of them from the standard or any good STL reference[4,5].

Here is a quick sketch of the outside and inside of flat_multiset:

```
template<
   typename value_type,
   typename key_compare = std::less<value_type> >
class flat_multiset
{
   ...
private:
   ...
   std::vector<value_type> elements;
   key_compare comparer;
};
```

We know the internal representation is going to be based on a random access sequence and std::vector fulfils this need. The standard associative containers all take a defaulted parameter that defines their internal ordering. The default function object type is std::less, which gives an ascending order, and an instance is stored for use by the member functions.

Simple construction and assignment require no extra effort to preserve a sorted order:

```
template<...>
class flat_multiset
{
public:
   explicit flat_multiset(
      const key_compare &comparer = key_compare())
    : comparer(comparer)
   {
   }
   flat_multiset(
      const flat_multiset &other,
      const key_compare &comparer = key_compare())
    : elements(other.elements), comparer(comparer)
   {
   }
   flat_multiset &operator=(const flat_multiset &rhs)
```

```
    {
      elements = rhs.elements;
      comparer = rhs.comparer;
      return *this;
    }
    ...
};
```

However, constructing a new flat_multiset from an iterator range, such as input iterators from a file or inserting a range of elements into an existing one, requires explicit sorting:

```
template<...>
class flat_multiset
{
public:
    ...
    template<typename input_iterator>
    flat_multiset(
      input_iterator begin, input_iterator end,
      const key_compare &comparer = key_compare())
     : elements(begin, end), comparer(comparer)
    {
      sort();
    }
    template<typename input_iterator>
    void insert(input_iterator begin, input_iterator end)
    {
      elements.insert(elements.end(), begin, end);
      sort();
    }
    ...
private:
    void sort()
    {
      std::sort(
        elements.begin(), elements.end(), comparer);
    }
    ...
};
```

The private sort member function has been factored out to make the behaviour clearer, wrapping up the use of the standard sort and the comparison object. In the case of the insert function, the first step is to append the new elements following the end of the existing sequence and then resort the whole lot. There are other approaches that can be more efficient, but this one is the briefest.

Many of the smaller operations are trivial to implement and just forward to the corresponding member of the contained std::vector:

```
template<...>
class flat_multiset
{
public:
    ...
    bool empty() const
    {
      return elements.empty();
    }
    void clear()
    {
       elements.clear();
    }
    ...
};
```

This simple delegation also applies to many of the typedefs that the flat_multiset should export. Conveniently enough, this also holds true for iteration. Many programmers find providing their own iterator types a little daunting, although when taken one step a time, the task often turns out to be comparatively simple[6]. It doesn't get much simpler than for flat_multiset:

```
template<...>
class flat_multiset
{
public:
    ...
    typedef std::vector<value_type>::const_iterator iterator;
    typedef iterator const_iterator;
    iterator begin() const
    {
      return elements.begin();
    }
    iterator end() const
    {
      return elements.end();
    }
    ...
};
```

The contained vector already has iterator types that have all the right properties; flat_multiset need only reexport them. There is one caveat, however: iterator access to the set must be const-only, which is why both iterator and const_iterator are equivalent to a vector's const_iterator. A non-const iterator would allow elements in a sequence to be modified, which could break the guarantee of a sorted order.

Searching for a value is simple enough when you realise the most important member function is equal_range, which returns a pair of iterators, delimiting the upper and lower bound of where the value occurs or would occur in the sorted range:

```
template<...>
class flat_multiset
{
public:
    ...
    std::pair<iterator, iterator> equal_range(
      const value_type &to_find) const
    {
      return std::equal_range(
        elements.begin(), elements.end(),
        to_find, comparer);
    }
    iterator find(const value_type &to_find) const
    {
      std::pair<iterator, iterator> found =
        equal_range(to_find);
      return found.first != found.second ?
        found.first : elements.end();
    }
    std::size_t count(const value_type &to_find) const
    {
      std::pair<iterator, iterator> found =
        equal_range(to_find);
      return found.second - found.first;
    }
    ...
};
```

Iterator (pointer) arithmetic makes counting the number of times a value occurs quite trivial.

So far, so good. But now things become a little hairier. We need to consider how to insert and erase a single value efficiently. A simplistic approach to inserting a single element would be to use push_back to append it to the range and then resort the whole lot. A more efficient approach is to find the location that the element should be placed in and shuffle up the elements that follow. Erasing a value presents a minor challenge because it requires the use of modifiable iterators to identify and remove the subrange in which a value occurs. But these are only minor challenges:

```
template<...>
class flat_multiset
{
public:
   ...
   iterator insert(const value_type &new_value)
   {
      return elements.insert(
         std::upper_bound(
            elements.begin(), elements.end(),
            new_value, comparer),
         new_value);
   }
   std::size_t erase(const value_type &to_erase)
   {
      std::pair<muterator, muterator> found =
         std::equal_range(
            elements.begin(), elements.end(),
            to_erase, comparer);
      std::size_t result = found.second - found.first;
      elements.erase(found.first, found.second);
      return result;
   }
   ...
private:
   typedef std::vector<value_type>::iterator muterator;
   ...
};
```

The muterator (mutable iterator) typedef is just there to simplify the code in erase and other member functions that require such a capability.

Regardless of language, all such algorithmic work is twiddly and the major challenge sometimes seems to be to read LISP! For programmers without any experience in functional programming, the bracketing density appears a little high when using the STL.

That's the hard work done. The standard tells you which functions are expected of associative containers, the standard library provides the relevant underlying mechanism and all the resulting member functions are short, although sometimes a little fiddly.

So, other than the names, what is the minimum change needed to turn a flat_multiset into a flat_set? Of the functions shown, only two require changes – the private sort function, which must now also eliminate duplicates, and the single value insert function, which must now ensure that duplicates are not introduced.

The sort function sounds innocent enough. Looking back through the code in the article, you will find a use of unique that looks like it might solve the problem, leading to the following:

```
template<...>
class flat_set
{
   ...
private:
   void sort()
   {
      std::sort(
         elements.begin(), elements.end(), comparer);
      elements.erase(
         std::unique(elements.begin(), elements.end()),
         elements.end());
   }
   ...
};
```

This solution is simple, tempting and, alas, wrong. The out-of-the-box version of unique uses straightforward equality to cut down the range. This may or may not match up with the templated sorting criteria that have been used with the container. In the case of flat_set<std::string> it will do the right thing because the < and == operators will be used and are consistent with one another. However, if a case-insensitive ordering has been provided as a parameter to flat_set, equality checking will not eliminate duplicates with the same spelling but different case.

It transpires that associative containers with unique elements do not use equality to discard duplicates. They use *equivalence*, a subtly different concept, which sometimes gives the same result and sometimes doesn't. If we consider the ordering of an associative container to be *less-than*, then equivalence of two elements is when neither element is not *less-than* the other. To make sense of this construct we can define a helper function object type to perform this logic for us, using it with the overloaded form of unique that permits you to pass your own filter function:

```
template<...>
class flat_set
{
   ...
private:
   class equivalent
   {
   public:
      explicit equivalent(
         const key_compare &comparer)
       : comparer(comparer)
      {
      }
      bool operator()(
         const value_type &lhs,
         const value_type &rhs) const
      {
         return !comparer(lhs, rhs) &&
               !comparer(rhs, lhs);
      }
   private:
      key_compare comparer;
   };
   void sort()
   {
      std::sort(
         elements.begin(), elements.end(), comparer);
      elements.erase(
         std::unique(
            elements.begin(), elements.end(),
            equivalent(comparer)),
         elements.end());
```

```
    }
    ...
};
```

The last piece of the jigsaw is insertion of a single element. This cannot be the same as the one for flat_multiset, or even a slightly tweaked version of it, because its signature differs. The return value for single-element insertion into a unique associative container must indicate whether or not a new value was inserted – a new element will not be inserted if the value already exists – and an iterator to either the newly inserted value or the existing value:

```
template<...>
class flat_set
{
public:
   std::pair<iterator, bool> insert(
      const value_type &new_value)
   {
      std::pair<muterator, muterator> found =
         std::equal_range(
            elements.begin(), elements.end(),
            new_value, comparer);
      bool exists = found.first != found.second;
      iterator result = exists ?
         found.first :
         elements.insert(found.first, new_value);
      return std::make_pair(result, !exists);
   }
   ...
};
```

In creating a family of types, even one as small as flat_set and flat_multiset, there are design considerations in addition to performance and conformance to STL expectations. Copy and paste would be an inappropriate way to share the common logic between the two types, as would public inheritance. Private inheritance of flat_multiset is a possibility but not a particularly fetching one. Factoring out a common but private base to both flat_set and flat_multiset would be a cleaner and more decoupled approach. Alternatively, delegation can be used, either one to another or, better still, both forwarding to a third implementation type.

The STL is an open framework based on requirements and not simply a fixed collection of code – some containers and algorithms, plus hangers on. It is an extensible model that allows you to customise and introduce your own abstractions, which may be, as in the case of flat_set and flat_multiset, hybrids of existing concepts and code – the interface of existing types with the implementation of another type. The flat associative containers have the benefit of space economy and, for read-mostly use, speed. Balancing these trade-offs is the most common reason for defining your own commodity types. ■

References

1. Kevlin Henney, "Bound and checked", *Application Development Advisor* 6(1), www.appdevadvisor.co.uk
2. Kevlin Henney, "If I had a hammer", *Application Development Advisor*, 5(6), www.appdevadvisor.co.uk
3. Scott Meyers, *Effective STL*, Addison-Wesley, 2001
4. Matthew Austern, *Generic Programming and the STL*, Addison-Wesley, 1999
5. Nicolai Josuttis, *The C++ Standard Library*, Addison-Wesley, 1999
6. Kevlin Henney, "Promoting polymorphism", *Application Development Advisor*, 5(8), www.appdevadvisor.co.uk

ADA's free e-mail newsletter for software developers

By now many of you will have had a free on-line newsletter which is a service to readers of ADA and packed with the very latest editorial for software and application developers. However, some of our readers don't get this free service.

If you don't get your free newsletter it's for one of two good reasons.

1. Even though you subscribed to ADA, you haven't supplied us with your e-mail address.
2. You ticked the privacy box at the time you subscribed and we can't now send you our newsletter.

Sign up for this free service at www.appdevadvisor.co.uk/subscribe.htm

www.appdevadvisor.co.uk/subscribe.htm

SIGS Application Development ADVISOR