Elements of Modern C++

Lecture 9: Various Libraries and Techniques

Thorsten Jørgen Ottosen
nesotto@cs.aau.dk

Department of Computer Science
Aalborg University

December 2, 2008

Quote of The Lecture

Davids Johnson, Assen, The Netherlands

Get your data structures correct first, and the rest of the program will write itself.

Let us look at some other libraries in Boost that we have not seen before.

Boost.Circular Buffer

An efficient way to get a queue:

```
boost::circular_buffer<int> cb(3);

// Insert some elements into the buffer.
cb.push_back(1);
cb.push_back(2);
cb.push_back(3);
assert( cb.full() );

int a = cb[0];  // a == 1
int b = cb[1];  // b == 2
int c = cb[2];  // c == 3

// The buffer is full now, pushing subsequent
// elements will overwrite the front-most elements.

cb.push_back(4);  // Overwrite 1 with 4.
cb.push_back(5);  // Overwrite 2 with 5.
```

Otherwise it has the usual random-access container interface.

Boost.Circular Buffer (2)

We also have some new functionality:

```
typedef std::pair<pointer, size_type> array_range;
array_range array_one();
array_range array_two();

pointer linearize();
bool is_linearized() const;

void rotate(const_iterator new_begin);
```

Notice that `rotate()` takes $O(1)$ time by simply adjusting its iterators. This can be very handy.

Boost.Bimap

When working with e.g. `map<Key,Value>`, we often need a fast way to find

- a value given a key, and/or
- a value without the key, and/or
- a key given a value.

In such situations we can either

1. Do an $O(n)$ time search through the values.
2. Have two containers: `map<Key,Value>` and `map<Value,Key>` to facility $O(\log n)$ or $O(1)$ lookup (depending on using an ordered or unordered map).

Linear time lookup is often too slow: it can transform an outer loop into the inefficient $O(n^2)$ instead of the efficient $O(n \log n)$ or better.

Furthermore, there are potential problems with *exception-safety* because we might need to keep the two containers synchronized.

Boost.Bimap (2)

Boost.Bimap is an *elegant* solution to these problems:

```
#include <boost/bimap.hpp>
typedef boost::bimap< int, std::string > bm_type;
bm_type bm;

bm.insert( bm_type::value_type(1, "one" ) );
bm.insert( bm_type::value_type(2, "two" ) );

std::cout « "There are " « bm.size()
         « "relations" « std::endl;

for( bm_type::const_iterator iter = bm.begin(),
                             iend = bm.end();
     iter != iend; ++iter )
{
    std::cout « iter->left « " <-> "
              « iter->right « std::endl;
}
```

So this works *almost* like e.g. `std::map<int,std::string>`. The difference being `left/right` vs. `first/second`.

Boost.Bimap (3)

The class defines **map views**:

```
typedef bm_type::left_map::const_iterator
                             left_const_iterator;

for( left_const_iterator left_iter = bm.left.begin(),
                              iend = bm.left.end();
     left_iter != iend; ++left_iter )
{
    std::cout « left_iter->first « " -> "
              « left_iter->second « std::endl;
}

bm_type::left_const_iterator left_iter = bm.left.find(2);
assert( left_iter->second ==  "two" );

bm.left.insert( bm_type::left_value_type( 3, "three" ) );
```

The left view is like an `std::map<int,std::string>`. So it is *efficient* for looking up the integers.

Boost.Bimap (4)

Conversely, the right map view works as `std::map<std::string,int>`. It is therefore efficient for looking up the strings.

```
bm_type::right_const_iterator right_iter =
                                        bm.right.find("two");

assert( right_iter->second ==  2 );
assert( bm.right.at("one") == 1 );

bm.right.erase("two");
bm.right.insert( bm_type::right_value_type( "four", 4 ) );
```

Notice how we do not insert `(4,"four")` in this view.

Boost.Bimap (5)

Notice how all pairs of values must be *immutable* in the map:

```
bm.left.find(1)->second = "1"; // Compilation error
```

Furthermore, both keys must be unique:

```
bm.clear();

bm.insert( bm_type::value_type( 1, "one" ) );

bm.insert( bm_type::value_type( 1, "1"   ) );
bm.insert( bm_type::value_type( 2, "one" ) );

assert( bm.size() == 1 );
```

This is a natural consequence of having two unique indexes.

Boost.Bimap (6)

It turns out that bimaps can be configured at *compile-time*:

```
  typedef bimap< int, std::string > bm_type;
  typedef bimap< set_of<int>, set_of<std::string> >
bm_type2; assert( typeid( bm_type ) == typeid( bm_type2 ) );
```

So to solve our problem about unique keys, we can simply write:

```
typedef bimap< multiset_of<int>,
               set_of<std::string> > bm_type;
bm_type bm;
bm.insert( bm_type::value_type( 1, "one" ) );
bm.insert( bm_type::value_type( 1, "1"   ) );
bm.insert( bm_type::value_type( 2, "one" ) );
assert( bm.size() == 2 );
```

This is f****** ingenious! To allow the third insertion, we could simply say

```
typedef bimap< multiset_of<int>,
               multiset_of<std::string> > bm_type;
```

Boost.Bimap (7)

We can choose among a variety of configurations:

1. `set_of`
2. `multi_set_of`
3. `unordered_set_of`
4. `unordered_multiset_of`
5. `list_of`
6. `vector_of`

We can also configure the individual relations just like with standard containers:

```
  typedef bimap< multiset_of<int,std::greater<int> >,
                 unordered_set_of<std::string,ignore_case> >
                     bm_type;
```

Boost.Bimap (8)

Using Boost.Lambda placeholders, we can do some very cool searches:

```
typedef bimap<int,std::string> bm_type;
bm_type bm;
...
bm_type::left_range_type r;

r = bm.left.range( 20 <= _key, _key <= 50 ); // [20,50]

r = bm.left.range( 20 <  _key, _key <  50 ); // (20,50)

r = bm.left.range( 20 <= _key, _key <  50 ); // [20,50)

r = bm.left.range( 20 <= _key, unbounded ); // [20,inf)

r = bm.left.range( unbounded , _key < 50 ); // (-inf,50)

r = bm.left.range( unbounded , unbounded ); // (-inf,inf)
```

The last search is equivalent to `std::make_pair(cont.begin(),cont.end())`.

Boost.MultiIndex

Boost.Bimap is actually build on this library.

```
typedef multi_index_container< employee,
                indexed_by<
    // sort by employee::operator<
    ordered_unique<identity<employee> >,

    // sort by less<string> on name
    ordered_non_unique<member<employee,std::string,
                                &employee::name> >
                        >
> employee_set;

void print_out_by_name( const employee_set& es )
{
  const employee_set::nth_index<1>::type&
                                name_index = es.get<1>();
  std::copy( name_index.begin(), name_index.end(),
             std::ostream_iterator<employee>(std::cout) );
}
```

Metaprogramming makes this type-safe and efficient. Unlike Bimap, there is no limit to how many indexes you can create. The class is therefore useful as an in-memory database.

Flat Sets and Flat Maps

Often called sorted lists or sorted vectors. The classes exposes the same interface as `std::set<Ket>` or `std::map<Key,T>` on top of a buffer like `vector<T>`:

```
boost::interprocess::flat_set<int> idSet;
idSet.insert( 42 );
boost::interprocess::flat_map<int,int> idToSizeMap;
idToSizeMap[42] = 42;
```

- However, the is no overhead associated with node-allocations which can make the them *much* faster for small maps of small types.
- The reference and iterator invalidation scheme then follows the underlying container, and not the rules for normal node-based associative containers.
- They are very useful. E.g. if used in an adjacency list representation of a graph, they allow us to detect edges in $O(\log n)$ time while keeping the storage as low as with a list.

Intrusive Containers

Since dynamic node allocation is often the major performance bottleneck, the question arises on how we may avoid it.

```
class Node
{
private:
    const Node*                parent_;
    std::vector<const Node*>   children_;
    // other data members

public: // construction
    explicit Node( const Node* parent = 0 );

public: // modifiers
    void addChildNode( const Node& n )
    { children_.push_back( n ); }
```

If the pointers are never invalidated, we can put such nodes on the stack.

Intrusive Containers (2)

Luckily, a container like `std::deque<T>` never invalidates references to inserted elements (unless we call `erase()`):

```
class Graph
{
    std::deque<Node> nodes_;

    void grow()
    {
        Node n(parent);
        nodes_.push_back( n );
        parent->addChild( nodes_.back() );
        ...
    }
```

Of course, this does require a little carefulness. For example, as above, we cannot call `addChild()` before we have a *stable* reference to the object.

Notice that we can not use `std::vector<T>` here—`std::deque<T>` is *essential* here.

Boost.Intrusive

This method has been generalized with Boost.Intrusive:

```
#include <boost/intrusive/list.hpp>

using namespace boost::intrusive;

class Foo : public list_base_hook<>
 /**/ ;
```

This adds the necessary pointers to your class. We can then use it:

```
typedef list<Foo> FooList;
Foo     foo;
FooList list;
list.push_back( foo ); // fast
assert(&list.front() == &foo_object);
```

Like in our home brewed scheme, we manage the life-time of the objects in e.g. an `std::deque<T>`.

Boost.Intrusive (2)

The library defines a variety of intrusive containers:

1. `slist` : a single-linked list.
2. `list` : a double-linked list.
3. `set/map` : balanced binary trees.
4. ... and many more sets (even hash-based)!

Of course, different container require you to derive from (or define) different node-structures.

To summarize, the benefits are

1. It is *very* fast. Memory allocation overhead is minimal, and localization of data gives better cache hit rates.
2. Better exception-safety guarantees.
3. We can compute an iterator to an element in $O(1)$ time.

This is a *brilliant* library!

Boost.Serialization

C++ has no reflection mechanism, and can as such not provide automatic serialization. A custom method affords more flexibility and performance though:

```
class gps_position
{
private:
    friend class boost::serialization::access;

    template<class Archive>
    void serialize(Archive& ar, unsigned version)
    {
        ar & degrees;
        ar & minutes;
        ar & seconds;
    }
    int degrees;
    int minutes;
    float seconds;
    // ...
};
```

This is a *beautiful* example of how to exploit **symmetry**.

Boost.Serialization (2)

We can then save and load the data:

```
gps_position g(35, 59, 24.567f);
{
    std::ofstream ofs("filename");
    boost::archive::text_oarchive oa(ofs);
    oa « g;
}

gps_position newg;
{
    std::ifstream ifs("filename");
    boost::archive::text_iarchive ia(ifs);
    ia » newg;
}
assert( g == newg );
```

The library comes with a variety of archives: xml, binary and text. Portable binary archives also exists.

Boost.Optional

Often you have a function which return a value, but which might fail. Here are some examples.

```
double sqrt(double n );
char get_async_input();
point polygon::any_point_inside();
```

Throwing an exception is out of the question, because we *must* handle a "missing" return value.So we do the following:

```
std::pair<char,bool>  get_async_input();
std::pair<point,bool> polygon::any_point_inside();
```

Alternatively, we add an extra `bool&` parameter.

We then might use it

```
std::pair<char,bool> p = get_async_input();
if ( p.second )
   do_something(p.first);
```

Boost.Optional (2)

Instead we have a class that may delay construction:

```
boost::optional<char>  get_async_input();
boost::optional<point> polygon::any_point_inside();
```

By default, the objects are empty:

```
boost::optional<int> i;
assert( !i );
```

This conversion to "something boolean" is then used idiomatically:

```
boost::optional<Bar> bar;
...
if( bar )
{
    std::cout « *bar « std::endl;
    bar->foo();
}
```

So it is modelled a bit like a smart pointer, but copy-semantics are deep and with a const-correct interface.

Boost.Optional (3)

So because of delayed construction `optional<T>` is a great tool if

- your type does not have a default constructor, and/or
- the default constructor is expensive, and you want to avoid that overhead.

Then how do we initialize it?

```
boost::optional<int> i = 42;
assert( i );
```

Alternatively, we might *forward* an arbitrary list of arguments with **in-place factories**:

```
typedef boost::optional< std::vector<int> > optional;
optional vec = boost::in_place(42u,42u);
assert( vec->size() == 42u );
assert( vec[0] == 42u );
```

This avoids needless copying in the current language.

Boost.Optional (4)

To implement delayed construction, `optional<T>` must use **placement new**:

```
void* operator new(void*, std::size_t);
```

We can use this operator to construct an object in some memory region of our choice:

```
template< class T >
class optional
{
    boost::aligned_storage<sizeof(T)> memory_;
    ...
    optional( const T& )
    {
        new (memory_.address()) T;
    }
   ~optional() { get().~T(); }
```

Placement new is also used heavily in containers. The normal operator new always return suitably aligned storage, but with placement new, it is a precondition that memory is aligned properly.

And The Rest ...

We never got to talk about these libraries. But I think it is useful to know they exists:

- Boost.DateTime: when you need to work with calendars and time.
- Boost.Filesystem: when you need to traverse folders and manipulate files.
- Boost.Format: when you need *type-safe* printf-like formatting.
- Boost.IOStreams: when you need to *filter* or *compress* data coming from/going into standard streams.
- Boost.Iterator: when you are writing your own container and needs to define custom iterators.
- Boost.Regex: when you need to apply regular expressions.
- Boost.Units: when you need to implement code that relies on physical formulas.
- Boost.Signals: when you need to implement the observer pattern.
- Boost.Asio: when you need to work with sockets.

Summary

- Use `circular_buffer<T>` for efficient queues.
- Use `bimap<T,U>` when you need bi-directional mappings.
- Use `multi_index_container` when you need more than two views on your data.
- Use `flat_set<T>` or `flat_map<T,U>` when types are small and cheap to copy, and when the number of stored objects is low.
- Use intrusive containers (homebrewed or Boost.Intrusive) to avoid the bottleneck of node-allocation.
- Manual serialization is efficient and flexible with Boost.Serialization.
- Use `optional<T>` to delay construction, to avoid default construction, or to specify results that can be empty.
- Remember that placement new requires memory to be aligned.