

Re: Maintenance of c++ code

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2006-03/msg00033.html>

- *From:* piotr5@xxxxxxxxxxxxxxxxxxxx (Piotr Sawuk)
 - *Date:* 05 Mar 2006 10:39:42 GMT
-

In article <postmaster-D6D607.10545701032006@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>, "Daniel T." <postmaster@xxxxxxxxxxxxxx> writes:

In article <dtvfmf\$j0\$1@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>, piotr5@xxxxxxxxxxxxxxxxxxxx (Piotr Sawuk) wrote:

In article <postmaster-84682A.14091723022006@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>, "Daniel T." <postmaster@xxxxxxxxxxxxxx> writes:

In article <dt91dn\$me4\$1@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>, piotr5@xxxxxxxxxxxxxxxxxxxx (Piotr Sawuk) wrote:

The comment is supposed to answer the question: What does the function do? (The code tells us how it does it, but that isn't always the most clear picture.)

I see, so for example:

```
template<class V>
class rIt : public std::list<V>::reverse_iterator
{
...

//applies non-symmetric operator& in reverse
static int op(const V& p1,const V& p2) {return p2 & p1;}
};
```

But this leaves the question where to put the comments for the

Re: Maintenance of c++ code

template-interface? For example above `op()` is contained in both, `rIt` and `It`, so that using one of them as a template-parameter would yield a special behaviour which isn't obvious from the code.

In this case the comment would be: "`op(p1,p2)` executes operator&

The point is to provide a brief, high level description of the function.

OK, so I do need to provide a comment for abstract functions, instead of merely explaining what their code is doing! In your example your comment uses a technical term for explaining the time-efficiency and the output of your function, I bet it would be possible to also invent such technical terms for my comment -- I just didn't wish to confuse the reader with made-up phrases like for example "`op(*p1,*p2)` applies '&' in the iterator-order of `p1,p2`" or similar.

You somehow keep forgetting that not OOP with virtual functions, but OOP with templates is a whole new world for me...

That's a problem, because templates are not a method of implementing OOP. They are not a means of doing different things with the same (abstract) type, they are a means of doing the same thing with different (concrete) types. Templates implement generic programming not object-oriented programming.

Actually OOP does more: it allows doing different things with the same data-structure, provided the VTable is stored together with this structure. This allows for storing an object and retrieving it later on for this OOP-stuff. Templates truly can't do that! My opinion isn't that far from yours, I just emphasize efficiency. Maybe the compiler is able to optimize for early-binding, but I don't think any c++-compiler would be able to optimize out the VTable and pass objects in the registers of the underlying assembly-language when "pass by reference" was used in the function-definition. Only higher-level languages can do that!

Using templates is as good as true abstract types, it's just lacking in the late-binding department (there is none). But as my experience told me, late-binding is actually rarely needed (i.e. many abstract objects and the functions using them could be re-written so that only early-binding is used and no additional member for type-identification is needed). It's just that the use of templates is a design-decision which can't be un-done later on that easily, and which is somehow incompatible to the design-decision of using virtual members (because typenamees are not "virtual", and

Re: Maintenance of c++ code

sending the base-class will always slice-off all typedefs which are important for template-stuff). In the same way as defining a function needs to take care to take parameters only by reference, (since otherwise those parameter's virtual members wont be used correctly), in the same way defining a templated function needs to take care to create an individual template-parameter for each parameter of the function. For example as I learned stl-algorithms are a bad design for OOP through templates because iterator-ranges are expected to consist of 2 equal types (instead of making one iterator constant in the standard, such that end-conditions could be handled individually depending on each iterator's type). And at the same time stl-algorithms are a bad design for real OOP since iterators are taken as a pair of equal type instead of taking one of them by reference. just try to use some stl-algorithm on:

```
template <class Iterator_>
class double_iterator : public Iterator_
{
public:
typedef Iterator_ iterator_type;
typedef double_iterator<Iterator_> Self_;

double_iterator() : iterator_type() {}
explicit double_iterator(iterator_type i) : iterator_type(i) {}

double_iterator(const Self_ &i) :
iterator_type(static_cast<iterator_type>(i)) {}
template <class Iter_>
double_iterator(const double_iterator<Iter_>& i)
: iterator_type(static_cast<iterator_type>(i)) {}

Self_ & operator++() {
iterator_type::operator++();
iterator_type::operator++();
return *this;
}
Self_ operator++(int) {
Self_ tmp = *this;
iterator_type::operator++();
iterator_type::operator++();
return tmp;
}
Self_ & operator--() {
iterator_type::operator--();
iterator_type::operator--();
return *this;
}
Self_ operator--(int) {
Self_ tmp = *this;
iterator_type::operator--();
iterator_type::operator--();
}
```

Re: Maintenance of c++ code

```
return tmp;
}
};

//TODO:I don't think double_iterator<double_iterator> would work as expected!
//Warning: "==" and "!=" are asymmetric: the second argument is turned into
// a range [j,j+2) in which first argument is searched for...
template <class Iter_>
bool operator==(const double_iterator<Iter_> i,
const Iter_ j)
{
Iter_ tmp=static_cast<Iter_>(i);
return tmp == j || --tmp == j;
}
template <class Iter_>
bool operator==(const Iter_ j,
const double_iterator<Iter_> i)
{
Iter_ tmp=static_cast<Iter_>(i);
return tmp == j || ++tmp == j;
}
template <class Iter_>
bool operator!=(const double_iterator<Iter_> i,
const Iter_ j)
{return !(i == j);}
template <class Iter_>
bool operator!=(const Iter_ j,
const double_iterator<Iter_> i)
{return !(j == i);}
```

handling those highly unlikely cases. But the idea to merely display proper usage isn't new for me, it's what I did in the examples posted here — just no asserts since my code is supposed to be used in combination with other code which could get thoroughly tested.

Unfortunately you didn't do that, at least not in what you posted, and I expect you didn't do it at all because the code didn't do what you said it should do.

You're right, I made the mistake to merge my test right into the code, for the purpose of deleting it once the debugger told me that the code is producing the expected result. As to what the code is supposed to do:

Re: Maintance of c++ code

No, it's supposed to find the first integer in the list for which the operator "p1 & (p2+1)" is zero, i.e. the first number *j with *j+1 having none of *i's bits set to 1 (as opposed to "&&" which is the logical true-false and-operator of c++). The return-value is undefined in the other cases because my object is supposed to hold such a value. So, you were right that my program had bugs (missing braces), but my intentions were correctly formulated in the text...

is not what your find_sawuk_number did do because I forgot to put braces around "*j" before adding 1. However in the beginning of this thread I said that my code is supposed to demonstrate different behaviour of 2 operators depending on the variable "sig" defined

Re: Maintance of c++ code

during object–construction. I never said that my code would do anything other than demonstrating this. In the later version of my code I merely dropped the variable "sig" and used typedefs instead.

One unit == one block, one function, one class, one package, one program... At each level there are tests that must be performed to ensure that the particular unit in question does what it's supposed to do.

Thanks. So it seems, putting asserts all over the place is enough for an unit–test of those smaller pieces when the big bundle is tested anyway...

Anyway, each container–type has a different storage–policy, and there are many more possibilities than those few defined in stl. In my opinion it is impossible to create code which is useful for all cases. That's just a warning. Now I'll try to forget this prejudice and consider your proposal.

Your quite right. Some algorithms in the standard library work with any container, some only work with particular containers. What's your point?

My point are user–defined containers! As I said below, for example the map–container does store my objects in key–value pairs, and a container created by some other programmer might create some different means of accessing the values of his container. It's true that I'm stressing "code–reuse" a little bit too much, but I for example would like to use stl's find–algorithm together with above double_iterator class without the need of some means for tracking whether the container stores an even or an odd number of elements, or if the container's iterator has been increased an even or an odd amount of times...

But as I said, it's just a prejudice, maybe I really missed the point...

Re: Maintenance of c++ code

An iterator is *any* class that can be used in an iterator context, that is:

```
// needed for output iterators
```

```
// for forward iterators, all of the above
```

```
// for bidirectional iterators, all of the above and  
--p and p--
```

```
// for random access iterators, all of the above and
```

For what you are doing, the implementation of a bidirectional `reverse_iterator` would be useful to look at. All it needs to be parameterized on is the type of bidirectional iterator it's reversing.

Your iterator should work much like the `reverse_iterator` except you have a method to change it's direction. The `reverse_iterator` template class doesn't need to know anything about the container it is iterating over.

You're right, `begin()` and `end()` could be templated to work with any container which provides an `rbegin()` and `rend()` member-function...

Of course if you want your iterator to *also* have the characteristics of a container (ie have a `begin()` and `end()` member-function) then I'd say your class is not separating its concerns very well.

The only reason why `begin()` and `end()` are static member-functions is because of asserting that the element searched for is actually in the container (as opposed to relying on the segfault occurring when the value of `end()` is accessed). However, in my `ConvexHull` program (just say a word and I will post it here) I also have a container which does require `begin()` and `end()` to be a static member of a similar iterator, since part of the information stored in the container is encoded in the position of `end()` relative to the elements. That's why it got there in the first place. If the stl-container my own container is based on would have random-access iterators, then also this `begin()` and `end()` would stop being needed. you are right that I probably should re-think my design and start using some sort of dynamic segmented array (without random-access-iterators) instead. That way I could track if an iterator is odd or even at constant time, without loosing constant-time front-insertion. Unfortunately my code is heavily relying on the fact that iterators are never invalidated (i.e. I would need to use vector instead of a simple array), and I see no possibility how this could become more efficient

Re: Maintenance of c++ code

than my approach with begin() and end() being functions of the iterators...

Further you told me to evade inheriting the original iterators, and that means that I would need to implement all the functions which are usually part of an iterator: ++, --, *, ->, ==,... Of course those would merely consist of calling the appropriate iterator's functions and returning a new object initialized with the result. Also the iterator's typedefs need to be re-implemented if I wish to use any of stl's generic functions. Could you explain to me what gain I would receive from this additional work?

You need to implement almost all those things anyway because your iterators behavior is different than a list iterator's behavior.

No, it's exactly the same except for some additional member-functions. But you're right, no stl-typedefs are needed with std::iterator. I already thought about creating a virtual_iterator template with all operators being virtual functions, but then I realized that none of the stl-algorithms does actually take its iterator-parameters by reference anyway. Also, std::iterator doesn't provide virtual operators, it even doesn't encapsulate any kind of value-type (i.e. your class below is missing "protected: int value"

Once you have a function or class working with a particular concrete type, ask yourself, would it work with other concrete types? If so, and those types aren't related by inheritance, then the function or class would work well as a template...

I see, so that's what those stl-developers forgot to do: repeat above for *all* abstraction-levels of the terms "input" and "output"...

I'm currently working on a group of template classes that implement the active object pattern and will work with many different and otherwise completely unrelated classes. The idea is that by wrapping an object of some class with my "ActiveObject" all member-functions will return immediately rather than block until complete.

Interesting. And how does your class access the names of the classes'

Re: Maintance of c++ code

member-functions? Through template-parameters? Or do you abuse the VTable (i.e. override the usual dispatching)? Or do you merely apply that behaviour on the classes' standard operators? I never heard of any general-purpose pattern-implementation, there always was some case where the implementation couldn't be used for making use of the pattern it's supposed to implement...

I leave you with some good advice (IMHO) from Allen Holub:

functionality described in the paragraph. The excuse "I didn't have the time to add in the comments" is really saying "I didn't design the code before I wrote it and don't have time to reverse engineer it."

Hehe, design is truely a critical thing, unfortunately for me it is impossible to do any design without knowing what is possible in a programming-language. So test-compiling is an important part of the design-phase for me. such test-code then unfortunately does stay without comments...

Better send the eMails to netscape.net, as to evade useless burthening of my provider's /dev/null...

P

.