ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 23

December 1997

Contents

Editorial

From C++ To C

Once upon a time I held sway over a set of interacting multi-threaded objects marshaled with the power of object oriented design methods. Now I find myself faced with a large body of aging C code, written by mean fisted unix wizards. You know that tight scrawl they have: 40 columns wide, 80 columns deep. I've got that sleepless 'rewrite' feeling. But, change it, and break it. Not wise on a schedule that runs in internet time (one year = three months).

Why C?

This code builds and ships on 17 platforms. C is portable. C++ is not. But, the big news of this issue is that it will be next March! (See Francis' column in the Standard section.)

The C++ Standard has taken so long to arrive that many projects have switched to Java to provide the 'write once, run anywhere' holy grail.

Bad C

So, we're stuck with C for the moment. But, it's not the language that makes code bad. You can write really good C. So what's unpleasant about this code:

- Big functions
- Many exit points
- Backward and forward goto's. Both to repeat algorithms, and to clean up re-sources.
- Unused functions, declarations, and in-cludes.
- No function naming conventions.
- Features thinly spread over the source instead of tightly corralled in their own modules.
- Functions with many side-effects, instead of one operation.
- Transfer of object ownership between caller and callee.
- No hope for reuse.

Good C

So how can this general badness be migrated to something more palatable?

- Enforce partitioning. Low coupling, high cohesion.
- Define module interfaces.
- Provide explicit structure constructors and deconstructors.
- Provide object scoping with this pointers
- Naming convention of <ob-ject>_<operation>

Look at Safer C and Lakos for help

How can you build new features, in a timely manor, on top of old code? I think you've got to spend time re-organising and rewriting.

Movies

It's odd to work in a company where people can make it their lifestyle. They feed you, clothe you, and entertain you. Free food, drink, t-shirts, jackets, and film tickets. They'll book your holiday, arrange an evening out, send a birthday gift, or wait at home for the plumber. There's even a dentist on wheels. Weird!

Anyway, the point is that I've been to see some films recently, and I know you won't have got them yet, so I get to do some film reviews.

Starship Troopers was really bad. Teenage romance script with lots of limbs being ripped off. Unfortunately not so bad that it enters the 'so bad it's funny' category.

Alien Resurrection was ok'ish. Not as bad as Alien 3, but not as good as 1 or 2. A few neat twists on the theme but they've run out of ideas, and the ending was soppy.

Gattaca. Fantastic. Set in a near future around the theme of human genetic manipulation. Engrossing complex story with great cinematic atmosphere.

Sign off

Appropriate seasonal salutations to you all, and remember… every child is expecting an LDAP server under the tree this Christmas!

Copy Deadline

All articles intended for publication in *Overload 24* should be submitted to the editor, by Janurary 15th.

John Merrells
merrells@netscape.com

Software Development in C++

UML - Objects and Patterns by Richard Blundell

Introduction

Last time I gave some background on the *Unified Modelling Language* (UML), and discussed how to use the UML to describe and document classes and their relationships with other classes [1]. This time, after a brief memory jogger, I shall discuss some changes that have occurred following the publication of version 1.1 of the UML in September. After this, I shall show how objects, rather than classes, can be represented, and how the concepts covered so far can be used to describe typical design patterns.

A Refresher

To jog your memory, classes were shown as rectangles with up to three compartments, the top one containing the class name, the next showing the class *attributes* (member variables), and the bottom one listing class *operations* (methods). *Associations* between classes were shown using solid lines. An *aggregation* diamond was used to show how one class could hold references (i.e. C++ pointers or references) to others, whereas a filled diamond showed *composition*. *Roles* of associations and *multiplicity* values were used to adorn each end of association lines to add further information. Finally, a hollow *generalisation* arrow was employed to show inheritance relationships between classes. Examples of some of the notation covered are shown in the figures below.

UML 1.1

At the beginning of September 1997, version 1.1 of the UML was published. As well as some extensions, simplifications and unifications of the existing notation, there were a number of small changes that either modified or extended what I described last time. In order to keep abreast of the latest developments, I shall quickly cover some of the main changes to static structure diagrams here.

The *list compartments* described last time can now hold the name of the compartment centred at the top, so you may see classes with the labels "attributes" and "operations" explicitly showing which is which, and to minimise confusion if one is omitted. Furthermore, user-defined list compartments can now be appended to show additional information such as exceptions thrown by the class, or business requirements addressed. These user-defined compartments should, of course, be labeled appropriately and consistently.

Several handy boolean *properties*[1] have been defined for operations and attributes, and can appear, in curly braces, after the particular element in the list. The **{frozen}** property[2] signifies an attribute that cannot change once defined, such as a C++ const member variable. The **{query}** property does the equivalent thing for an operation, showing that it does not modify the state of the class instance when called. Typical "Get…" methods could be marked {query}. Finally, **{abstract}** signifies an operation that has no implementation defined, and so corresponds to a C++ pure virtual function. An alternative to this last one is to write abstract operations in italics. You often see

[1] I shall discuss properties and constraints in the future. For now, they can be taken to be tags that can be attached to elements in a model.

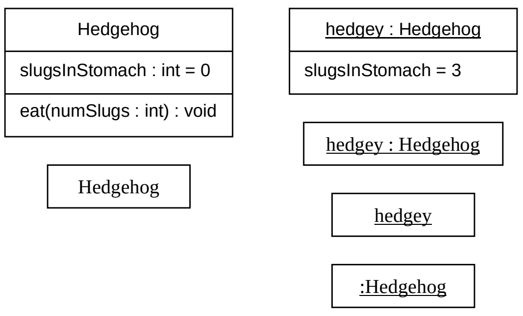
[2] This is shorthand for *{frozen = true}* – properties are key-value pairs, and the default for boolean properties is 'true.'

class names in italics as well if the classes are abstract.

Another extension was made to visibility markers, the symbols that show access levels to attributes and operations. Instead of the symbols +, # and -, the keywords *{public}*, *{protected}* and *{private}* can be used. These will be more familiar for those used to C++ and Java, and have the advantage that they can apply to blocks of attributes or operations just like in a class declaration, and so need not be continually repeated. Some tools, however, tend to use coloured icons for visibility markers by default.

Objects

In the UML, there is a distinction that is made between *types* and *instances*. A number of model elements in the UML are members of type-instance pairs. For example, objects are *instances* of a class[3] (the corresponding *type*). Similarly, the *associations* between classes that we saw last time have instances called *links*, which show how objects interrelate. *Parameters* passed to operations can be thought of as types, whereas the *values* that are bound to them at run time are instances of those parameters. A *call* itself can be thought of as an instance of an *operation*. In most of these cases, because of the similarity between each member of a pair, the same symbol is used to describe both. The difference between them is often shown by underlining the name of the element (and by giving it a different name as will be seen later).



[3] In fact, in UML, a class can itself be an instance of the type *metaclass.*

Figure 1 – Classes and Objects in the UML. On the left we have two representations of a class. On the right we have four representations of an object, including an unnamed object at the bottom.

Examples of representations of a class and objects of that class are shown in figure 1. Note the similarity between the two. Objects are shown as a rectangle with a name compartment at the top, and an optional attribute list below it. Operations are not usually shown in objects, since they are defined in the class.

The name compartment holds the name of the object in the form:

```
object-name : class
```

The class is optional, as is the object name (although not both!). A nameless object is used to denote anonymous objects, which can be used to demonstrate the role of an object without specifying any particular instance. In the attribute list, the types of attributes are often omitted, since these are set in stone in the class element and so do not need to be listed again. Instead, values for the attribute can be shown (see the top example in the figure), or a series of values can be given to show how the attribute changes over the course of some process.

Links

Associations exist between classes, and *links* exist between objects. Links tend to omit some of the information that the corresponding association would show, in the same way that objects omit some of the details shown in classes. An association name can be shown next to the link, and if so, it is underlined to show that the link is an instance of that association. Role names can also be added, but multiplicity is often not necessary.

Static methods and members

Last time I described the visibility marker for class-operations and class-attributes (i.e.

static methods and members in C++), as opposed to object or instance ones, although I am not quite sure why! The more usual way to show a class operation or attribute is to underline it. This is another case of the type-instance pairing in the UML. A static class method or static member variable can be thought of as class-wide and hence exist as instances per class rather than per object. They are, in a sense, already instantiated, and can be called or used without any appropriate objects being available.

Design Patterns

To describe the workings of a particular design pattern, the classes (and objects) involved in the pattern are described and their interactions are defined. A static structure diagram, as introduced in the last article, fits the bill exactly, because all inheritance and composition information can be documented, along with associations and any required operations and attributes of the collaborating entities. It is probably time to look at a few examples.

The Prototype pattern [2] is basically a pattern that enables the easy creation of objects by a client without it needing to know what the objects actually are in advance. This is achieved by requiring all classes of objects that it is to create derive from a common abstract base class (ABC) that it *does* know about. This ABC has a virtual method called something like clone(), which allows a new object to be cloned from an existing one. All the client then needs to do is take an existing object, call clone() on it, and Hey Presto!, a new object is created, no matter what derived class it belongs to. The basic arrangement is shown in figure 2.

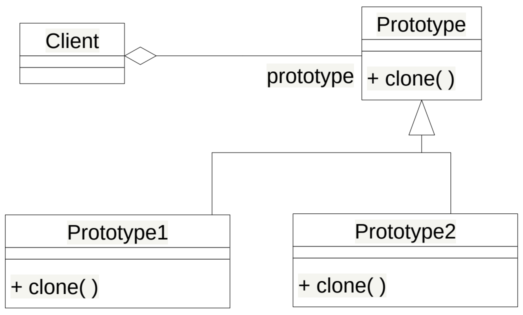


Figure 2 – The Prototype pattern

As can be seen from the figure, the client has a reference (in the UML sense – it may well be a pointer in the code) to an object derived from Prototype. When it wants to create a new one, it just calls the virtual clone() method, which returns a pointer to the new object. If, later on, a new derived class is added, then the client can continue to create objects of this class as long as it has one to start off with. You could, for example, hold a list of available objects somewhere that the client can use as its initial objects to clone.

To use this pattern in a system, you would replace the standard class roles shown above with the classes in your system that participate to use the pattern. An example for this is shown in figure 3, and this may help to explain the operation in a less theoretical manner. Here we have a *GeneticEngineer* class. Objects of this class can then clone any *Animal* that happens to walk by, even if they don't know what they are! (The recent fuss in the papers about cloning *Sheep* was obviously overstated 8-). Note the UML *note* that I have used to annotate the GeneticEngineer class, showing an idea of the implementation of the createLife() method.

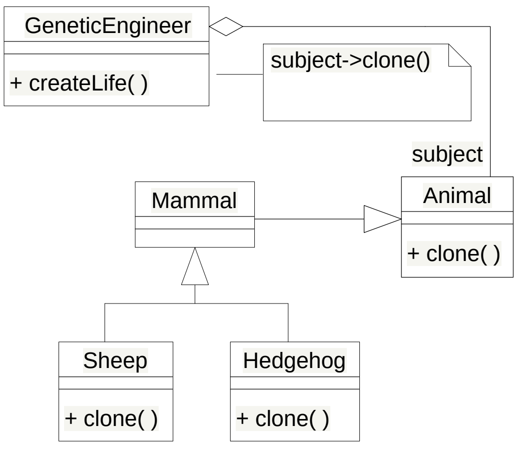


Figure 3 – The prototype pattern in use

Patterns as Collaborations

Typical system models contain many interacting classes, and it is sometimes useful to highlight the presence of a particular pattern within the model. This can be achieved using a dashed collaboration ellipse, as shown in figure 4, which is labelled with the name of the pattern. Dashed lines link the classes (or objects) that are involved in the collaboration to the ellipse, and these lines are labelled with the standard role names within the pattern. This is a concise method of indicating the presence of patterns within a model.

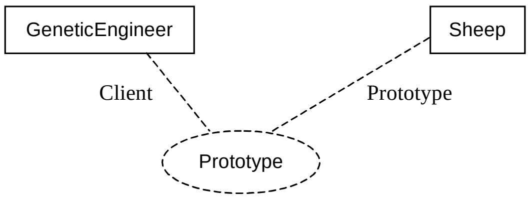


Figure 4 – Using a collaboration symbol

Another pattern is documented in figure 5, with the presence of an object using a third pattern highlighted. The main section of the diagram shows a set of classes co-operating to form the Observer pattern. This pattern enables objects to subscribe to a "service," if you like, provided by a Subject. The Hedgehog class, which is derived from Observer, can subscribe to the service provided by SlugFarm, by calling its `attach()` method. SlugFarm then adds this Hedgehog to its list of observers. Whenever something happens that subscribers might want to know about, such as the birth of a new Slug, it `update()`s all of its observers. The updated information can come from one of the observers as well. If a Hedgehog eats one of the SlugFarm's slugs, it can `notify()` the SlugFarm, which then `update()`s all of the other Hedgehogs.

The other pattern shown in the figure is the Singleton pattern. The singleton object's required uniqueness is indicated by showing that the Singleton pattern is in use. Only one instance can therefore be created in the lifetime of the system.

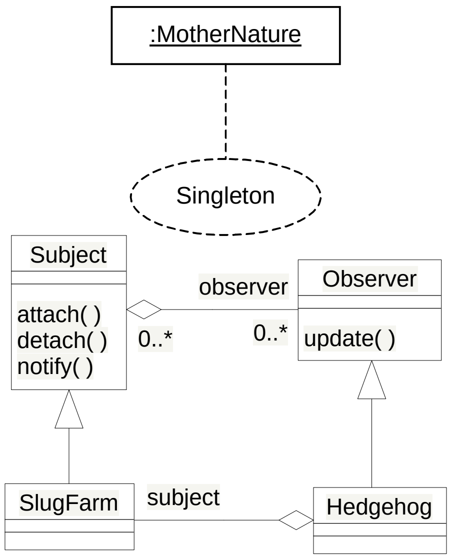


Figure 5 – The Observer pattern and a Singleton object.

Conclusion

Apart from some finer details, we have now covered static structure diagrams, which can contain classes and objects (*class diagrams* and *object diagrams* are common terms for static structure diagrams that contain predominantly classes and objects respectively). We have also covered briefly the description of patterns in the UML. Next time I plan to show how to document some of the dynamic behaviour of systems.

Richard Bundell
rpb@mail.ndirect.co.uk

References

[1] Blundell, R.P., *An Introduction to the UML*, Overload 22 pp 7-10.

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Candidates for inline by Francis Glassborow

I was a little stunned by Alan Griffiths' article titled 'Premature Optimisation.' Many programmers currently seriously overuse the inline keyword. The cure to this is not an almost draconian prohibition of its use but a carefully considered list of places where it should be used by default and a list of places where it should be considered for a release version. It is then entirely reasonable to require programmers to sign-off on all other uses.

I have discussed the issue of the extra compile time overhead for the extra function names with several implementors. For what it is worth their unanimous opinion is that if it matters there are problems with your coding technique because your translation units are getting too big. They point out that the name of an inline function never escapes a translation unit unless it is not actually inlined (its address is taken or the compiler rejects your hint).

On the other hand there are places where the decision to inline should be seriously questioned if you have an inferior linker. For example a function that includes a local static can seriously add to the complexity of link time operations. More about this later.

I completely agree with Alan that stability of header files is important. A decision to change a low-level header file should be treated with grave suspicion because there will be a massive one time cost for doing so. Those developing large projects should consider having a few fixed dates each year when any desirable changes are implemented across the board so that you pay the price once for all the changes that are made.

The process of development of higher level components should be isolated from the development of a complete application. There are well known mechanisms for doing this such as the so-called 'Cheshire Cat' handle class. One interesting feature of this technique is that you can quickly recover efficiency for a release version by converting most of the handle's interface to inline forwarding functions for building the release version. The decision to do this should depend on whether the added performance is worth having (the answer in most cases is "It isn't")

Now let me look at candidates for my list of when to inline.

Candidates for Inlining Qualification

Accessors

Consider:

```
class X {
  int i;
public:
  int get() { return i;}
  void put(int newValue) {i = newValue;}
};
```

Any change to this class would, of course, cause recompilation of all translation units that include it directly or indirectly. However it would be rare to change the access functions unless the data was itself changed. I know that hypothetically we can change the access functions but such isolated change is extremely rare and almost certainly signifies a conceptual change to the public interface even if the actual function declarations have been left alone.

Contrast this case with (a classical style class):

```
class Y {
  char * mystr;
public:
  // suitable constructors & destructor
  char const * get() { return mystr; }
  void  put(char const * );
};
```

The get() function's implementation is again the only reasonable way to provide read access and so the burden of justifying not-inlining it lies with the opposition. On the other hand any reasonable implementation of the put() function is going to be more than a single statement so any suggestion that it should be inlined should be treated with grave suspicion. However if you change the implementation to:

```
class newY {
  string mystr;
public:
  // suitable constructors & destructor
  char const * get()
    { return mystr.c_str(); }
};
```

You might revisit the decision, as I have, because we now have a simple call to a member function of string. The time to make the change is when you change the type of the data member, which will force recompilation anyway. At the same time you might wish to augment the interface with a function that returns a string const &. Note that such a decision starts to lock down your implementation in a way that return by value does not because the reference requires an object to bind to that has some continued existence. I do not remember seeing comments about this problem in any of the books that I have read. Perhaps it is worth some thought. One thought in passing is that a const & can bind to a temporary so it would seem that:

```
string const & wrong(){
  string x = "abc";
  return x;
}
```

generates a hanging reference because the return value is bound to x that is destroyed on exit from wrong() while:

```
string const & right(){
  return "abc";
}
```

should work as the return is bound to a temporary that is only destroyed when the reference is destroyed.

I would not want to write code that relied on such a tenuous distinction.

Forwarding Functions

There are numerous cases where you wish to transfer the provision of functionality to another function. In each case the forwarding function does not do anything so the only conceivable late change would be to replace the forwarding mechanism with a direct provision of functionality. Realistically such a change only happens because a change somewhere else invalidates the forwarding action. Such changes are very rare and almost always result in substantial changes that cause general recompilation of dependant code.

A simple example to help you understand what I am writing about:

```
class Degree {
  double angle;
public:
  explicit Degree(double a = 0) :
    angle(a) {}
  // all the other defaults do the right
  // thing provide complete set of
  // arithmetic operators. For example:
  Degrees & operator +=(Degree);
  Degrees & operator *=(double);
};
```

Note that it would be wrong to provide an operator double because that would allow you to multiply Degree by Degree, which is conceptually wrong. The significance of the use of explicit is that it prevents you adding any built-in arithmetic type to a Degree without making it explicit via a cast.

Now consider how we provide the normal operators

```
Degree operator+(
    Degree lhs, Degree const & rhs)
    {return lhs.operator +=(rhs);}
```

```
Degree operator*(
    Degree lhs, double rhs)
    {return lhs.operator+=(rhs);}
Degree operator*(
    double lhs, Degree rhs)
    {return rhs.operator+=(lhs);}
```

(Yes, I have already read the Harpist's article)

Now I think those three global functions (I would put them in a namespace if I was doing the job properly) should be inline definitions because I cannot imagine how that would be wrong. All they do is to forward the data to the function that does the real work. Indeed I have ensured that the compiler can see all it needs to manage the pass by const & without flushing cache's etc. Indeed if my manager instructed me that these were not to be inline I would start questioning his/her understanding of coding. (Now Alan can flame me for that provocative comment).

Of course you need to check that your forwarding function really does no more than forward to another function. I do not count evaluation of parameters because that has to be done whether or not you inline the function. Actually that is another place that the compiler may be able to provide extra optimisation.

Resolution of Overload Ambiguity

This is a special case of forwarding functions. The problem is how to deal with a case where your compiler claims that it can resolve a call to an overloaded function because of ambiguity. The classic solution from the programmer is to use a cast to force the correct selection. There are two problems with this. The first is that the programmer has to determine which is the correct choice. That may mean quite a detailed study of the actual component in order to reach a correct conclusion.

The second problem is that you have to litter your code with casts and thereby hide the ones that matter. A fundamental property of quality C++ code is the concept of locality. You should only do something once, and if you ever wish to change your mind it should require a single change to your code.

As the Harpist discovered the following code fragment meets a problem:

```
void fn(long);
void fn(double);
int main(){
  fn(1);     //ambiguity error
}
```

This may seem surprising. I would find it all the more surprising to see a cast in such a case. Every time you write superficial casts you lessen the significance of the vital ones. The correct way to resolve this ambiguity is to add another function to the overload set. In this case you would normally add:

```
inline void fn(int i)
{fn(long(i));}
```

I would need a whole lot of persuasion to remove that inline qualification. By the way, when compilers catch up with the recent decisions in standardising C++ you will be able to write:

```
inline void fn(int i)
{return fn(long(i));}
```

In other words the same form will work for all pure forwarding functions instead of having a different form for those that have a void return.

Does It Matter?

At application level the use of inline is probably an irrelevance as you are using high-level components. However if the designers of low-level components get it wrong you can be hit quite hard. One implementor I was talking with recently quoted me a performance gain of about 20% between code that never used inline and that which used it as outlined above. Changing low-level decisions does exactly what Alan dreads, forces large scale recompilation.

There is one special case that is worth consideration and that is the C++ idiom for providing static data (i.e. what some of you call global data). There are all kinds of problems surrounding such data in multiple translation units and its possible use before it has been initialised (during the program start-up). The idiom that side-steps this problem is to replace myType globalT; with:

```
myType & globalT()
{ static myType t; return t; }
```

Some programmers feel this is a good candidate for an inline function. I will not argue strongly either way on this. I can see good and bad points on both sides. Now that inline functions have extern linkage in C++ making such functions inline will work. As you are unlikely to be changing these very often in your code I see little risk of the decision to inline them generating lengthy recompilations. On the other hand the existence of a local static in an extern inline function might adversely affect the performance of more primitive linkers.

And that leads me to one final point. Alan grumbled about the template inclusion model. The reason for that is that current compiler technology results in quite unacceptable build times if separate template compilation is attempted. Almost all the implementors eighteen months ago stated that they could provide what Alan wants but the performance would be so bad that the only people who would ever use it would be those applying conformance tests. The remainder were not even willing to consider it.

So over to you. If you disagree I will need some hard evidence (real code written to the above specifications) not just mere speculation and hand waving. Otherwise I think it would be useful to consider what should be in my list of 'always inline' and what should be in my list of 'consider inline for release versions.'

Francis Glassborow
Francis@robinton.demon.co.uk

The Draft International C++ Standard

The Casting Vote by Francis Glassborow

Sean Corfield normally writes our reports of goings on at WG21/X3J16 meetings but as he was unable to attend the recent meeting in Morristown it falls to me to try to substitute for him. Fortunately this was not a meeting at which much technical work was done.

The main order of business was to resolve all the comments made on the National Body (NB) votes to promote Committee Draft 2 (CD2) to a Final Draft International Standard. Initially we had had five negative votes including the UK's. One of these was trivial to resolve. Martin O'Riordan had recommended that Ireland vote 'yes' with a single comment on the possibility of providing static versions of `operator new` and `operator delete`. His NB has a policy that comments only accompany negative votes so Ireland had voted 'no' with a single comment. Ireland's comment was discussed in some detail at Core 1 but we already knew that whatever the outcome Martin would be recommending that Ireland change its 'no' to 'yes'. In the event we decided that on technical grounds to leave the current constraint in place.

Australia had already signified that it was probably satisfied and as long as we did not make some drastic change to meet the requirements of another NB they would change their vote to 'yes.' I cannot remember who the third 'no' was from (I think it was Japan and they sent a message saying they were satisfied) and if I stop to find out this will not reach your editor in time. The other two were the UK who had declared that the version of `auto_ptr()` in CD2 was too dangerous for us to accept.

(You will find an article on this elsewhere in this issue).

Intensive email discussion had largely convinced others that we had a serious point and were not just being perverse. Many hours of technical deliberation (largely electronic) had resulted in what looked like a technical solution as a result of the insights of Greg Colvin and Bill Gibbons. Their proposed solution was presented at a technical session on Monday evening. It won the day with only a handful of negative votes (a great improvement from London where the Library Group had been about evenly divided on the then proposed solutions). By the time it came to a vote the next day there was only one negative vote (based on a belief that a small part of the solution would not work) and the WG21 votes were unanimous on the issue. We also added a non-normative note to emphasise what `auto_ptr()` was designed to do in the hope that this would further limit its abuse.

That changed the UK's vote to 'yes.'

Meantime a lot of small details were being cleaned up in response to NB comments. One silly constraint was removed – that which restricted the function used by `for_each()` to a non-mutating function. We could not see why this constraint had been included originally and we could not imagine an implementation that would need such a constraint. Our best guess was that the original intent was to prevent changing the container rather than the contents.

Some final work was done on the formal grammar and some work was done on the use of `typename`.

Almost everything was at this level, important but not controversial. There were lots of things that we would like to have done and there were a number of things that would have been done had it been 1995 rather than 1997.

The final controversy concerned a late comment from France. France had effectively been non-participants for several years but recently returned to active membership of WG21. When they voted on CD2 they stated that they had not had time to complete their understanding of template issues (join the merry band) and would comment in detail next time. Tom Plum, WG21 convenor drew their attention to the fact that they had been voting on a final CD and therefore there would be no next time for comments if the ballot was resolved in the affirmative. They asked us to consider late comments. In the interests of increasing international consensus we agreed.

Their major point concerned the point of instantiation of a template. Currently the requirement is not earlier than the point of use and not later than the end of the translation unit (I know that this is a loose paraphrase and experts would want to cross some 't's). If it makes a difference the program is ill-formed. France's problem with that is they would like to be able to use declared, but not yet defined, classes (what C calls incomplete types) in templates as long as the class is defined (completed) before the end of the translation unit (source code file). For that reason they wanted the point of instantiation to be strictly defined as the end of the translation unit.

This seems reasonable as it would only appear to make technically ill-defined code well-defined. Unfortunately it touches on a very sensitive area where we had lent over backwards to provide the maximum of freedom to compiler implementors (resolving conflict between separate compilation and the inclusion model). We had a technical session on Tuesday night (the final draft had to be ready for proofing by mid-day Thursday) and after almost two hours we still lacked consensus. The crunch issue was that even if all present were convinced of the merits of the change, we could not feel certain that absentees would agree. What was making things worse was that the only French representative was a comparatively young man who seemed to struggle with both English and the obscurer technical issues of C++ (to be fair, I would hate to discuss Java issues in French). The UK finally cut the knot by advising Tom Plum that the proposed change would endanger the UK 'yes' vote. In other words meeting France's requirement would not lead to greater consensus. Transatlantic phone calls (that five hour difference is awkward when you need to talk to a specific national expert on a technical issue) suggest that France may change their vote to 'yes' anyway.

On Thursday evening we attended an enjoyable reception put on by our official hosts (AT&T) with Andy Koenig and his friends entertaining us with a variety of recorder music spanning the centuries. That Andy was there effectively signified that the draft was complete and ready to be voted out as a FDIS. Late that evening Josée Lajoie (for once not head of the Canadian delegation) declared loudly 'Hey, We're done.' The printed word cannot possibly represent the mixture of surprise and delight that she managed to inject into that declaration.

Friday morning was just the formal voting and administrative closure. When we got bogged down on discussing future arrangements Mrs Plauger (Bill's wife) loudly advocated that we take a vacation. That seems excellent advice, as barring some completely unforeseeable catastrophe we have a C++ standard (ISO rules only allow the correction of blatant typos at this stage. Anything that could conceivably change the meaning is forbidden)

In March we will have to get down to the vitally important matters of planning for future meetings. Once we have a full IS we will need to maintain it by responding to defect reports and requests for clarification. We also need to start work on things such as Garbage Collection. We need to gain guided experience of such things so that there will be understanding based on experience by the time we start consideration of the next release of C++ (rest assured that work on the next release does not even start for almost six years)

Future Participation

As we move from the arcane discussion of how to specify the extremes of the language to the resolution of questions about C++ we reach a stage where far more people should involve themselves with national committees (or panels as BSI calls them). This is the best place to deepen your understanding and meet people who are likely to be able to clarify some of the murky issues.

The focus now changes from 'Should we do this or that?' to 'What exactly does this mean?' The best people to answer such questions are those who were involved in writing the text. It is time for the expert practitioner to join in. And one view of an expert is a person who understands how much more there is to learn.

UK members of ACCU who have paid their ISDF supplement and want access to standardisation documents should contact Neil Martin who will make appropriate arrangements. You should remember that access is for the purposes of standardisation and not so that you can give free copies of copyright materials to your friends.

Conclusion

I guess that there will be lots of commentators explaining how we could have done our work better. There will also be many people highlighting our mistakes. In such a massive undertaking there are bound to be some errors. However I think we have done an excellent job and I am proud to have been a participant. If you focus on using the language at an appropriate level of abstraction you will find that it works very well. Language lawyers have to focus on corner cases, but the ordinary user should keep away from the edges. As one committee member put it to me, there is a difference between a fence and a cliff edge, both are boundaries but exploring them by taking one extra experimental step has rather different consequences.

What the ordinary programmer now needs to do is to learn good coding habits based on a language that should remain stable for most of a decade. It may be a little time before all the compilers catch up but they now have a finishing line to cross.

Authors who want to write good books with extended shelf lives can now spend time getting the text right. Of course most won't as we have seen from the terrible quality of many books on C.

A more serious problem for the standards community is how we keep the team together. Some companies are going to look at a stable standard and question the benefits of funding further participation. I have no doubt that there are massive hidden benefits to continued participation. Not least of these is keeping your top experts happy. True experts need the chance to meet and to share with others, some of whom will have even more expertise.

Francis Glassborow
francis@robinton.demon.co.uk

The Story of auto_ptr A Cautionary Tale by Francis Glassborow

One consequence of the introduction of exception handling into C++ was the need to ensure that dynamically allocated resources were de-allocated if an exception was thrown through them. The most appropriate mechanism is encapsulating resource allocation in a class whose destructor releases them. For example:

```
class PortHandle {
  Port * handle;
public:
  PortHandle(): handle(0) {}
  // other constructors
  void grab(Port);
  void release();
  ~PortHandle(){release();}
  // other functionality including
  // assignment
};
```

This is just a skeleton to help you grasp the idea. At any stage handle either contains the address of a `Port` object (providing all the functionality for handling a port) or it is the null pointer. Anytime a `PortHandle` goes out of scope either through normal flow or because of an exception the port will be released. In other words the dynamic acquisition of a port resource will be exception safe.

One problem with this is that we naturally handle dynamic resources with pointers. Ideally we should continue to use variables with the syntax of pointers but with augmented semantics. C++ provides us with the mechanism for creating such auxiliary types. They are called smart pointers and the template mechanism allows us to create generic smart pointers that can instantiated to contain any type of raw pointer.

Let me narrow the focus to dynamic memory management. Fundamentally there are two cases where we wish to use dynamic memory for objects, polymorphic single objects and arrays of homogenous objects. Note that arrays of polymorphic type do not work unless all variations of the type have identical size and layout.

Dynamic Arrays

The use of pointers to handle dynamic arrays is a pure artefact of C's syntax/semantics. Indeed it is one of the causes of serious problems to programmers. It may have seemed elegant to K&R but I think that it is better characterised as a hack. We have no need to perpetuate this idiom in C++ (except in exceptionally low-level, sub-basement, component implementation). Unless severe efficiency constraints dictate otherwise we should be using an STL container for collections of homogeneous objects. The obvious first candidate is `vector<T>`, which behaves as closely to a raw array as desirable, but no closer.

It is the task of such components as STL containers (or user written versions that follow similar design criteria) to encapsulate `new[]` and `delete[]`. I would view any user level code that contained either of these with the gravest suspicion.

Polymorphic Singletons

Clearly there is no need to use dynamic allocation for objects whose exact type is known at compile time. You may be tempted to question that on the basis that you use a dynamic idiom to allocate large objects on the heap instead of having them consume precious stack space. The appropriate idiom for such is to use a handle or surrogate class. In other words you encapsulate the dynamic allocation in a class that outwardly behaves exactly like the real object. If you do not know how to do this go and find out how to use the ‘Cheshire Cat’ idiom.

Experienced class designers of polymorphic types manage polymorphism types through polymorphic objects that are provided by proxy or surrogate classes. However this is not always desirable, and we also have to cope with instances where the components we want to use are insufficiently finished. In addition there are many times that class designers need to handle dynamic objects precisely to remove that burden from the application programmer.

What we need is a smart pointer that will manage memory for any and all plain objects. The UK placed provision of such as a requirement on the Library some time prior to the release of CD1. At the same time we were aware that there were several aspects to the problem.

Let me look at them.

If we just want to create a dynamic object for current use that must be destroyed before we exit from the current block (return from function) then we do not even need a smart pointer. The following would seem to meet all our needs:

```
template <typename T> class Holder{
  T * handle;
  // remove copy semantics
  Holder & operator = (Holder const &);
  Holder(Holder const &);
public:
  Holder(T * tptr = 0): handle(tptr){}
  ~Holder() throw(){delete handle;}
  // allow capture of a raw pointer
  void operator = (T * tptr)
    {delete handle, handle=tptr;}
};
```

Of course you will need to be very careful about uses of the raw pointer. Indeed I might replace that last function with:

```
void operator = (T * tptr) throw(inUse)
{
  if (handle) throw inUse();
  handle=ptr;
}
```

So that a `Holder` object was never reassigned. This kind of design decision is a balancing act and it takes time and experience to get it right.

The problem with `Holder` is that it does not meet the requirements of those that want to return a dynamically created object from some form of factory function. It would have met the UK’s original requirement but those working on the issue decided that whatever solution was provided the problem of exception safe return of a dynamic object had to be catered for.

There is an entirely different idiom that meets this need and that is the use of a counted pointer. Unfortunately there are at least two problems with that. The idiom can either be implemented with a non-invasive (does not require the support of the object being pointed to) technique or by an efficient invasive one (only works for types that have been designed for use with a counted pointer). Though a couple of experts think they now have better methods of implementing non-invasive counted pointers at the time of CD1 most believed that the non-invasive technique carried too much overhead to be acceptable.

The second problem is that counted pointers allow programmers to create closed cycles that can only be destroyed by direct programmer intervention (or by some Garbage Collection techniques). That is unacceptable when the basic motive is to ensure correct resource release when an exception is thrown. For that reason work on providing a counted pointer in the standard C++ Library was suspended – we were trying to ship a standard☺

The alternative was to design some form of smart pointer that enabled ownership to be transferred. This is problematical because transfer of ownership means that the object being copied (either by cloning or by assignment) is changed by the process. The standard copy semantics is what is called 'const copy semantics' – the process of copying does not change the original.

CD1 provided an `auto_ptr` that basically used simple reference (rather than `const` qualified) parameters for the copy functions (made a little more complicated by the need to support copying between `auto_ptr`s to derived and base classes, but let us keep focused on the basics). This meant that if you `const` qualified an instance of an `auto_ptr` either directly or through use of a `const &` parameter the instance could not be copied and so would not loose ownership. Because it could not loose ownership, the raw pointer it was encapsulating was safe from destruction elsewhere.

The horror scenario of create an `auto_ptr` to hold the address of a dynamic object, pass it by `const &` to a function, copy the `const &` to a local variable, forget to return ownership to the parameter before exit was not possible. The horror is that were such a chain of action not prevented, exit from a local block would destroy the object and create a hanging pointer in the original `auto_ptr` object.

Unfortunately it was then discovered that plain (unqualified) copy semantics failed to meet the design criteria. The reason is instructive. Consider:

```
auto_ptr<PT> & factory ();
```

That is useless because the returned reference is inevitably going to be a hanging one (the local version is in the process of being destroyed as it goes out of scope). So we have:

```
auto_ptr<PT> factory();
```

That is we return by value. The ownership of the pointer to the newly created `PT` object is passed to the return value before the local instance is destroyed. So far, so good. But how are we going to capture the return value? Unfortunately the return value is exactly that, a value. The rules specify that you can only bind a value to a `const &`. In this case the two options that might be used to capture the value (that means copying) have unqualified reference parameters. In simple terms they will not work to copy a value but only to copy an unqualified object or reference. The reason that it took some time for the designers to recognise this flaw is that most compilers circa 1995 did not enforce that constraint (many still do not).

What we need is a conversion operator to technically converts a value (so called rvalue for language experts) to an object (lvalue). The language allows you to write such an operator and then specifies that it will never be called. It is probable that this was an over-constraint because at the time of writing those involved could not imagine why you might want such an operator. The easy rule is to let the programmer write it (as the rule includes conversions to base classes, it would be a nightmare to detect declarations) but specify that it shall never be called (because it was believed that compilers would never look for it as it would already have a conversion, there were also some technical problems that would surface in some circumstances)

By the time this problem was fully appreciated it was deemed too late to revisit the issue in the core of the language. It then became a judgement call as to what was the least bad solution to making `auto_ptr` meet its design requirements. In the event it was decided to return full `const` copy semantics with transfer of ownership to `auto_ptr<>`. Initially only the UK viewed that as being unacceptable.

From our viewpoint the side-affects were worse than the disease. First programmers would find that passing by `const &` did not protect their `auto_ptr`s from loss of ownership. That is pretty bad and would require rather more education of C++ users than we could expect.

What was worse is that it led programmers to use collections of `auto_ptr`. We even had one world class expert express the belief that this could be done safely. I think it was Andy Koenig who pointed out that efficient implementation of some STL algorithms would lead to invalidated `auto_ptr`s in collections. I think this was the final blow that focused a good deal of high-power intellect on solving the problem. We knew that a simple change to the core language would fix the problem and allow the CD1 version to work but at this late stage we did not have the time to explore the potential side-effects of any such change. Experience had taught us at least one thing, apparently simple changes can have unexpected results. What we needed was a pure coding solution. Finally after much refinement Bill Gibbons tied down what will either become known as the Gibbons idiom or the Gibbons error. Let me avoid the complexities of templates (and believe me they add quite a bit to the analysis of this idiom)

```
class T {
  struct Tref {
    T const & tref;
    Tref( T t):tref(t){}
  };
  // whatever else T needs
public:
  // normal T members
  // now provide non-const copying
  T (T &);
  T & operator=(T &);
  // and a special conversion operator
  operator Tref()const;
  // and copying Tref’s
  T(Tref const &);
  T & operator(Tref const &);
};
```

That `private struct` is the key. Its constructor is allowed to bind a value to the `T const &` that is its data member. Some magic (casts) are needed in implementing the last two functions but they must work because a `Tref` will only be constructed from a value. `T const &` objects can be handled directly with the non-`const` copy functions and so `Tref` will not be constructed with a value from a plain reference. The lack of a constructor from a `T const &` ensures that a `Tref` cannot be constructed from copying a `T const &`. In other words any `Tref` object must actually bind its data to an rvalue of type `T`.

I think everyone agrees that this must work and any compiler that does not manage this is bugged. Where it gets more difficult is where we have a template class that uses a member template to manage relationships between the template parameter and bases/derived types of the template parameter. All compilers tested failed to manage that case and failed in different ways. In every case the implementors declared that the failure was a bug.

Study of the development of `auto_ptr` reveals quite a lot about C++. Those of us who blazed the trail learnt quite a bit more. For example qualifying a copy constructor with `explicit` means you cannot pass such a type by value though you can create a local copy. One day I will find a use for that little gem.

Warnings

I am writing this article against a deadline (I am already a day past the copydate) and lack the time to check all the fine detail so there is likely to be more than one error but I hope that you find it interesting and useful.

One vital thing to grasp is that `auto_ptr<>` is intended to handle a very limited class of problems. It does not work in STL containers. Even if you manage to find some contortion that continues to allow you to create such containers they will fail at some time. Hopefully this latest version (unlike the CD2 version) will fail at compile time.

If you need a container of polymorphic objects you will need to implement your own smart pointer (probably a version of counted pointer) if the designer of the polymorphic hierarchy has not provided some suitable handle class. Used properly `auto_ptr` can be extremely useful, the latest version is harder to abuse but do not take that as a challenge.

Example

```
class Cat {
  myType * smile;
public:
  Cat & operator= (Cat const &);
  // rest of class definition
};

Cat & Cat::operator=(Cat const & c)
{
  auto_ptr<myType>
    temp(new myType(*c.smile));
  delete smile;
  smile=temp.release();
  return *this;
}
```

Note that we do not need a try block for this version of copy assignment to be exception safe. It is true that `~myType()` might throw but in that case you have far more serious problems; destructors should not throw exceptions. The `temp` object protects the copy of the right-hand side until it is attached to the left-hand operand. Note that this solution is apparently no more complicated than the one you would write in the absence of exception handling. The belief that EH results in programmers having to write much more code is mistaken. With EH you have to learn to use your tools correctly.

Francis Glassborow
francis@robinton.demon.co.uk

C++ Techniques

STL Vectors by Sergey Ignatchenko and Dmitry Ligum

The Standard Template Library (STL) is part of the standard C++ library. The STL is intended for the organization of data storage (STL containers) and processing (STL algorithms). Using the STL in programs with complicated data structures saves considerable development time, and makes source code more readable and easier to maintain. At the same time the STL is efficient enough, and, if properly used, causes minimal overhead.

To compile examples from this article you must include the following fragment in your program:

```
#include <vector.h>   //or <vector>
#include <algo.h>     //or <algorithm>
#include <iostream.h> //or <iostream>
// If the STL is separated into
// a the “std” namespace.
using namespace std;
```

Containers

One of the key concepts of the STL is the container. A container is a data structure, intended for storage and manipulation with objects of some other type (for example a linked list or tree). The C-style array is an example of the simplest non-STL container. Let's consider `vector` as an example STL container.

Vector's behavior is very close to the behavior of an ordinary C-style array. The main difference being that `vector` is able to increase its size as necessary. Let's consider an example of C-style array usage:

```
int init_array( int* a, int max_size )
{
  for( int n = 0; n < max_size; ++n )
  {
    int i;
    cin >> i;
    if( i == 0 ) break;
    a[ n++ ] = i;
  }
  return n;
}

const int MAX_SIZE = 64;
int a[ MAX_SIZE ];
int n = init_array( a, MAX_SIZE );
for( int j=0; j < n ; ++j )
    cout << a[ j ] << endl;
```

In this example a user fills the array with integers, then these integers are processed, and finally they are printed. The example contains a serious problem: it is unknown how many integers the user will enter. This leads to a large value of the MAX_SIZE constant being selected, which in some cases may result in extra memory usage. In similar cases it is often better to use a vector.

```
void init_vector( vector< int >& a )
{
  for(;;)
  {
    int i;
    cin >> i;
    if( i == 0 ) break;
    a.push_back( i );
  }
}

vector< int > a;
init_vector( a );
for( int j=0; j < a.size() ; ++j )
  cout << a[ j ] << endl;
```

It should be mentioned, that operator [ ] (getting element by its index) is a specific feature of the `vector` container. Other STL containers may not have this feature.

Iterator

Every STL container has its own type of iterator. An iterator is an object, which is used for enumerating the elements of the container. The iterator for C-style arrays is an ordinary pointer. Let's compare usage of pointers and iterators for enumerating the elements of a container 'a':

```
int a[ MAX_SIZE ];
int n = init_array( a, MAX_SIZE );
int* pEnd = a + n;
for( int* p =  a; p < pEnd; ++p )
  *p = rand();
for(const int* cp= a; cp < pEnd; ++cp )
  cout << *cp << endl;
```

The equivalent `vector` example being:

```
vector< int > a;
init_vector( a );
for( vector<int>::iterator p =
a.begin(); p < a.end() ; ++p )
  *p = rand();
for( vector<int>::const_iterator cp =
a.begin(); cp < a.end() ; ++cp )
  cout << *p << endl;
```

In most cases an iterator can be considered as a pointer to a container element, with one exception: for most iterators `operator->` is not defined, and the construction `(*p).f()` should be used instead of `p->f()`.

For every STL container the member function *begin*() returns an iterator pointing to the first element of the container, and member function *end*() returns an iterator pointing one past the last element in the container. Iterator returned by function *end*() always points to a non-existent element of the container. If the container '*c*' is empty, then c.begin()==c.end().

It should be mentioned that iterators pointing to an element of the container might become invalid after some modifications (see list of literature). In the case of `vector`, this can happen as a result of almost any insert or erase:

```
vector< int > a;
a.push_back( 123 );
vector< int >::iterator iter= a.begin();
int i1 = *iter; //OK
```

```
a.push_back( 456 );
int i2 = *iter; //potential error
```

To avoid such potential errors, it is possible to use indexes instead of iterators.

Vector Modification

To insert an element into a vector the member function *insert*(iterator pos, const T&x) can be used. The element is inserted at a position which is pointed to by the *pos* parameter, and all following elements of the vector are moved towards the end of the vector. Thus, v.insert (v.begin(), e) inserts an element e at the beginning, and v.insert(v,end(), e) inserts it at the end. The above mentioned construction v.push_back(e) can be considered an inline version of v.insert(v.end(), e).

The function *erase*(iterator pos) deletes an element at position *pos*, and all following elements of the vector are moved towards the beginning of the vector. It is alo possible to use the *erase*(iterator first, iterator last) member function to delete a range of elements of the vector. In this case all elements, from **first* to **(last-1)* will be deleted. To erase all elements of the vector v.erase(v.begin(), v.end()), or v.clear() could be written. Function *pop_back*() erases the last element of the vector, and this is an inline version of v.erase(v.end() - 1), but not of v.erase(v.end()).

In practice it is often necessary to insert or erase elements by its index within the container. This can be achieved by applying pointer arithmetic to vector iterators (this feature is specific to vector). Thus, to insert an element at the third position could be written as v.insert(v.begin() + 3, e).

Vector Range Check

The STL standard does not stipulate any control over correctness of indexes and iterators during use of vector. It means that an attempt to use a non-existing element may cause any result, including program crash. Thus, similarly to use of C-style array, programmer is fully responsible for correct use of vector.

The only exception is member function *at*(). This function is similar to operator [ ], but if the index is invalid, it throws an *out_of_range* exception. The authors suggest that use of function *at*() instead of operator [ ] in most cases is unsuitable: range checking is usually needed only for debugging, while the overhead caused by extra checking code will remain in release version.

Complex Objects In Containers

To be a vector element, type X must satisfy some requirements. These requirements are met automatically if type *X* is one of the following: A C numerical type (*int*, *char*, *double* etc.) or a standard C++ library class *string*, or any of the STL containers. If *X* is a custom class, the programmer must ensure that class *X* has:

default constructor X()
    generated automatically, if all data members of class *X* have default constructors, and class *X* has no constructors

copy constructor X(const X&)
    generated automatically, if all data members of class *X* have copy constructors

assignment operator operator=(const X&)
    generated automatically, if all data members of class *X* have assignment operators

Following code shows a problem that often arises in practice:

```
struct X
{
  int i;
  double d;
  string s;
  vector< int > v;
  X( int ii, double dd );
};
```

```
vector< X > vx; // Compile-time error
```

Here class *X* already has a constructor, which is why the compiler will not generate a default constructor. To correct the error, default constructor *X*() {} should be added to class *X*.

In the case of storing complex objects in containers, consideration of the object "life-time" is essential. There is strict rule: the container element "life-time" can not exceed the container "life-time", i.e. container destructor calls destructors of all its elements. Obviously, element destructor is also called if the element is deleted by erase functions.

Sorting and Searching

Let's consider another STL concept: algorithm. Algorithm is an operation over a container. Algorithms are not bound to particular containers. This "container independence" is achieved by using iterators. Among algorithms, applicable to vector (as well as to C-style array), sort/search algorithms are most often used in practice.

If a programmer wants to use the sort or search algorithms, the contained object type must provide a compare operation. For trivial types, such as int, this is done automatically, and for user-defined classes it must be done explicitly as follows:

```
class Person
{
public:
  char FirstName[32];
  char LastName[32];
  bool operator <(const Person& p) const
  {
  return strcmp(LastName,p.LastName)<0;
  }
};
```

The STL sort and search algorithm family consists of the following functions:

sort(iterator first, iterator last)
sorts range from *first to *(last-1).

lower_bound(iterator first, iterator last, const T& e)
carries out binary search at the certain, previously sorted range in container. If one or more elements equal to e found, returns iterator, pointing the first element found, else returns position where the element e can be inserted to preserve ordering.

upper_bound(iterator first, iterator last, const T& e)
same as lower_bound, but if more than one or more element equal to e found, returns iterator, pointing one past last element found.

binary_search(iterator first, iterator last, const T& e)
carries out binary search at the certain, previously sorted range in container. Returns *true*, if at least one element found. Rather seldom used in practice.

Let's consider an example of the sort and search algorithm in use:

```
vector< Person > v;
init_person_vector( v ); // fills vector with some data
sort( v.begin(), v.end() );
for(;;)
{
  Person p;
  cin >> p.LastName;
  vector< Person >::iterator il = lower_bound(v.begin(), v.end(), p);
  vector< Person >::iterator iu = upper_bound(v.begin(), v.end(), p);
  for( vector< Person >::iterator i = il ; i < iu ; ++i )
  cout << (*i).FirstName << " " << (*i).LastName << endl;
}
```

This vector of Person elements is sorted in the order defined by Person::operator< (in this particular case by LastName), and after that all Persons possessing defined LastName are searched.

It should be mentioned that the sort and search algorithms could be used for C-style arrays. For example, sort operation for a array, containing n elements, looks like sort (a, a + n).

Vector Implementation

The C++ standard does not specify, how this or that container or algorithm shall be implemented, it just limits properties of the particular container/algorithm. Nevertheless, data structures, used for implementation of containers are usually the same. This is also valid for vector.

Typical vector consists of two parts: header and data. Vector header is stored where object of type vector<T> was constructed: in static memory, on stack or in dynamic memory. Vector data is always stored in dynamic memory. This gives possibility to change vector data size dynamically. Memory allocated for vector data is usually larger than needed for vector elements.

When an element is inserted into a vector, it is constructed in reserved memory, and if reserved memory is already exhausted, the vector data is reallocated. To manage memory reservation (and reallocation process) vector member functions *capacity* and *reserve* are used. (See list of literature).

A typical vector iterator is implemented on the basis of an element pointer. Resulting from this, the following (and many other) properties become obvious:

1. pointer arithmetic can be used with vector iterators;
2. after reallocation all iterators, as well as element pointers, become invalid.

Sergey Ignatchenko ipsign@redline.ru

Dmitry Ligum ligum@rtsnet.ru

Literature:

1. STL Programmers Guide, available at www.sgi.com/Technology/STL
2. Bjarne Stroustrup "The C++ Programming Language" Third Edition.
3. David R. Musser, Atul Saini, "Stl Tutorial & Reference Guide : C++ Programming With the Standard Template Library"

Whiteboard

Rational Values Implementation Part 2 by The Harpist

This time I want to look at a few issues that arise when implementing operators. Remember that my main motive in writing this series is to explore various design issues related to a pure value based class. As I only have experience, coupled with some reading and general intelligence, to guide me I have no doubt that I will miss alternatives. All I claim is that what I am providing is a basis for discussion, as well as being better than what I find in most books.

There are several issues that are specific to implementing rationals. I consider these to be of lesser importance, though I would be delighted to discover algorithms that dealt with such problems as large denominators.

Arithmetic Operators

Built in types have two sets of arithmetic operators, those that modify their left-hand operand (assignment and compound assignment operators) and those that return a temporary value by some mechanism. Our uneducated instincts suggest that the latter group is in some way more primitive than the former. However many aspects (not least, they only deal with two 'objects' while the latter deal with three: the two operands and the return value) of the former suggest that we have to view them as the primitive operators.

So that we have something to focus on let me deal with multiplication. I hope that you know that we start by providing a class scope implementation of `operator *=`. In other words we start by declaring and implementing a function that has all the access it needs to private data.

Multiplication

If you follow the textbooks you will write a declaration something like:

```
Rational & operator *= (
  Rational const &);
```

That will certainly meet our requirements in that it will provide a mechanism whereby the left-hand operand will be modified by multiplying it by the right-hand one. But we should consider the probable needs of the application programmer. Those using specific implementations of rational numbers are very likely to be working in areas of computationally intensive programming. These people value every potential gain in computational time. They really care that we program in a way that gives the compiler the maximum opportunity for optimisation. Note that I am not saying we should optimise, merely that we should avoid inhibiting it. We should also recognise that speed is most likely to be the issue.

In that light examine the conventional choice to pass the right-hand operand by `const &`. Most writers argue that this is more efficient than passing by value. In terms of passing the operand that may well be the case. However there are other issues that you should be aware of. Passing by `const &` severely inhibits the compiler because while you may know that the passed object will not change the compiler does not. All that a `const &` parameter guarantees is that only `const` member functions will be applied to uses (direct or derived from) of that parameter in the body of the code. The compiler cannot assume that it has the only mechanism for accessing the underlying storage and so must be very careful of any cached data.

One way that the problem can arise is in multi-threaded code where more than one thread has access to the object being passed by `const &`. Another thread might attempt to change the value while your `operator *=` was using it. That would mean that you would need to use a lock during the execution of the operator.

That reminds me of another problem, the compiler generated copy constructor and assignment constructor do not provide protection against what I believe is called race conditions, in other words another thread might change the value while it was being copied. It would be useful if I could tell the compiler to compile my library for multithreading and as a result get automatic locks during compiler generated copy functions.

It seems to me that this area particularly needs attention because I am not aware of any way that I can provide a lock when using a constructor-initialiser list. Perhaps this is another place that needs to be addressed when supporting multi-threaded code. My gut reaction is that construction should always behave as if it were an atomic operation.

Perhaps some expert on this area of coding could comment.

What I am getting at is that serious consideration should be given to declaring:

```
Rational & operator *= (Rational);
```

In other words, pass by value. The compiler provided copy semantics can be implemented very efficiently and the result would allow the compiler to apply many other optimisations. The best I can say is that it would certainly be worthwhile benchmarking this alternative.

We should also consider providing some overloads to this operator because several of these can be implemented more efficiently. For example if we wanted to multiply a rational by an integer we would wish to avoid the conversion of an integer to a `Rational`. So we declare:

```
Rational & operator *= (long);
```

This works fine until we realise that it has now broken code such as the following fragment:

```
Rational r;
r *= 1.2;
```

The compiler is faced with selecting from our set of overloaded functions and will select the one taking a `long` because the built-in conversion from `double` to `long` is considered a better match than the user defined conversion via the constructor that takes a `double` as parameter. This means that once we elect to provide an overload for integer right-hand operands we must also add one for floating point ones. So we add:

```
Rational & operator *= (double);
```

When I added this to my implementation and tested it I got an unpleasant and unexpected side effect. It handled all floating-point right-hand operands correctly but issued ambiguity errors when the right-hand operand is an `int`. I have always felt that I would like more calls of ambiguity but I cannot feel that there should be ambiguity between promotion from `int` to long as opposed to conversion from `int` to `double`. Is Borland correct in this? (*Well Visual C++ gives the same error. Francis*). So it seems we must now add:

```
Rational & operator *= (int);
```

This could (and possibly should) be provided by inline forwarding:

```
Rational & operator *= (int i)
{ return (*this.operator *= (long(i)); }
```

I must confess that I had not realised that providing overloads for `double` and `long` created this problem. Even worse, when I changed `double` to `long double` I started to get ambiguity errors when the right-hand operand was a `double`. I hope this is just an implementation problem because otherwise it makes writing overloads for arithmetic types a nightmare. In the above we have had to add overloads to correct a wrong choice by the compiler (I am not complaining because I think the reasons are sane and acceptable) with the result that I am getting silly ambiguity calls. Conversions to higher capacity/precision types must be better than inter-conversion between integer and floating-point types.

Please note that the use of `inline` definitions to provide correct overload resolution must be correct. If you do this from the start there is no chance that its existence (of an `inline` definition) could force large scale re-compilation of code at some later stage. Only a decision to replace the forwarding process by an alternative process would cause this. However a late decision to replace a normal definition with an `inline` one would cause precisely what we wish to avoid – massive rebuilding of a substantial product. Despite Alan Griffiths' comments in Overload 22, this kind of design decision needs to be made early to provide exactly the kind of stability he seeks.

Now you understand the issues concerning how we should provide `operator *=` for our `Rational` class let us move on to the more commonly used multiplication operation. But just before we do please note that the provision of compound assignment operators such as `*=` is not just for completeness, they are extremely valuable when you want to manipulate larger scale mathematical constructs such as matrices.

I am beginning to see more writers recognise that the provision of compound assignment operators means that they do not need to abuse friendship to provide efficient implementations of the plain arithmetic operators. Remember that the problem here is to assure that the use of a built-in arithmetic type as the first operand will work correctly in the context of a user-defined second operand. For example, we not only want `<Rational> * <Rational>` and `<Rational> * <int>` but also `<int> * <Rational>`. That means that we will need at least one global declaration. The textbook standard for such a function is something like:

```
inline Rational operator *(
    Rational const& lhs,
    Rational const& rhs){
  Rational temp = lhs;
  return temp *= rhs;
}
```

I guess that the inlining is a little more debatable in this case, however a good compiler should be able to make it very efficient and, I believe, in the context should be given the latitude to do so. Inefficient compilers might generate fatter code than a simple function call but I would not want to use such a compiler for numerical work. Providing value types is normally a low-level abstraction where the designer should consider efficiency constraints as well as ease of use. It is at this level that we should be willing to write a lot so that we cover all reasonable expectations.

The next decision that needs to be taken is whether we should replace the `const &` parameters with value ones. This decision has been made more complicated by the London decision of the C++ Standards Committees that the compiler shall not have the right to optimise away copy construction required for a value parameter. However if we stay with our decision to allow the compiler to generate this constructor I believe that it can still minimise the overhead. I think that good compilers should produce a pretty lean implementation of:

```
inline Rational operator * (
    Rational lhs, Rational rhs)
{ return lhs *= rhs; }
```

Writing this focused my attention on the traditional idiom, which is clearly wrong. It always was, but the London decision makes it even more so. Regardless of any decision about the second parameter the first one clearly should be passed by value as the first thing the traditional idiom does is to copy it. So in cases where we are not concerned with maximising the compilers freedom to optimise we should write:

```
inline Rational operator * (
    Rational lhs, Rational const & rhs)
{ return lhs *= rhs; }
```

If we wish to provide the special case optimisations (as we did for `operator *=`) we will have to write two global functions per case, one for each operand that might be an operand that we wish to specialise. For example if we want to handle multiplication by an `int` we need:

```
inline Rational operator * (
    Rational lhs, int rhs)
{ return lhs *= (long)rhs; }

inline Rational operator * (
    long lhs, Rational rhs)
{ return rhs *= (long)lhs; }
```

The cast is to ensure a direct call to the general case that we implemented in class.

I hope the above discussion of exactly which prototypes should be considered proves useful to you. I am sure your editor would be happy to receive alternative views.

Implementation Considerations

I want to conclude by drawing your attention to some low-level aspects of implementing arithmetic operations for Rationals.

One serious problem is that of keeping the vales of the numerator and denominator small. Even if we are using a BigInt type we should recognise that large values will take longer to manage than small ones. As we always reduce our Rational numbers to a canonical form where the numerator and denominator are co-prime we have a couple of possibilities to consider. For example:

```
Rational & Rational::operator *= (
    Rational r)
{
  // create temporaries with
  // denominators exchanged
  Rational t1(numerator, r.denominator);
  Rational t2(r.numerator, denominator);
  t1.simplify(), t2.simplify;
  // at this stage all common factors
  // have been eliminated
  numerator =
    t1.numerator * t2.numerator;
  denominator =
    t1.denominator * t2.denominator;
  // floating-point cache invalidated
  converted = false;
  return *this;
}
```

The effective need to create two temporary `Rational`s just to call simplify suggest we should revisit this aspect of the design. If you are using the modern style where you encapsulate a class and all its helpers into a namespace then providing a utility function to reduce two numbers by eliminating common factors would be more efficient. At the same time the `typedef` providing the `integer_type` should also be moved out to `namespace` scope. An alternative, more traditional, approach would be to provide the `simplify` functionality as a `class static` function taking two parameters of type `integer_type`. My personal preference is to use `namespace`s as they provide good encapsulation of the functionality of a type (which, as here, is often far more than just the class definition).

Implementing the specialisation for the right-hand operand as a `long` reveals why this specialisation is desirable:

```
Rational & Rational::operator *=(
    long rhs)
{
  // keep the current numerator
  integer_type temp = numerator;
  numerator = rhs;
  simplify();
  numerator *= temp;
  converted = false;
  return *this;
}
```

Note that this code assumes that `integer_type` supports compound assignments. It is because programmers make such assumptions that designers of high-capacity/precision arithmetic types should provide a full range of arithmetic operators. I have no doubt that failure to do so is a serious design flaw.

Implementing the specialisations for floating-point types (forced on us because of the overloading rules) should probably by forwarding to the standard form. You are going to have to do most of the work to convert the operand to a `Rational` and so little would be gained by trying to avoid completing the task. There is one other option that you might consider, particularly as this type may well be used in matrices. That is to make these specialisations `private` and thereby force the user to do the conversion explicitly. This gains little in single instances but could gain a lot when doing matrix arithmetic on matrices of `Rational`s.

Providing division operators for `Rational`s is an interesting exercise in using forwarding functions and specialistaions. For example:

```
Rational & Rational::operator / (Rational r)
{
  if (r.numerator == 0) throw DivideByZero;
  Rational temp(r.denominator, r.numerator);
  converted = false;
  return ( operator*=(temp) );
}

Rational & Rational Rational::operator / (long rhs)
```

```
{
  integer_type temp = denominator;
  denominator = rhs;
  simplify();
  denominator *= temp;
  converted = false;
  return *this;
}
```

The basic rule all the time is to delay multiplication as long as possible because that is the operation that is most likely to cause overflow.

I think that about covers the ground. What do you think?

Something Else

The design allows for some special values to be held for those working in areas where signed infinities and signed zeroes are useful. It is trivial to allow signed zeroes because the sign bit is being held independently. More interesting is that using a zero `denominator` can represent infinity and the use of zero for both `numerator` and `denominator` can represent and indeterminate value. In some types of mathematical work these features can be useful. Very little extra work is required to support them.

The Harpist

A Finite State Machine Design II by Einar Nilsen-Nygaard

In the last article I'd got as far as presenting the initial design and some class declarations for a finite state machine (FSM) design. The implementation is being done using the STL.

Recap

To recap the design, I'd ended up with two classes and an interface:

- *StateMachine* - the main controlling class.
- *State* - a class encapsulating the value associated with a state and any associated actions.
- *ActionInterface* - an interface for specifying actions to be carried out.

All three of the above are parameterised by (note the facilities that these classes must provide):

- *StateValue* - the class representing the value of states. This class must provide `operator==` and `operator<`.
- *Stimulus* - the class representing the type used to trigger state transitions. This class must provide `operator==` and `operator<`.

The aim of this design is to allow users of the classes to pick them up and easily assemble a FSM suitable to their needs, with the only coding necessary being that related to implementing actions and complex *StateValue* and *Stimulus* classes.

Implementation

Last time I presented the interfaces to the classes that came out of the design. Now I'll put down a first-cut implementation for these classes. Let's look at *StateMachine* first.

sm.cc

The constructor, which is used to initialise the starting state of the FSM.

```
template<class StateValue,class Stim>
```

```
StateMachine<StateValue,Stim>::StateMachine(const StateValue &initialState)
: currentStateValue(initialState)
{ }
```

A default destructor that does nothing useful for now.

```
template<class StateValue,class Stim>
StateMachine<StateValue,Stim>::~StateMachine()
{ }
```

The following two methods implement adding and removing states from the FSM. This is done by value. States are stored in a STL map container, accessed by the value of the state they represent.

```
template<class StateValue,class Stim>
bool StateMachine<StateValue,Stim>::addState(const State<StateValue,Stim> &state)
{
    stateMap[state.value()] = state;
    return true;
}

template<class StateValue,class Stim>
bool StateMachine<StateValue,Stim>::removeState(const State<StateValue,Stim> &state)
{
    StateContainer::iterator s = stateMap.find(state.value());
    if( s!=stateMap.end() )
    {
        stateMap.erase(s);
        return true;
    }
    return false;
}
```

The following two methods form the top-level interface for adding and removing actions. These are added by reference, allowing multiple states and FSMs to reuse action objects. Note that the underlying State object is what actually holds the reference to the action object.

```
template<class StateValue,class Stim>
bool StateMachine<StateValue,Stim>::attachAction
(const StateValue &sv,ActionTime at,ActionInterface<StateValue,Stim> *ai)
{
    StateContainer::iterator s = stateMap.find(sv);
    if( s!=stateMap.end() )
    {
        return (*s).second.attachAction(at,ai);
    }
    return false;
}

template<class StateValue,class Stim>
bool StateMachine<StateValue,Stim>::detachAction
(const StateValue &sv,ActionTime at,ActionInterface<StateValue,Stim> *ai)
{
    StateContainer::iterator s = stateMap.find(sv);
    if( s!=stateMap.end() )
    {
        return (*s).second.detachAction(at,ai);
    }
    return false;
}
```

This is how we get the FSM to change state. The return value indicates whether or not the stimulation was successful or not. A false return value means that the current state did not have a valid transition for the given stimulus.

```
template<class StateValue,class Stim>
bool StateMachine<StateValue,Stim>::stimulate(const Stim &stim)
{
    StateContainer::iterator currStatePair = stateMap.find(currentStateValue);

    if( currStatePair!=stateMap.end() )
    {
        StateValue nextStateValue;
        if( (*currStatePair).second.getNextStateValue(stim,nextStateValue) )
        {
            if( nextStateValue!=currentStateValue )
            {
                StateContainer::iterator nextStatePair
                    = stateMap.find(nextStateValue);
                if( nextStatePair!=stateMap.end() )
                {
                    (*currStatePair).second.leave(this);
                    currentStateValue = nextStateValue;
                    (*nextStatePair).second.enter(this);
                    return true;
                }
            }
        }
    }
    return false;
}
```

This simple accessor allows us to access the current state value of the FSM.

```
template<class StateValue,class Stim>
const StateValue &StateMachine<StateValue,Stim>::getCurrentStateValue()
{
    return currentStateValue;
}
```

As it turns out, *StateMachine* turns out to be a very simple class. Its main responsibility is to serve as a repository for the states and to indicate to the states that they are being left and entered. The two most important aspects of the FSM are delegated to the *State* class - the triggering and management of actions and knowledge of valid state transitions. In fact, *StateMachine* has no direct knowledge of the transitions at all!

Moving on to the *State* class, we can see how actions are stored and triggered and how transitions are managed.

state.cc

The State class has a simple constructor. It only initialises the value represented by this state.

```
template<class StateValue,class Stim>
State<StateValue,Stim>::State(const StateValue &sval)
: sval_(sval)
{ }
```

The copy constructor ensures the state value, any referenced actions and the state transition map are copied properly.

```
template<class StateValue,class Stim>
State<StateValue,Stim>::State(const State &pattern)
: sval_(pattern.sval_),
  before_(pattern.before_),
  during_(pattern.during_),
  after_(pattern.after_),
  tmap_(pattern.tmap_)
```

```
{ }
```

States have a simple destructor that does nothing for now.

```
template<class StateValue,class Stim>
State<StateValue,Stim>::~State()
{ }
```

The assignment operator is necessary for correct operation with STL containers.

```
template<class StateValue,class Stim>
State<StateValue,Stim> &
State<StateValue,Stim>::operator=(const State<StateValue,Stim> &pattern)
{
    if( this!=&pattern )
    {
        sval_ = pattern.sval_;
        before_ = pattern.before_;
        during_ = pattern.during_;
        after_ = pattern.after_;
        tmap_ = pattern.tmap_;
    }
    return *this;
}
```

Simple accessor for the state value represented by this object.

```
template<class StateValue,class Stim>
const StateValue &
State<StateValue,Stim>::value() const
{
    return sval_;
}
```

This is how we add knowledge of next states to the FSM. The transitions must be set up prior to adding the states to the StateMachine instance itself. This is a potential flaw!!! (See later.)

```
template<class StateValue,class Stim>
bool
State<StateValue,Stim>::addTransition( const Stim &stim,const StateValue &nextSval)
{
    TransitionMap::value_type t(stim,nextSval);
    pair<TransitionMap::iterator,bool> retval = tmap_.insert(t);

    return retval.second;
}
```

This method allows the containing StateMachine class to find out what the value of the next state should be in response to a particular input stimulus. If a state has no next state for the given stimulus then false is returned to the client.

```
template<class StateValue,class Stim>
bool State<StateValue,Stim>::getNextStateValue
(const Stim &stim,StateValue &nextSval)
{
    TransitionMap::iterator n = tmap_.find(stim);
    if( n!=tmap_.end() )
    {
        nextSval = (*n).second;
        return true;
    }
    return false;
}
```

The next two methods handle adding and removing actions to be executed before a state is entered, while a state is active or just before state is left. Note that the actions are stored by reference. The bulk of the methods are spent identifying the correct internal collection to work on (one of before, during and after).

```
template<class StateValue,class Stim>
bool
State<StateValue,Stim>::attachAction( ActionTime at,
                                      ActionInterface<StateValue,Stim> *ai )
{
    switch( at )
    {
    case Before: before_.insert(ai); break;
    case During: during_.insert(ai); break;
    case After:  after_.insert(ai);  break;
    default:     return false;       break;
    }
    return true;
}

template<class StateValue,class Stim>
bool State<StateValue,Stim>::detachAction( ActionTime at,
                                      ActionInterface<StateValue,Stim> *ai )
{
    bool retval = false;
    ActionContainer *actionSet = 0;

    switch( at )
    {
    case Before: actionSet = &before_; break;
    case During: actionSet = &during_; break;
    case After:  actionSet = &after_;  break;
    default:     return false;         break;
    }
    ActionContainer::iterator toErase = actionSet->find(ai);
    if( toErase!=actionSet->end() )
    {
        actionSet->erase(toErase);
        return true;
    }
    else
    {
        return false;
    }
}
```

When a state is entered we must execute all the appropriate actions - call all the “before” actions followed by all the “during” actions.

```
template<class StateValue,class Stim>
void State<StateValue,Stim>::enter(StateMachine<StateValue,Stim> *sm)
{
    ActionContainer::iterator iter1(before_.begin());
    while( iter1!=before_.end() )
    {
        (*iter1)->start(sm);
        iter1++;
    }
    ActionContainer::iterator iter2(during_.begin());
    while( iter2!=during_.end() )
    {
        (*iter2)->start(sm);
        iter2++;
    }
}
```

When we leave a state we should stop all the “during” actions and call all the “after” actions.

```
template<class StateValue,class Stim>
void State<StateValue,Stim>::leave(StateMachine<StateValue,Stim> *sm)
{
    ActionContainer::iterator iter1(during_.begin());
    while( iter1!=during_.end() )
    {
        (*iter1)->stop(sm);
        iter1++;
    }
    ActionContainer::iterator iter2(after_.begin());
    while( iter2!=after_.end() )
    {
        (*iter2)->start(sm);
        iter2++;
    }
}
```

Using The Classes

I’ve always found the most useful way of presenting a new piece of code to anyone is to work through an example. This also gives a chance for a measure of peer review and an opportunity for comments and questions that may result in overall improvements to your original ideas - both the design and implementation aspects.

So, what I’ll present is something I’m familiar with in my day-to-day work - a simplified state machine representing a FSM used to control a generic xDSL (Digital Subscriber Loop) line card. Briefly, these cards are an emerging technology starting to be used to provide high bandwidth connections to the Internet.

For management purposes we’ll assume the card has the following “states” associated with its operation:

- Decommission - the card is not currently operating and may not be used.
- Normal - the card is operating normally with no problems.
- Warning, Minor, Major & Critical - a range of severities, from a potential problem to service affecting fault.
- Downloading - the card is involved in downloading a new copy of its own software.

Next, we need to have some inputs to the system. We can list these in a table showing the input, current state and next state. Any stimulus/state pair not shown in this table will be deemed invalid. Note that the state model is much simplified.

Stimulus	**Current State(s)**	**Next State**
decomm	<any state>	Decommission
comm	Decommission	Normal
warn	Normal	Warning
minor	Normal | Warning	Minor
major	Normal | Warning | Minor	Major
crit	Normal | Warning | Minor | Major	Critical
startdload	Normal	Download
enddload	Download	Normal
clear	Warning | Minor | Major |	Normal

	Critical	

Table 1 xDSL Card State Transitions

To simplify the example I'll use strings to represent all states and stimuli. Going on from the table above we can declare the state machine itself as:

```
StateMachine<std::string,std::string>
      xDSLfsm("Decommission");
```

This assumes that any new card will start in the decommissioned state. Most likely we would in reality query the card and initialise the state machine appropriately. Next we can declare the states themselves:

```
typedef State<std::string,std::string>
      xDSLstate;

xDSLstate decommission("Decommission");
xDSLstate normal("Normal");
xDSLstate minor("Minor");
xDSLstate major("Major");
xDSLstate warning("Warning");
xDSLstate critical("Critical");
xDSLstate download("Download");
```

So, now we have the state machine itself and the states we wish to model. The final stage before we can add states to the state machine is to define the transitions, which we do as follows:

```
decommission.addTransition(
    "commission",     // the stimulus
    normal.value() ); // the next state
```

This can be repeated tediously until all the transitions listed in the table are specified, so I won't list them all here!

Looking at the last few lines of code I see the first potential improvement - I have currently defined the `addTransition` interface to take the value represented by the state. Perhaps it should take an actual state instead? This would have the benefit of ensuring there was actually a valid state object created at some point for the transition.

Finally, we can add the states to the state machine itself:

```
xDSLfsm.addState(decommission);
xDSLfsm.addState(normal);
```

```
xDSLfsm.addState(warning);
xDSLfsm.addState(minor);
xDSLfsm.addState(major);
xDSLfsm.addState(critical);
xDSLfsm.addState(download);
```

At this stage the state machine is now ready to accept input from external sources. However, it won't actually do anything useful in it's current condition. To remedy this we're going to have to add some actions to the state machine via classes derived from *ActionInterface*, but I'll leave that until next time!

[The base source code for this article can be picked up from: `http://www.rhuagh.demon.co.uk/fsm-code/`. *This consists of the files sm.h, sm.cc, state.h, state.cc and actionif.h.]*

Einar Nilsen-Nygaard
EinarNN@atl.co.uk
einar@rhuagh.demon.co.uk

Debug new and delete Preamble by Peter A. Pilgrim

About a year I wrote a few articles for CVu [1] [2] on how to check the integrity of dynamic memory allocation and de-allocation in C. I supplied some cut down code examples of a debuggable module which I was actually using in a real-world practical development. I elaborated on what most C users knew well already; the special problems that are encountered when they use `malloc` and `free`. There can be memory leakage, corrupted memory blocks, underrun of a memory range, overrun of an array range, double frees, and pointers to memory that is not properly initialised. How do you check the integrity of the dynamic memory, and how can you track it? There were a number of solutions presented, for ACCU members to use in C in CVu 8.5 and CVu 8.6 from contributors including myself. However, I have not seen a C++ solution appear in either CVu or Overload, but they surely exist, right? (I know somebody sent a

C++ solution to Francis, but that person did not sign it!)

Replacing Malloc and Free

In C one can define macros that very cleanly override the standard memory management functions, because they can be emulated. For example:

```
#ifdef DEBUG_MALLOC
#define DBG_MALLOC(nbytes) \
     dbg_malloc(ptr)
#else
#define DBG_MALLOC(nbytes)
     malloc(nbytes)
#endif
```

Replacing New and Delete

Things aren't quite so simple in C++ because most memory allocation is performed through the global `new` and `delete` operators. However, a mechanism is provided for overloading them with your own implementation [3]. Once you replace the global `new` and `delete` operators and link them in they will be used by whatever needs free store. This flexibility allows us to write debuggable versions of these operators. However, you must implement them according to convention, and quite carefully. They may be called at any point in your program where heap space needs to be allocated or deallocated.

The signatures for the global new and delete are defined as:

```
void *operator new(size_t);
void operator delete(void*);

void *operator new[](size_t);
void operator delete[](void*);
```

The function `operator new()` is called to allocate a suitable number of bytes for an object. The function `operator new[]()` is called to allocate space for an array of objects. It is important to note that the standard implementations of `operator new()` and `operator new[]()` do not initialise the memory returned. However, we will use this fact to our advantage later on.

In the most recent draft of the C++ Standard, an exception is thrown when `new` can find no store to allocate. This exception is an object of type `bad_alloc`. You may only find this implemented in the very latest versions of your C++ libraries.

```
#include <iostream>
#include <new>
#include <exceptions>
using namespace std;

void blowupFreeStore() {
  try {
    for (;;) new char[10240];
  }
  catch (bad_alloc) {
    cerr << "No more core!"
  }
}
```

In older C++ environments, you must always test the returned pointer from `new` with the `null` pointer in order check if your allocation was successful [3].

Even with the more recent C++ environments from a couple of years ago, you could always configure what `new` should do if the free store is exhausted. Whenever new fails, it first calls a function specified by the call to `set_new_handler()` [6]. This is an important hook that actually allows library writers to implement a C++ garbage collector, which is beyond the scope of this series of articles.

So, we have for example:

```
void no_more_core()
{
  cerr << "new failure: no more core!"
  throw bad_alloc();
}

int main()
{
  set_new_handler(no_more_core);
  for (;;) new char [10240];
  cout << "this never gets printed\n";
}
```

In the very latest committee draft (CD2), available on the web-site at Warwick University [7], defines `new` and `delete` as follows:

```
namespace std {
  class bad_alloc;

  struct nothrow_t {};

  extern const nothrow_t nothrow;
  typedef void (*new_handler)();
  new_handler
    set_new_handler(new_handler new_p)
    throw();
}
```

The above defines the exception class `bad_alloc` and a new type called `nothrow_t`. CD2 defines how a standard C++ conforming environment can support the traditional global `new` and `delete` operators which return a null pointer if no more free store can be found. The draft defines these alternative definitions:

```
void * operator new(size_t size,
     const nothrow_t& ) throw ();
void operator delete(void *ptr,
     const nothrow_t& ) throw ();

void * operator new[](size_t size,
     const nothrow_t& ) throw ();
void operator delete[](void *ptr,
     const nothrow_t& ) throw ();
```

Notice that these definitions do not throw any exceptions at all. Also, if no free store can be found, then the function specified by the normal `set_new_handler` is not called, but a null pointer is returned. Since the CD2 is so new (as I am writing on 18th November 1997) my GNU C++ Compiler 2.7.2 does not support the alternative none throwable global operators, therefore I could not test these new features.

With this information we can begin to think about how we might write debuggable `new` and `delete` operators. Your C++ compiler's run time stub function, which is executed by the operating system when `main()` is about be called, may have to construct global variables before `main()` is called, and possibly destroy any variables after `main()` returns. This means that the global `new` and `delete` operators may be invoked even before `main()` is called (and afterwards if `main()` returns normally).

What this means is that any debuggable heap allocation must be careful not to get itself into a dangerous recursive loop. It is clearly possible to write a global `new` operator that calls itself again and again, either directly or indirectly, for example by using a STL collection class.

Badness To Detect

We would like to detect a number of pathological cases of bad dynamic pointer use. The first case being trying to free an undefined pointer which was not returned by new().

```
void Case1() {
  int *p;    // `p' is anything!
  delete p;
}
```

The second would be the typical freeing a pointer to a object which has already been freed.

```
void Case2() {
  char *s = new char [256];
  delete [] s;
  …
  delete [] s;      // Error
}
```

The third is the case where a pointer dereferences an element E of a dynamic array object A lower than the lowest bound of the array A. There should be way of detecting the corruption of the lower boundary.

```
void Case3() {
  int j;
  float *x = new float[10];
  for (j=-2; j<2; ++j) x[j] = j;
  delete [] x;
}
```

The fourth is the case where a pointer dereferences an element E of a dynamic array object A higher than the upper bound of the array A.

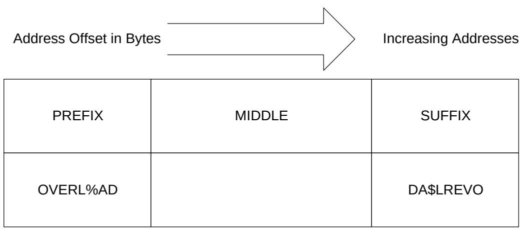
```
void Case4() {
  int j;
  float *x = new float[10];
  for (j=3; j<12; ++j) x[j] = j;
  delete [] x;
}
```

The fifth is the case where memory is leaked: free store that is allocated, but is not deallocation by the time the program or thread terminates.

```
Void Case5() {
  char *v = new char [128];
  strcpy(v, "peterp@xenonsoft");
  // Memory leak for `v'!
}
```

How to Detect Badness

The basic idea of heap space integrity is to use some special form of identification within the memory block itself. The algorithm allocates a block of heap memory bigger than the user requested, and divides this memory block, into three parts: the prefix, the middle, and the suffix. Some magic identifier bytes are written into both the prefix and the suffix, and a pointer to the middle returned to the user (appropriately recast to `void *`). I have sketched below in a diagram showing the memory block divided into three sections.



The prefix of the memory block is identified with the eight byte string 'OVERL%AD'. The string is reversed and a character changed to make up a different identifier for the suffix 'DA$LREVO'.

With these identifiers we can:

Case 1. Detect if a memory block is valid, that is, if it was allocated by operator `new`.

Case 3. Check if the lower bound of the block was corrupted by comparing the prefix part with 'OVERL%AD'

Case 4. Check if the upper bound of the block was corrupted by checking the suffix part with 'DA$LREVO'.

The remaining two cases; double-free and memory leakage will be considered in the next article.

Enjoy.

Peter Andrew Pilgrim
peterp@xenonsoft.demon.co.uk

1. "Dynamic Memory Integrity", ACCU/ CVu 8.5, Peter A. Pilgrim

2. "Dynamic Memory Tracking", ACCU/ CVu 8.6, Peter A. Pilgrim

3. "Advanced C++ Programming Styles and Idioms" by James O. Coplien, publ:Addison Wesley; Chapter 3.6: New and Delete.

4. "C++ Primer" by Stanley B. Lippman, 2nd Edition, publ.: Addison Wesley; The `new' operator pg. 114.

5. "Effective C++" by Scott Meyers, Publ.: 1992; Section 8: Adhere to convention when writing `new',

6. "The C++ Programming Language" by Bjarne Stroustrup, 3rd Edition, Publ: Addison & Wesley; Section 6.2.6: Free Store, Section 10.4.11: Placement of Objects, Section 14.4.4: Exceptions and New and Section 19.4: Allocators.

7. The recently published Public Preview of ISO C++ Committee Draft Version Two <http://www.maths.warwick.ac.uk/cpp/pub> at the University of Warwick.

editor << letters;

Managing Inline with macros

From Tony Houghton

I'm a new subscriber to Overload and I just read about the problems with inlining. I thought you might like to hear about an approach I took a while ago. I decided that the disadvantages of inlining during development were too great, but the advantages in a final product might be worthwhile; so I decided to make it possible to turn inlining on and off with a couple of macros predefined in the makefile. It does result in a little more typing of the source, but that shouldn't cause a sensible programmer to lose any sleep.

Take a simple class with functions we might want to inline:

```
class DumbClass
{
public:
  DumbClass(int a = 0)  { value = a; }
  void set_value(int a) { value = a; }
  int get_value() const { return value;}
private:
  int value;
};
```

This could be rewritten as:

```
// DumbClass.h

#ifndef __DumbClass_h
#define __DumbClass_h

class DumbClass
{
public:
  DumbClass(int a = 0);
  void set_value(int a);
  int get_value() const;
private:
  int value;
};

#ifndef DONT_INLINE
#include "DumbClass.cc"
#endif

#endif
```

```
// DumbClass.cc
```

```
#ifndef __DumbClass_h
#include "DumbClass.h"
#endif

INLINE DumbClass::DumbClass(int a)
{ value = a; }

INLINE void DumbClass::set_value(int a)
{ value = a; }

INLINE int DumbClass::get_value() const
{ return value; }
```

Then if you want inlining you use these compiler options: -DINLINE=inline And if you don't want inlining: -DINLINE -DDONT_INLINE

Tony Houghton
tonyh@tcp.co.uk

Beyond ACCU... C++ on the 'net

Introduction

Welcome to a revamp of 'ACCU and the 'Net'. Each issue we'll be presenting and reviewing online resources for C++ Designers and Programmers.

Enough rope to shoot yourself in the foot...

For an interesting site dealing with C++, OO and Java, have a look at Allen Holub's web site.

Holub	www.holub.com

Guru of the week...

If you're looking for C++ debates, visit Guru of the Week... I've only seen one of its 25 solutions grilled on ACCU.general so it must be getting something right. So far, it has dealt with many things including temporary objects, the Standard Library, class mechanics, overriding virtual functions, const-correctness, memory management, exception safety, OOP, and class relationships.

GOTW	www.cntc.com/resources/gotw.html

Standard C++.

Although the Final Draft International Standard (FDIS) isn't online yet, the 1997 C++ Public Review Corrective Draft 2 (CD2) document is available online from www.maths.warwick.ac.uk. CD2 is presented much like the reference manual section of Bjarne Stroustrup's "The C++ Programming Language" (2e). CD2 builds on that work and P.J. Plauger's book, "The Draft Standard C++ Library".

Here's a brief overview of CD2, to whet your appetite:

Core Language: Lexical conventions, Basic concepts, Standard conversions, Expressions, Statements, Declarations, Declarators, Classes, Derived classes, Member access control, Special member functions, Overloading, Templates, Exception handling, Preprocessing directives.

Library: Introduction, Language support, Diagnostics, General utilities, Strings, Localization, Containers, Iterators, Algorithms, Numerics, Input/output.

Annexes: Grammar summary, Implementation quantities, Compatibility, Future directions, Universal character-names.

Those references are fine but they need to be backed up by suitable tutorial material. My main C++ tutorial has been "The C++ Programming Language" (2e). In 1997 though, it has been complemented by "Effective C++", "More Effective C++" (by Scott Meyers) and "C++ Programming style" (Tom Cargill). Recently, I've had a look at "The C++ Programming Language" 3e, "Advanced C++ Programming" (James Coplien). Together, these books have answered some questions I had about C++. Curiously, they're all books published by Addison-Wesley.

FDIS	www.maths.warwick.ac.uk/c++
CD2	www.maths.warwick.ac.uk/c++/pub/wp/html/cd2/index.html
The Draft Standard C++ Library	*www.dinkumware.com/htm_cpl/index.html*
The C++ Programming Language	www.awl.com/cp/stroustrup3e

Addison-Wesley	*www.awl.com/cseng)*

ACCU contact details.

See Overload Issue 22.

Next issue... C++ libraries

Next month the STL and other C++ libraries will be looked at. This column already owes a lot to the suggestions made on ACCU.general (thanks).... so please post hints, links and opinions re: C++ libraries on ACCU.general.

Ian Bruntlett
Ibruntlett@libris.co.uk

Credits

Editor

John Merrells
merrells@netscape.com

4 Park Mount,
Harpenden, Herts, AL5 3AR,
U.K.

P.O. Box 2336,
Sunnyvale, CA 94087-0336,
U.S.A.

Readers

Ray Hall
Ray@ashworth.demon.co.uk

Ian Bruntlett
ibruntlett@libris.co.uk

Einar Nilsen-Nygaard
EinarNN@atl.co.uk
einar@rhuagh.demon.co.uk

Production Editor

Alan Lenton
alan@ibgames.com

Advertising

John Washington
accuads@wash.demon.co.uk
Cartchers Farm, Carthouse Lane
Woking, Surrey, GU21 4XS

Membership and Subscription Enquiries

David Hodge
davidhodge@compuserve.com
31 Egerton Road
Bexhill-on-Sea, East Sussex. TN39 3HJ

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 23* should be submitted to the editor, John Merrells < merrells@netscape.com>, by Janurary 15th.