ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 24

February 1998

Contents

Editorial

Managing Complexity

As Ada is to Pascal, as C++ is to C; these large scale languages are for managing the complexity of building large scale projects. You can implement anything in C that you can implement with C++, but C++ gives you a hand in many ways. It imposes conventions for naming, defining state, execution context, scope management, object lifetime, encapsulation, and interface definition. All features that lead to software which can be easily maintained, documented, and extended.

When faced with an aging mass (or is that a mess) of C, which must be maintained and enhanced, there's little choice but to step back and invest some time in reorganisation. A software system with poor structure breeds poor structure. Engineers tend to follow the style of the code they're working with. When there are few functions that can be reused, all new code will be a cut, paste and modification of something else. I've witnessed one project that pushed hard for a year without any restructuring of the previous version. The product shipped, late, and the next year was spent maintaining the bugs of the shipped binaries.

So, in order to repair the 'badness' I see I'm adopting an iterative rewriting approach, to apply the aspects of 'goodness' that I listed above.

Naming Conventions: Often, with many engineers on a project, a common naming scheme will be defined, and slowly diverged from. Something of the form <module>_<action> is common. I'm trying to move towards <structure>_<method>. Similar to <class>::<method>, huh? The simple benefit this provides is to help engineers find the code they need quickly. The secondary benefit is a change in state of mind. The function is a method on an object of well defined type.

Well Defined State: For each structure I create a <structure>_new and <structure>_delete function, and replace the splattering of calls to malloc and free. This single point of construction ensures that every instantiation will be initialised correctly and consistently. I assess how often each structure member is referenced, and if manageable, impose accessor functions. Often these can be upgraded to include code to maintain the well defined state of the object, and to provide some higher order of functionality. This benefits, and simplifies its callers.

Execution Context: Renaming each function within this new scheme forces each function to be identified with some structure. The first parameter then becomes a pointer to an instance of that structure (*this*).

Encapsulation: When a function can't be readily assigned to some structure it's usually because it embodies a number of intertwined concepts. It may have private knowledge of a number of structures, and have access to static level data. These kitchen sink functions can rarely be reused so must be carefully teased into composite parts. Moving the functional details of each structure into accessor functions simplifies the function and clarifies where its functionality belongs.

Well Defined Interfaces: I've been ensuring that internal functions are declared static (*private*), and that global ones are published in header files (*public*). I recently found a module which imported some functions from another with a cunning cut and paste of the function prototypes (*friend*). Definitely bad practice.

Motivation

I'm not imposing an object oriented design onto this software just because that's the way I like things. That's the way I see things, so that's the way I like them. The motivation is to reduce maintenance costs, the lead time of new engineers, and to ease the implementation of new features. With poorly encapsulated design an engineer must understand the entirety of the system before any useful work can be achieved. These few aspects of an object oriented approach to software development that I've described above should help move us towards this goal. Ultimately I hope that new features will be designed as components, making use of the newly repackaged and finally reusable existing code.

Copy Deadline

All articles intended for publication in *Overload 25* should be submitted to the editor, by March 11th.

John Merrells
merrells@netscape.com

Software Development in C++

UML – State-Transition Diagrams by Richard Blundell

Introduction

In Overload 22 and 23 we looked at some ways to document and communicate the static behaviour of a system using static structure diagrams – class diagrams and object diagrams. These types of diagram are extremely useful for many areas of system design and documentation. Sometimes, however, some of the dynamic behaviour of the system needs to be considered. State diagrams show how a system or single object of a given class behaves in response to various events and messages, and are one of the types of diagrams that the UML supports for showing dynamic behaviour. The format of these diagrams in the UML is very similar to that used in the OMT and Booch methods, and so some of what follows may be familiar to a number of you.

States, Events and Transitions

You probably all understand what is meant by a *state*. The idea is that an object, or system, or whatever we are modelling, will stay in its current state for some finite amount of time unless something happens – an *event*. In figure 1 we have a state called State. The rounded corners of the rectangle distinguish a state from the class symbols we saw in previous articles.

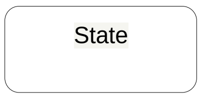

Figure 1 – A state called “State” in UML notation.

A transition can occur (or *fire*) from one state (the *source state*) to another (the *target state* – called OtherState in figure 2) as a result of an event. The solid arrow symbol used to denote a transition is the same symbol as that used for an association line with a navigability arrow on one end in a static structure diagram.[1]

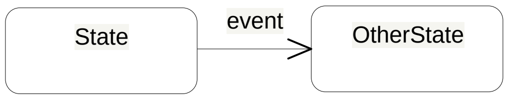

Figure 2 – Two states and a transition.

Events can come in a number of different flavours, depending upon what causes them (the syntax for these types of events in the UML is given later). A Change Event occurs, not surprisingly, when something changes, and is usually based on a Boolean expression becoming true (e.g. “when temperature > 0”). A Time Event occurs after a specified amount of time has elapsed (e.g. “after 1 minute”). Call Events and Signal Events are quite similar, and occur when another object initiates the transition. The difference is that the former events occur because of a direct call for an operation (method) from another object, whereas the latter occur from an explicit signal such as user input or an interrupt timer. In addition, hierarchies of signal events can be defined using generalisation, allowing transitions to occur if either the parent or child signal arrives.

A state machine has to start somewhere. A small filled circle is used to indicate the *initial state* in which the machine finds itself. When the machine terminates, an outlined circle can be used to show the *final state* of

[1] I didn’t mention *navigation* and the navigability of an association in previous articles. An arrow head on the end of an association line in a class diagram shows which way you can navigate, or find information about, the association. You can often not easily discover the participant of an association if you go against the flow of an association arrow. If no arrow heads are shown, it is assumed that navigation is either bi-directional or unspecified.

the machine. These two symbols are shown in figure 3. If a top-level state diagram models the lifetime of an object, then the initial and final states represent object creation and destruction respectively. They are actually pseudo-states in the sense that the system does not, and cannot, actually sit in these states.

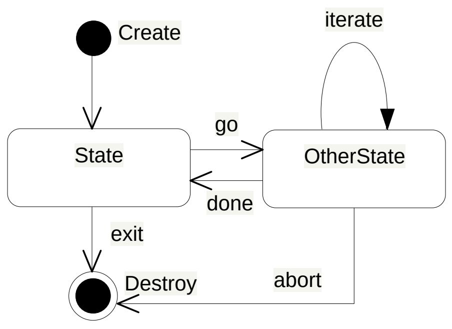

Figure 3 – A complete state-transition diagram, showing initial and final states, and self-transitions.

Self-transitions can be shown by drawing a transition line looping back to the state from which it began, as in the case of the iterate event in figure 3.

State-Transition Diagrams - An Example

I was trying to think of a good example to use to demonstrate state charts. I wanted something that most of you would understand or be familiar with. I always find it off-putting when a discussion presents an example that confuses me even more because of my unfamiliarity with the problem domain used! In the end I settled on a compromise. The subject matter may be foreign to a lot of you, but all of you who read Einar's article in the last issue of Overload should have heard of his Finite State Machine (FSM) model of a digital subscriber loop card. FSMs are ideal candidates for a state-transition diagram because they are described in terms of their different states and the possible transitions they can make between these states!

To refresh your memory, I have reproduced the table from Einar's article in table 1. Note that the table itself serves to document the FSM's behaviour. The format of this documentation, however, makes it hard to see quite what is going on. It also does not highlight errors in the design. For example, the only way to check that the system cannot get stuck in, or can never visit, a particular state, is to work through all possible states and stimuli in your head or on paper, and check they all make sense.

Stimulus	**Current state(s)**	**Next state**
decomm	<any>	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, Critical	Normal

Table 1 – Stimuli and Transitions for the FSM

In order to generate a state-transition diagram for this FSM, all we need to do is draw a box for each state, and then draw arrows to show possible transitions. We can also include documentation for the initial state of the device (Decommission), which is not present in the tabular format above. The result is shown in figure 4.

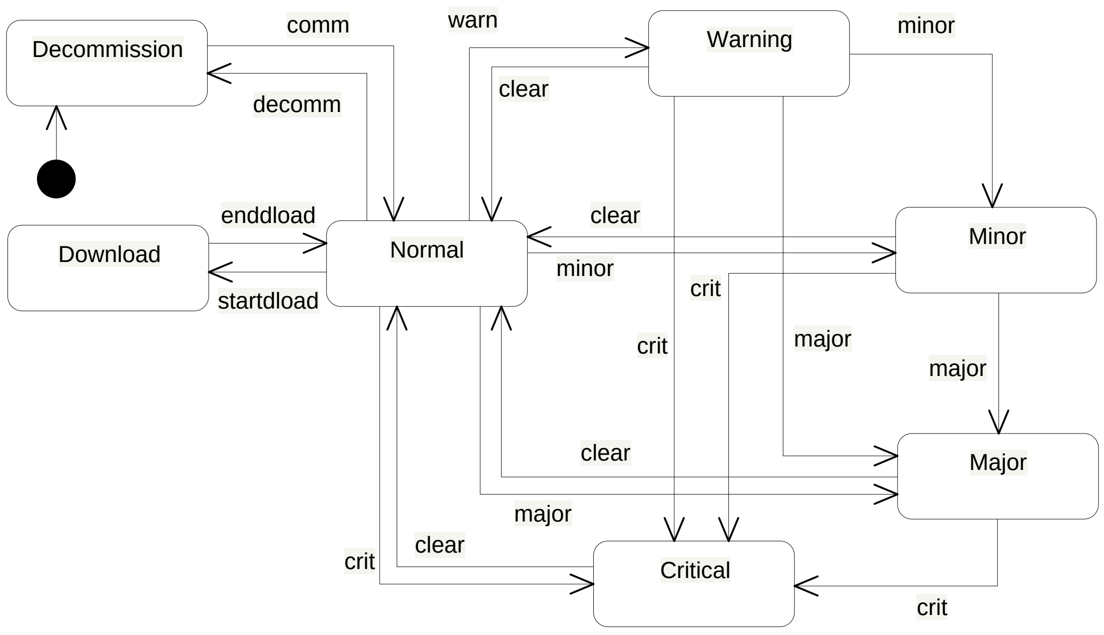

Figure 4 – UML state-transition diagram for the FS

Nested States

The diagram shown in figure 4 appears quite complex. This is because there are a number of similar error states that can all inter-convert as a result of a number of similar stimuli. This is where the use of nested states is useful. Nested states can be used to group areas of functionality where all the states are similar. They can also be used to show sub-algorithms that are present within a larger algorithm – in other words to show more detail of the inner workings of a state.

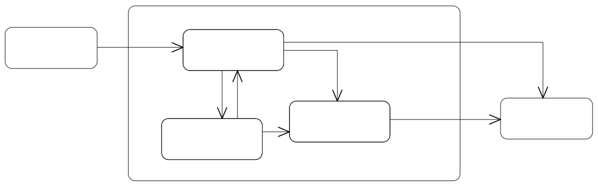

Figure 5 – Nested states in a state-transition diagram, with transitions into and out of the nested states.

Transitions can occur from outer states into the nested state, and transitions can occur from within a nested state to another outer state, as shown in figure 5. If the nested states form an independent and self-contained algorithm or state machine, initial and final states can be drawn within it, and entry and exit can be via transitions to the enclosing state, as shown in figure 6. Transitions to the enclosing state are equivalent to a transition to the initial state of the nested machine. An “action completed” transition from the enclosing state is equivalent to a transition (to and) from the inner final state. Other transitions can occur from the enclosing state, and these are equivalent to a transition from every enclosed state (although this transition can be masked [or overloaded] by an explicit transition in the nested machine for the same event). This consequently allows a potential economy in transition lines (see the “cancel” transition in figure 6).

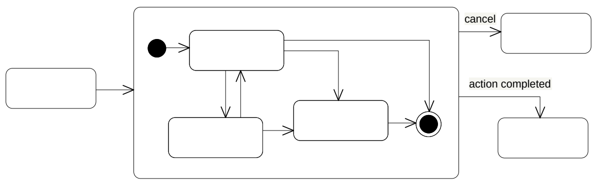

Figure 6 – A nested state machine, with transitions shown to and from the enclosing state.

In our example, there are a number of error states that can occur – *Warning, Minor, Major* and *Critical*. One possibility is to add an error state to our diagram as shown in figure 5, and nest the different types of errors within it. This simplifies the case where we clear an error condition, because a path can be shown from the Error superstate back to the Normal state, rather than from each of the inner states. This is, in fact, a slightly untidy use of nesting since there are so many transitions into and out of the Error state.

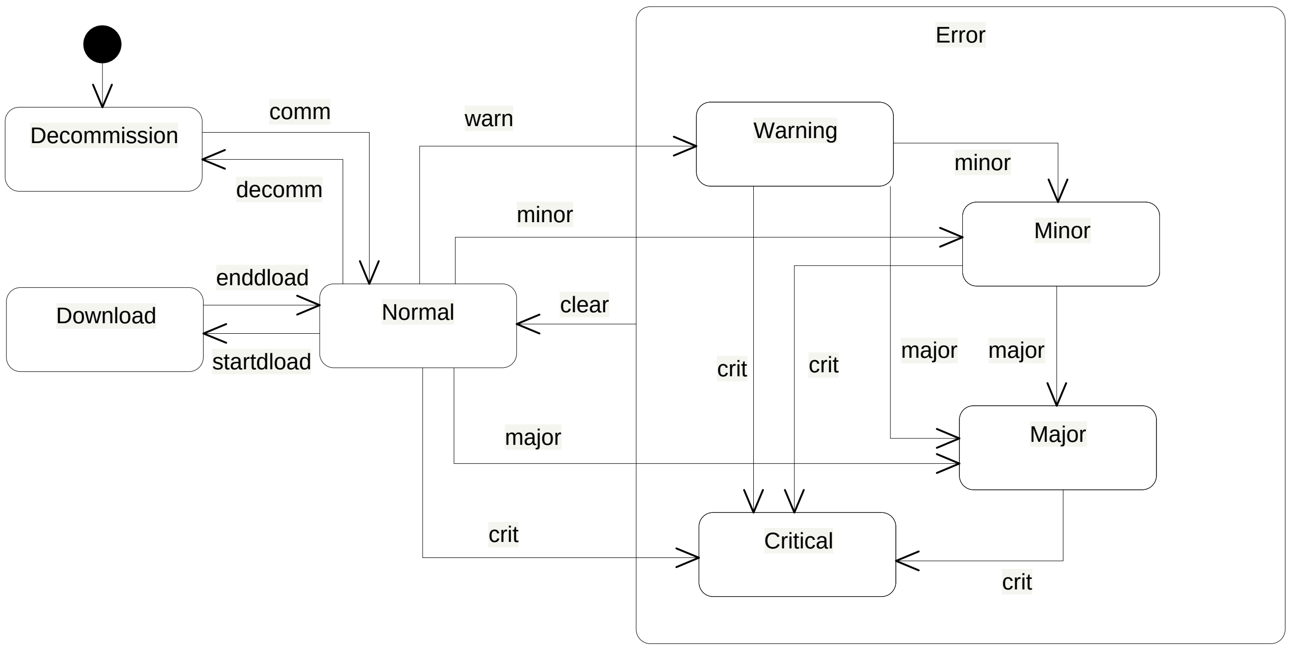

Figure 7 – UML state-transition diagram for the FSM using the Error state and nested substates.

Note how figure 7 makes it easier to check the transitions to and from the Normal state, as well as those between the error states, despite the fact that nearly every state can make a transition to every other one! Nesting states can help to remove clutter in two ways: by allowing functionality to be organised hierarchically with the possible suppression of finer detail; and allowing functionality to be partitioned into related activities.

Activity within States

So far we have just considered a state to be something that a system sits in statically until it receives an event. In general, however, a system will be doing something quite definite while it is apparently lying dormant in a state. It may be just waiting for something to happen, but it may also be processing keyboard input, updating a display, buffering data, or any number of things. Furthermore, there may well be some operations that need to be accomplished as soon as an object enters the state, such as setting flags or signalling its new state, and there may be things that it must ensure happen before it makes a transition to a different state. These operations could be modelled as another nested state machine, but you have to stop somewhere! This behaviour can be documented within the body of the state, although much of the detail will often be suppressed. The simple format of these so-called internal transitions is as follows:

event-name / action-or-activity

There are some special cases for the event-name. *entry* and *exit* specify the actions to be performed on entering and leaving the state. Note that these actions will be performed if a self-transition occurs (see figure 3), with first the exit action being performed, and then the entry one as the system re-enters the original state. If the operation of the state is itself modelled by another state diagram, the event-name *do* can be used with the name of the nested state diagram as the activity, implying that this state operates by running the nested machine. High-level state charts often use *do* a lot to allow the activity of a state to be described in simple terms or natural language, even if a state diagram has not even been prepared for the algorithm, for example:

"do / check order".

Some examples of these transitions are shown in figure 8.

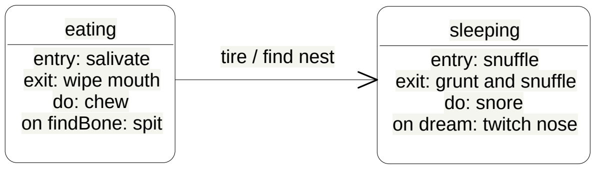

Figure 8 – Events and actions for internal and external transitions.

Actions during Transitions

The same format of "event-name / action" can be used to label transitions that have an associated action that occurs when the event fires (see figure 8). For example the transition:

"insufficient-funds / cancel order"

might link the states "Request Payment" and "Order Cancelled". The latter state signifies that there is no longer an active order, but the order was cancelled by the action of the transition itself. Note that if an action on a transition takes a significant amount of time, then it may instead really represent an activity that occurs within an additional state (so in figure 8 we might have a *walking home* state rather than the *find nest* action on the transition, with transitions *tire* and *when(found home)* linking the three states).

A couple of the common event types mentioned earlier have the following syntax. For a Change Event based on a condition becoming true, the *when* keyword is used:

when (controlRods == stuck) / explode,

and a Time Event uses the *after* keyword:

after (3 seconds) / hang-up.

The other types of event are denoted simply with the operation or signal name.

Concurrency

These days it is often important to consider how a concurrent system will behave. State-transition diagrams support concurrency by using dashed lines to separate the body of a state into parallel compartments. Each compartment represents one thread, and contains an individual state machine with its own initial and final states. Control passes to all the initial states concurrently, and each state machine continues until it reaches its final state. When all concurrent threads have reached their final states an "action completed" transition takes control away from the outer nesting state to the next (possibly also concurrent) state.

Conclusion

It should be pointed out that during the implementation of our Finite State Machine, the information contained within table 1 would be of enormous benefit. FSMs are often implemented with a single "Event()" method that works out the new state by using a lookup table. The data in Table 1 effectively flesh out this lookup table entirely. Why then the need for a State Chart?

I can think of two good reasons why State Diagrams are useful. Not all classes or systems act or are implemented as FSMs (in practice, only a tiny minority are). In this case, having tabular state/transition data for the class may be of little value for the implementers. The second reason is that these diagrams demonstrate clearly and concisely the behaviour of a system, and allow conceptual or functional errors or omissions to be spotted rapidly. The UML was designed for describing and modelling systems in general. Many of the modelling techniques of the UML are present for the conceptual and operational design and specification of a system. State-transition diagrams are useful for this phase of system development as well as at implementation time.

The Draft International C++ Standard

Embed with C++
By Kevlin Henney

Something interesting has happened to C++: it's got smaller. No, the ISO committee has not taken a set of shears to the core language and library, pruning it with a minimalist mandate. But someone has: a consortium of companies, mostly Japanese hardware and commodity electronics manufacturers, has taken the existing C++ standard and simplified it in line with the need of smaller embedded systems. In the West, P J Plauger is a key and keen advocate of the Embedded C++ spec (EC++).

You can download the spec, rationale, programming guidelines and other details from `http://www.caravan.net/ec2plus`.

Complexity

The needs of a small embedded system are few; it is more a case of knowing what is not required than what is. Embedded systems are lightweight pieces of software that run in a process control environment, typically controlling or responding to I/O from physical devices. Embedded systems can be found in your VCR, microwave oven, car, etc. Arguably they constitute the most widely used systems software in the world.

They must be standalone, frugal in their use of resources (this means code space, memory and execution time), and responsive to external events such as time or other device interrupts (hence the close association between the concepts of *real-time* and *embedded*).

Such software is not well served by full blown operating systems, large runtime, and much of the supporting paraphernalia we have in languages and libraries for large systems, GUI front ends, internationalisation, etc. From such a perspective we can see that assembler seems ideal for the job, and is indeed traditional. However, where there is assembler increasingly over the last decade or so C is likely to be found. And where there is C, why should there not be C++? In many respects C++ is a better C, providing a cleaner procedural language (better type system, `inline`, etc.), and there is also the addition of support for abstract data type programming and powerful user defined types. However, C++ now comes with a great deal more than this, and the extras are a potential stumbling block for some developers and platforms.

There are three kinds of complexity that we can identify:

Complexity of the language for the programmer who, in this case, is likely to come from a C and assembler background;

Complexity of the generated code and its memory consumption;

Complexity of the runtime support required, which overlaps to some degree with the previous point.

Turning back the clock

Given the tight memory constraints (we complain of the greedy software that seems to hog the many megabytes we have on our desktop, but reconsider this in terms of a handful of kilobytes) and performance requirements, features such as multiple inheritance, RTTI and exception handling demand a relatively high price for their use. You won't find them in EC++: no cunning *vtable* lookup mechanisms or layouts; no extra type meta-data hiding in your data segment; and no heavyweight stack frames to support exception handling.

Part of the philosophy that such features embrace is that we are programming in the large – indeed, C++ is often presented as a language specifically for programming in the large. In small programs this philosophy does not apply and therefore the features are not essential; in a non-embedded application these small overheads are typically non-issues, and are often the red herrings optimisation junkies slip on.

Multiple inheritance serves to express multiple classification schema, partial sets of properties and a method for mixing libraries and frameworks. In a small application whose classification scheme should be simple, whose properties are well known, and which is unlikely to use a multitude (if any) third party class libraries, the subtle issues thrown up by MI (code and language complexity) can be easily sidestepped by not supporting it.

One of the uses for runtime type information is to allow safe down and cross casting within a hierarchy, including casts through `virtual` base classes. Whilst safety is still an issue, there is little that `dynamic_cast`, `typeid` and `type_info` can offer to embedded systems. In this case with the baby goes the bath water: `static_cast`, `const_cast` and `reinterpret_cast` are all gone as well.

Hand in hand with safety comes the issue of reliability and exception handling. At first sight the C++ exception handling mechanism would seem to offer all the right features, however it imposes significant complexity at runtime for which there is a price to pay, e.g. the code support for dealing with destructing partially constructed objects. EH also requires much of the meta-data framework used for RTTI. It seems wise to remove this feature and resort to traditional error returns, relying on the simplified path coverage that is possible with a smaller self contained program to ensure a complete and well defined behaviour set. The memory consumption and execution time is far easier to determine without EH.

Thanks for the memory

And what of `new`? Once it returned null on memory exhaustion, now it throws `bad_alloc`... but EC++ has no exceptions. It may initially seem tempting to revert to null returning behaviour. However, an embedded system does not have the kinds of heap resources that larger applications get to play with – if, indeed, it has any. Dynamic memory must be carefully managed and usage limits must be known and planned for. Customised allocation becomes even more important in embedded systems than in larger applications, where often the off-the-shelf `new` and `delete` will do the job. Given this and the ability to install a handler with `set_new_handler`, it appears that programmers have all the control they need. This leads to the following simplification: `new` returns a pointer or the behaviour is undefined.

For a dedicated EC++ compiler – as opposed to someone working with the EC++ subset on an ordinary C++ compiler – a small optimisation is possible as there is no longer any need to check for a null return before executing a constructor on the newly returned pointer [2].

Where have all the keywords gone?

Some of the surprising things left out of the language include namespaces, templates and `mutable`. On closer inspection we can find some rationale for the first two. The use of namespaces is primarily motivated by programming in the large; mixing third party class libraries without conflict. This is clearly less of a requirement in embedded systems. In truth this was also probably motivated by the remoteness of such a feature from C and the aims of EC++: why trouble programmers who are essentially non-C++ programmers with the intricacies of `using namespace std` – let them eat prefixes!

[2] A change to the ISO C++ draft now makes this possible for some versions of `new`.

Unfortunately, this does create an area of incompatibility between EC++ and the forthcoming C++ standard. To write common code there is the issue of accessing library features that are in namespace `std` in one system and in the global namespace in the other. This can be resolved with judicious use of the `__embeddedcplusplus` macro defined for EC++ compilations:

```
#include <string>
#ifndef __embeddedcplusplus
using namespace std;
#endif
```

Perhaps another motivation for not including namespaces is the subtle name lookup rules, a minor complexity which is likely to trip up compiler implementers and users alike – this has certainly been the case with their introduction into the parent language.

An interesting area of potential conflict is that without namespaces one cannot have anonymous namespaces, which are intended to supersede the use of file scope `static`. Such is the intent that this use of `static` has been deprecated in the forthcoming C++ standard. Similarly, old style access declarations must be retained in EC++ because `using` has been omitted.

Following on from the issue of complex name lookup in namespaces is that of templates. They represent a useful feature for type safe, generic programming. However, in their current form it would be fair to say that there are a number of subtleties for implementers and users. The scale of an embedded application does not require such a useful uniform mechanism for reuse and safety, although the loss of the STL part of the standard library may be a bitter pill to swallow for current C++ programmers. Safety is no less of an issue, but the designers of this subset have definitely taken the view that we are talking about C programmers who have a `void *`, casts and preprocessor mindset.

There is no doubt that removing templates produces a simpler language, and one less prone to code bloat, but many will feel a slight twinge at the loss of such generic and type safe mechanism (interestingly, this is one feature that is often called for in Java even by some of the "hands off, don't touch that language" voices).

However, I can find no train of logic for dropping `mutable`. This supports the expression of logically `const` objects that may undergo physical state change, e.g. updating a cache on a query. The use of `const`, for whatever reason, is something that confuses many programmers. It is a design tool that can be carried through to implementation. If you do not understand `const`, you will not understand `mutable`.

The logic that led to the removal of this feature is subtly flawed. I suspect that the designers of the subset felt uncomfortable with the idea that one could have an immutable object with mutable parts, especially in an environment where some `const` objects are candidates for being placed in ROM. However, any object with a constructor is not really ROM-mable and so we are talking about traditional C type structures rather than class objects and so the argument does not apply. What will likely happen is that casts will be used to remove `const`-ness on "mutable" members, with slightly more drastic consequences: the behaviour is officially undefined.

To answer the question posed in this section's title, all the C++ keywords not used in EC++ remain keywords – in fact, they acquire the official status of "useless keywords".

Library visit

Whilst the impact on the language has been fairly comprehensive, it doesn't hold a candle to what has happened to the library. Clearly some of the language support has been affected by the changes, but most noticeable is the impact that the removal of templates has had: most of the library has gone!

Among the survivors we have...

- A `string` class based on `char`;
- `float_complex` and `double_complex`, rather than a templated `complex` class (and therefore a minor name incompatibility);
- Only `ios`, `istream` and `ostream` classes for `char` based I/O, with `cin` and `cout` as the only standard instances; and
- Most of the C library – although `assert` junkies should note that it has been dropped.

Gone is STL, the numerics library and internationalisation. However, some rationalisation can be found in terms of the purpose of embedded systems: a basic maths library that includes complex is an important requirement for signal and image processing; the rest of the numerics library would be "nice to have" but is not essential. And internationalisation? Not exactly a pressing issue on genuinely embedded systems! For many, the loss of perhaps one of the most comprehensive sets of general utilities in the form of the library's STL component is perhaps the hardest to stomach.

Discussion

In some respects Embedded C++ represents a return to its systems programming roots for C++. It is a mostly compatible subset that represents not only an interesting exercise in language subset design, but also a practical and revitalising influence on the language and its fortunes.

C++ is a general purpose language with low level roots that are easily exposed by simplifying it in line with a set of requirements. An interesting contrast to this is the interest in applying Java to high integrity and embedded systems: where features are removed from C++ to get to this simpler and more deterministic core, Java must be added to as this is not its core domain[3].

The Embedded C++ standard is a *de facto* one. It is defined by reference to the emerging C++ standard (i.e. omission or rewording of sections); by definition it cannot become officially stable before its larger parent does. Because of its relationship with C++, EC++ can be implemented either as a switch on full C++ compilers (as in the case of EDG front end), or as a language in its own right.

In an ideal world we would not need to define a subset, but we know that this world is not ideal. Compilers do not always take advantage of all the optimisations they might do. Vendors do not always provide compilers, or even full implementations, on the more restricted platforms. For small embedded platforms this can severely restrict the choice of a developer to either one poor C++ compiler or no C++ compiler at all.

The elimination of some features can be justified on the grounds of optimisation: no matter how well optimised exceptions and RTTI become, there is no code that has less overhead than no code, i.e. removing these language features will always result in an optimisation. For other features the case is perhaps less compelling: the overhead of MI is not present if it is not used, and templates have zero runtime overhead (although careless use could lead to code bloat). In response to this Dinkumware (`http://www.dinkumware.com`) provides both a core EC++ library and an STL for EC++, i.e. templates are included, but there are no heavy overhead features such as locales and exception safety related code. A superset of the subset, if you like. For many embedded systems – those of a higher spec – the scale of the software and

3 Historically Java (then Oak) was to be used for the development of embedded systems, but it has moved away from this centre, whereas C++ has merely added to its heritage. This explains the phenomenon of addition vs removal described here – as well as going a long way to explaining the difference in the size of each language!

the available resources do not fully justify the use of EC++.

There is another perspective that may help you understand the motivation for defining the subset. This is not just about optimising for a given set of platforms; programming is a human activity, and we must consider the human aspect. For existing C++ programmers, EC++ offers a little but not a great deal. For C programmers who might otherwise have had no intention, or opportunity, to move to full C++ EC++ offers them an opportunity. In other words, EC++ is optimised for a particular set of developers. A simple subset therefore has appeal to both vendors and developers, and therefore has the potential to increase the use of C++ and many of the techniques it supports. I welcome this, even though I do not necessarily agree with the rational behind all of the features (or lack thereof). Given a simple choice between programming without templates, exceptions and the like, versus programming an equivalent system in C, I know which I prefer.

But EC++ is not an excuse for programmers to indulge in wilful ignorance of the C++ language. It is a language subset for a subset of systems: given the high spec of many embedded systems these days, you would also not expect it on the majority of these. It really is a subset for tightly constrained systems, and goes further down than the Ada 95 subsets for similar systems can manage.

Some might be concerned that EC++ will affect the idiomatic use of C++; I can confidently say that although pure EC++ cannot embrace the most recent usage idioms, it is closer to the spirit of modern C++ than the majority of existing C++ code.

Without trotting out too many cliches, a field of languages provides you with "horses for courses": Embedded C++ runs a different race to the others. As yet, to quote Plauger, it is not a "strongly hyped" language.

Kevlin Henney
kevlin@acm.org

C++ Techniques

pointer<type>
By Jon Jagger

The built-in pointer is very powerful. And very dangerous. It's powerful because it can be used for many purposes. It's dangerous for the same reason. For example

```
class dodgy {};
void very(dodgy *ptr) { ptr++; }
```

Incrementing (or decrementing) a built-in pointer that doesn't point into an [array] makes no sense. The built-in pointer type is too powerful [1]. In C++ we can rectify this by creating different pointer classes for different pointer uses. I hope to cover specific pointer classes in coming articles but for now I'm just going to get the ball rolling with a general look at a pointer class.

A good place to start is a minimal pointer class. What is the minimal interface for a pointer class? To answer that let's look at a minimal interface for a built-in pointer.

```
class base {};
base object;

base *ptr = &object;// initialisation
ptr = &object;      // assignment
*ptr;               // dereference: *
ptr->method();      // dereference: ->
if (ptr != ptr);    // comparison: !=
if (ptr == ptr);    // comparison: ==

if (ptr);     // comparison: !=
              // null ptr, implicit

if (!ptr);    // comparison: ==
              // null ptr, implicit
```

Based on this, a first cut might be...

```
// accu/pointer.hpp
...
namespace accu
{
  template<typename type>
  class pointer
  {
  public: // construct/copy/destroy
    pointer( type *p = 0 );
    // default copy constructor
    // default copy assignment operator
    // default destructor
  public: // dereference
    type &operator*() const;
    type *operator->() const;
  public: // conversions
    operator bool () const;
    bool operator!() const;
  private: // state
    type *ptr;
  };
}

namespace accu // relational operators
{
  template<typename type>
  bool operator ==
    (const pointer<type> &lhs,
     const pointer<type> &rhs);

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

This is almost the minimal interface I have in mind, but not quite. What about public inheritance?

```
class deriving : public base {};
deriving lesson;
base *raw = &lesson;// initialisation
raw = &lesson;      // assignment
```

We need to ensure the pointer<base> object can be initialised/assigned from a pointer<derived> object.

```
pointer<base> ptr = &lesson;
ptr = &lesson;
```

This can be done. It requires two template member functions: a template copy constructor and a template copy assignment operator.

```
// accu/pointer.hpp
...
namespace accu
{
  template<typename type>
  class pointer
  {
  public:
    ...
    template<class derived>
    pointer
    ( const pointer<derived>& rhs );
    ...
    template<class derived>
    pointer &operator=
      ( const pointer<derived> &rhs );
    ...
  };
}
```

There are a couple of minor points of interest. Firstly, I have used <class derived> and not <typename derived>. Secondly, pointer<type> and pointer<derived> are separate types. pointer<type> has no access to the private data of pointer<derived>. For example, the following will not compile as a definition of the template copy constructor.

```
// accu/pointer_template.hpp
...
namespace accu
{
  ...
  template<typename type>
  template<class derived>
  pointer<type>::pointer
  ( const pointer<derived> &rhs )
    : ptr(rhs.ptr)
  {
    // empty
  }
}
```

I will return to this problem. Before I do, I'd like to cover a subtlety involving the template copy constructor. The C++ standard clearly states that a template constructor is never a copy constructor [2]. In other words, the presence of a template constructor does not suppress the implicit declaration of the copy constructor. A similar rule applies for a template copy assignment operator. Let's take a moment to think about those implicit declarations. There's the copy constructor, the copy assignment operator and the destructor. Be clear what these invisible compiler generated methods are...

```
// accu/pointer_template.hpp
...
namespace accu
{
  ...
  template<typename type>
  pointer<type>::pointer
  ( const pointer &rhs )
    : ptr(rhs.ptr)
  {
    // empty
  }
  ...
  template<typename type>
  pointer<type>
  &pointer<type>::operator=
  ( const pointer &rhs )
  {
    ptr = rhs.ptr;
    return *this;
  }
  ...
  template<typename type>
  pointer<type>::~pointer()
  {
    // empty
  }
  ...
}
```

There are two things about these compiler generated implicit methods you might question. Firstly, because they are implicit they're not, well, explicit. There is something to be said for having them in hard, visible ink in the interface. Especially in a teaching environment. Or if you want to single step while debugging. Secondly, they may not be quite what you want. It is impossible for any of these three to generate an exception (just as it is impossible in the corresponding raw pointer expressions) yet they do not have a throw() specification. For me these two factors tip the balance. Here's the revised class definition.

```
// accu/pointer.hpp
...
namespace accu
{
  template<typename type> class pointer
  {
  public: // construct/copy/destroy
    pointer( type *p = 0 ) throw();
    pointer( const pointer &rhs ) throw();
    template<class derived> pointer( const pointer<derived> &rhs ) throw();
    pointer &operator=( const pointer &rhs ) throw();
    template<class derived> pointer &operator=( const pointer<derived> &rhs) throw();
    ~pointer() throw();
  public: // dereference
    type &operator*() const;
    type *operator->() const;
  public: // conversions
    operator bool () const throw();
    bool operator!() const throw();
  private: // state
    type *ptr;
  };
...
}

namespace accu // relational operators
{
  template<typename type>
  bool operator == (const pointer<type> &lhs, const pointer<type> &rhs) throw();

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

I have left the operator*() and operator->() declarations without a throw() specification. The bodies of these operators are ideal places to check for a null pointer and throw an appropriate exception. However, what is an appropriate exception? The C++ standard basically gives a choice of two. logic_error and runtime_error. A logic_error is an error that the user could (at least in theory) avoid. Dereferencing a null pointer<type> is avoidable since the user can make the check themselves. For example via the bool conversion operator. A reasonable exception is therefore a logic_error. One way to implement this would be create a private method called check_not_null() which operator* and operator-> could then call. However, check_not_null() would then appear in the interface. Private but still visible. Really it is part of the implementation. I prefer my interface files to be as clean as possible. Also, there is still the problem of how to implement the template copy constructor, the template assignment operator and the global comparison operators. One solution is to provide a simple auto_ptr-like accessor called get(). It might be important to allow easy access to the underlying raw pointer (to use dynamic_cast for example).

```
// accu/pointer_template.hpp
#if !defined(ACCU_POINTER_INCLUDED) || defined(ACCU_POINTER_TEMPLATE_INCLUDED)
#error include "accu/pointer.hpp" : pointer_template.hpp must not be included directly
#endif
...
#define ACCU_POINTER_TEMPLATE_INCLUDED
...
#include <exception>
...
namespace // unnamed
{
  template<typename type> void check_not_null( type *ptr )
  {
    if (ptr == 0) throw std::logic_error(“pointer: null”);
  }
}

namespace accu // construct/copy/destroy
{
  ...
  template<typename type> template<class derived>
  pointer<type>::pointer( const pointer<derived> &rhs ) throw()
    : ptr(rhs.get())
  { // empty }
  ...
  template<typename type> template<class derived>
  pointer<type> &pointer<type>::operator= ( const pointer<derived> &rhs ) throw()
  {
    ptr = rhs.get();
    return *this;
  }
  ...
}

namespace accu // dereference
{
  template<typename type>
  type &pointer<type>::operator*() const
  {
    ::check_not_null(ptr);
    return *ptr;
  }

  template<typename type>
  type *pointer<type>::operator->() const
  {
    ::check_not_null(ptr);
    return ptr;
  }

  template<typename type>
  type *pointer<type>::get() const
  { return ptr; }
}

namespace accu // comparison
{
  template<typename type>
  bool operator == ( const pointer<type> &lhs, const pointer<type> &rhs ) throw()
  { return lhs.get() == rhs.get(); }

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

One issue that still remains unmentioned is whether the constructor should be explicit or not. Consider the consequences if the constructor was made explicit...

```
void oops( const pointer<base> &lhs,
const type *rhs )
{
  pointer<base> local = lhs;
      // FAILS, have to use 1
  if (0 == lhs)...
      // FAILS, have to use 2
  if (lhs == 0)...
      // FAILS, have to use 2
  if (0 !== lhs)...
      // FAILS, have to use 3
  if (lhs != 0)...
      // FAILS, have to use 3

  pointer<base> local(lhs);
      // WORKS, 1
  if (!lhs)...
      // WORKS, 2
  if (lhs)...
      // WORKS, 3
}
```

Is this better? It's perhaps a matter of personal preference. But there is a difference. Which is more explicit?

```
if (lhs)...         or
if (lhs != 0)...
```

I think the answer largely depends on the level you're viewing from. You might argue that the latter is more explicit because it's explicitly comparing lhs to the null pointer. But is it? Zero is not the null pointer. It's zero! By the same token you might argue that the former is more explicit because it's not explicitly comparing lhs to zero. At a higher level you can read if (lhs) as “if lhs is true” or “if lhs is valid”. Whatever you feel, ultimately even if the constructor is explicit, you can make all versions of the comparisons work. You just have to provide global operators. For example...

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

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

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

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

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

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

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

Jon Jagger
jjagger@qatraining.com

STL Algorithms: Finding By Francis Glassborow

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

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

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

C offered just one library mechanism for searching: ***bsearch*()**. You might guess that ***bsearch*** applies a binary search. There is nowhere in the ISO C Standard that places any such burden on the implementor. The only limitation is that the elements of the array that match the required criterion shall come after all those that compare less than and before all those that compare greater than. In other words the array shall have been so sorted that a binary search would work. Of course most implementors will use a binary search because that would seem the obvious solution to the problem but their choice is a pure quality of implementation issue.

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

Sequences & Containers

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

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

Unordered Sequences

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

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

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

Where is the next instance of something?

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

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

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

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

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

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

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

How many instances of a value occur in the container?

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

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

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

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

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

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

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

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

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

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

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

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

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

Sub-sequences

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

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

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

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

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

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

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

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

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

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

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

Sorted Sequences

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

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

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

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

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

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

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

Conclusion

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

Francis Glassborow
Francis@robinton.demon.co.uk

Whiteboard

Rational Value Comments By Graham Jones

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

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

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

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

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

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

Graham Jones

Rational Values Part 3
By The Harpist

First let me thank Graham for taking the trouble to write this letter drawing my attention to one of the obscure corners of mathematics that is easily forgotten, even by those who know of its existence. *Continued fractions* is a powerful tool and one that deserves to be better known. Perhaps it deserves some coverage in Overload so that programmers can add them to their toolkits.

Now, let me address the rest of his comments.

Purpose

My aim in writing about a rational class was to cover various aspects of the design of a pure value based class. I was not trying to provide an industrial strength implementation. I am a great believer in providing knowledge in a relevant context. I think too many writers introduce things like ‘mutable’, ‘explicit’ etc. and then thrash around for an example. I prefer to start with something that seems reasonable as an objective and then see how various facilities become possible answers to problems.

Things like the use of `mutable` to provide what is, in essence, a caching facility. That is a general idea that can be used in many different circumstances.

One of the major problems of class design has been that different designers have very different views as to what is required. Often several views are equally valid and only the application domain can make one preferred to another. This problem lies at the root of the difficulties that WG21/X3J16 experienced in designing a string class. Everyone has their own needs when it comes to strings. You cannot even achieve a compromise by providing a slim base class from which individuals can derive their own application specific version because some of the differences lie deep within the low-level design.

Think about how C managed arrays. It provided a very primitive facility that actually fails to meet the needs of all but very low level programmers. When the language designers were asked why they had not provided something that was more robust and relied less on responsible programming they claimed that every application domain makes its own demands on the array concept and that those working in these domains should craft their own array abstraction. Sadly very few programmers seem to understand this. Instead they blame C for providing minimalist facility.

Curiously the C++ `string` class (or to be precise, the `basic_string` template class) is what happens when you try to take the union of everyone’s wishes. The interface is very fat and I suspect that implementations will be less than efficient. In many cases this will be fine. The majority of programmers will find that an instantiation of the standard template class will meet their needs adequately. Memory demands and fat interfaces are relatively less important these days. However where more performance is needed programmers will need to write their own string abstraction.

We should get into the habit of encapsulating our components into suitable namespaces. Even were I certain that a particular class was the perfect abstraction I should still respect the views of others by wrapping my work in a namespace.

Using a namespace

When we look at the implementation of the rational abstraction we realise that some parts of that implementation necessarily leak out of the class. Things like the implementation of `operator<<` and `operator>>` for streams have to be outside the Rational class. However they are inherently part of the abstraction. Now that we have `namespace` all this baggage should be encapsulated into a `namespace`. So we should write:

```
namespace RationalSpace
{
  class Rational
  {
    // all the normal class
    // based material
  };
  // all the conventional
  // out of class support
}
```

One advantage of a namespaces for encapsulating an abstraction is that they can be re-opened. By this I mean that extra material can be added elsewhere at a later stage. Obviously it would be bad programming practice to invade some other programmer's namespaces but the extensibility of namespaces serves a similar purpose to `public` inheritance used to add features to a class. One thing you cannot do in a namespace is to override functionality provided elsewhere in the namespace.

Many things that we have previously provided within a class scope might now be exported into an encapsulating namespace scope. When programmers get used to using declarations rather than just lazily writing `using` directives we will be able to place such things as `enum`s and `typedef`s into the `namespace`. This will simplify the correct use of namespaces. Applying this to my Rational abstraction and we might get:

```
namespace RationalSpace
{
  typedef unsigned int integer_type;
  class RationalException {};
  class RangeError:
    public RationalException{};
  // more exception classes
  class Rational
  {
    // private interface
    public:
    // public interface
  };
  // out of class functionality
}
```

Note the empty classes that provide types for exception handling. Of course exception objects do not have to be vacuous but very often all we need is a mechanism to identify what kind of exception occurred. Building them into hierarchies is desirable because it allows users to catch distinct exception types or bundles of them.

I still haven't introduced you to the full power of `namespace`. I should do something to identify that this material is part of a tutorial. The obvious thing is to wrap all my tutorial material into a namespace. So we get:

```
namespace Tutorial
{
  namespace RationalSpace
  {
    // as before
  }
}
```

and I can re-open `namespace Tutorial` to add other tutorial material. Unfortunately that name is a little too obvious and once the idea gets around we will have problems from using components from different people's `namespace Tutorial`. I need to be a bit more verbose and write something like:

```
namespace TheHarpistsTutorial
  {
  namespace RationalSpace
  {
    // as before
  }
}
```

By now you are beginning to think that all this is fine in theory but it is all becoming very verbose. If you have understood what namespaces are intended to provide you will realise that:

```
using namespace
TheHarpistsTutorial::RationalSpace
```

removes the verbosity at the cost of polluting the global namespace and opening the door to all those name conflicts we are trying to avoid. There is no point in providing a facility that is so clumsy that people are going to go back to the bad old ways. What we need is a nice short alias and C++ provides just what we need. We can write:

```
namespace Rational =
TheHarpistsTutorial::RationalSpace;
```

This mechanism doesn't just reduce verbosity but it also allows us provide our choice of components with locality. Suppose that I wish to switch from using the component choice to an industrial weight one provided by someone else. I can write:

```
namespace Rational =
JohnSmithsIndustrial::RationalNumber;
```

As long as the substitute provides equivalent functionality the rest of my code will work. You might think that this is pushing my luck, but what about being able to switch between versions of the same library? If the library implementor understands the use of namespace they will ship new releases in versioned namespace so we might have:

```
namespace MyLibrary_v1_0
{
  // version 1.0
}
```

Later we would have:

```
namespace MyLibrary_v1_1 {
  // new release
}
```

and so on.

The user who is uninterested in what version they are using can be insulated from the feature by having header files that start with:

```
namespace MyLibrary = MyLibrary_v1_0;
```

in the first release, and equivalent statements for later releases. Those who care would be able to use more primitive headers that used the 'true' namespace rather than the alias. They could then use the alias mechanism to select the version they wish to use if they wanted to be able to control change. That way you can keep multiple versions on your system and not have to worry that a work around for an earlier version will blow up on a newer one.

Of course all this only works if we can persuade library vendors to use the facilities of the language.

To Convert Or Not

Graham raises an excellent point with regard to my decision to have a constructor that has a `long double` parameter. In some application domains it would certainly be a mistake. It is also true that providing it encourages abuse. But where we would part company is in where that abuse would arise. Programmers often know various mathematical constants as decimals and it becomes tedious to have to convert these to good rational approximations. I think most would expect to be able to create a rational from a floating-point type. Anyway even if you disagree, this is a tutorial exercise. However just because a responsible programmer can make a considered decision is no reason for allowing an irresponsible compiler to play fast and loose. It was for this reason that the keyword `explicit` was introduced into the language.

Just as good programmers qualify global functions with `static` until they know that they want to use them outside the current translation unit they also qualify constructors that can be called with a single argument with `explicit`. That way the compiler cannot use such constructors as conversion operators. If we make such a qualification so that the constructor in question is declared in Rational as:

```
explicit Rational(long double);
```

then the following code will fail to compile:

```
int main()
{
  Rational rat;
  rat = 1.2;
  return 0;
}
```

In order to get it to compile we would need to write:

```
rat = Rational(1.2);
```

Incidentally this is one of the few cases where I would not use a new style cast. The so-called function-style cast, which is really an explicit call to a constructor, seems more descriptive to me.

Now if you elect to go along with Graham and avoid a constructor taking a `long double` then I think that you should provide a function that takes a `long double` and returns a `Rational` so that programmers can call that function if they wish to. The question that may still arise is whether that should be in-class or merely in the encapsulating namespace. I can think of arguments both ways. Note that, to be fully useful, in-class it would need to be a `static` class function (otherwise you would need to have a `Rational` object to use the function). Of course you would need to place it in-class if it needed access to any private member functions of Rational, but they in their turn would need to be static members. On balance, now that we have `namespace` to encapsulate utilities I would tend towards providing it as a utility function in the encapsulating namespace.

Before I move on, there is another thing that you should always do when providing a public interface, you should always provide an exception specification for any user provided destructor. I do not believe that there is any choice here. The exception specification for a destructor should always be that it does not throw. If I provide a destructor for my Rational class its prototype should be:

```
~Rational() throw();
```

If you wonder why a destructor should never throw an exception (so it must handle any possible exceptions internally) think about what happens when an exception is being processed – destructors get called. Nested exceptions are one thing but overlapping ones must be bad news.

Interfaces & Implementations

I cannot help but think that Graham has confused interface design with implementation. Let me examine the problem a little further and see if what useful insights might come from our disagreement.

Graham claims that as we are likely to want to use a Rational type only where exact computation is desirable we should first choose an implementation that supports this. To me this seems like saying that implementation drives design.

My perspective is very different. Design is largely concerned with getting your interface correct. Application drives design. Part of the early phase is to create a public interface together with a test suite whose job will be to ensure that we maintain the semantics of our objects as we refine the implementation.

Actually many class designers expose details of their implementation when they select the return types of member functions. That is a serious flaw that needs to be addressed by better training. If for no other reason I would have set about designing my Rational class in such a way as to hide the implementation details. In essence there are three ways of doing this, by using `typedef`s (as I did in my first cut), by using an opaque type and by using a template.

Each of these methods has both advantages and disadvantages. When we gain skill we will be able to mix them.

The principle advantage of using a `typedef` is that when the underlying type is a built-in it results in efficient code which can lean heavily on built-in operators. Its biggest disadvantage is that it does not create a true type. So let us look at the other options:

Opaque Types

We can achieve opaque type in several ways. The simplest is via a class wrapper:

```
class Integer
{
  long int value;
public:
  Integer(long int v):value(v){}
  operator long int (){return value;}
};
```

That is about as simple as you can get. An `Integer` will behave exactly like a `long int` except that it will have a different type that can be used for overloading. You may worry that something like this will cause a problem:

```
class Another
{
  long int value;
public:
  Another(long int v):value(v){}
  operator long int (){return value;}
};
```

You might fear that an `Integer` could be used where an `Another` was required. You would be mistaken. The conversion from `Integer` to `Another` via `long int` requires two user-defined conversions. While you can perform that with a cast the compiler cannot do the conversion on its own initiative. It is a pity that Microsoft did not use this mechanism instead of `typedef` for many of its MFC types.

If you want more control, remove the conversion operator. More still? Then make the constructor `explicit`. Once you remove the conversion from `Integer` you will need to start providing functionality. For example if you want an integer type that can only be modified by addition and subtraction you will need something like this:

```
class LimitedInt
{
  int value;
public:
  LimitedInt(int v) : value(v) {};

  LimitedInt add(LimitedInt const & rhs)
const { return (value + rhs.value);}

  LimitedInt negate()const
  {return LimitedInt(-value);}
}

LimitedInt operator +
    (LimitedInt const & lhs,
    LimitedInt const & rhs)
{return lhs.add(rhs);}

LimitedInt operator –
    (LimitedInt const & lhs,
    LimitedInt const & rhs)
{return lhs.add(rhs.negate());}
```

Of course you now know that this sort of thing should be wrapped in a namespace. You can probably think of a number of other points that should be considered such as possibly providing `operator +=()` and `operator -=()` as member functions. You might also consider calling `negate operator-()`. All I want to illustrate is how easy it is to produce your own variations on the built-in types.

Another feature of these user-defined versions is that they are classes and so can be used as base classes if you wish.

You also have control of the conversion rules but supplying your own promotions to replace those inherited from C is rather harder though not impossible.

A rather simpler opaque integer type is using an `enum`. However this time you have to remember that there are no constructors so you have to use a cast when doing arithmetic. For example:

```
enum Integer {low=-1000, high=1000};
Integer x=Integer(12);
Integer y=Integer(14);
Integer z= x + y;// ERROR, no conversion
z = Integer(x+y);   //OK
```

The purpose of the two enumerated values is to ensure that the type is valid for at least that range. C++ does not guarantee that all integers will be valid values for an `enum`. The rule is a little complicated but certainly everything between the `low` and `high` values will be OK. Note that a weakness of using an `enum` type is that you do not have the ability to control the arithmetic operations to the same extent that you do with class types.

Let me return to using classes to provide opaque types. Once you have a class type as a base you might consider deriving from it to save having to write more than necessary. This is an excellent principle and any time you are tempted to write:
`typedef Sometype Mytype;`

You should consider derivation instead. However I would council you to think carefully about the degree to which you want to expose the base class. For example:
`class Mytype: public Sometype {};`

Is usually better than a `typedef` because `Mytype` becomes a true type. On the other hand there is an automatic conversion from derived to base so that `Mytype` will behave exactly like `Sometype`. If that is what you intended (which would have met many of Microsoft's needs for handle types) then fine. However you may want to prevent the conversion from `Mytype` to `Sometype`. It is largely a matter of style whether you write:

```
class Mytype : Sometype
{ };
```

or

```
class Mytype
{ Sometype value; };
```

I say largely because one recent extension in C++ makes the first more favourable. You can pull in the functionality that you want from the private base class with using declarations. That option isn't available if you use the layering version. A small point but one worth consideration.

I could write a lot more on the subject of opaque types but I will leave it for now else this article will never get finished. I already have the editor moaning that my writing style lacks polish (well I wish I had time to both learn what to write about and to polish my writing.)

One final thought about my Rational class, should it be a template class? Perhaps I will explore the answer to that next time round. In the meantime please let me have your thoughts on that and the rest of this article.

The Harpist

Protecting Member Data's Right to Privacy

By Mark Radford

Introduction

About two years ago, when I was working on the development of a two dimensional CAD program, I had a long running debate with a colleague: I maintained that when implementing and object-oriented design (OOD) in C++ the data members of a class must be *private*. He argued that they should be *protected* (or even *public!*), as making them *private* was too restrictive. After all, if you make them *private*, you need to write functions which will both set and return their values, so you might as well make the data members *public* (or *protected*, if access is needed only by derived class member functions). My cause was not helped by the use of public and protected data in the Microsoft Foundation Class (MFC) library, which we were using in order to develop for Microsoft Windows.

The objective here, is to show why keeping data *private* is not only a good idea, but is an essential C++ practice. First, I will show the benefits of *private* data. Then, present two examples of how failure to adhere to this rule will cost in the long run.

A Closer Look at Encapsulation

When expressing object-oriented designs in C++ class member data <u>must</u> be made *private*. The only possible exceptions to this rule are *enums* and values which are declared *static const*. This view is widely supported in C++ literature (for example KH96, AH95 and JL96), and for several good reasons:

1) One of the key elements of object oriented design is encapsulation. Throughout the literature, this is something on which there is general agreement (for example see GB94, JR91). Making the state of an object private, and therefore hiding it from the object's clients, is a natural way to implement encapsulation.

2) The very nature of OOD suggests that if you need to expose member data as *public* or *protected*, then something has gone wrong at the design level! In OOD, the idea is that objects will communicate at an abstract level by passing messages to each other; these messages will either request a service from the receiver, or notify the receiver of an event.

3) Keeping member data private makes the C++ code much more resilient to change, as shown in the examples below.

4) It is easier to debug the code by placing trace messages in access functions, or by putting break-points in them when using a source code debugger.

Returning to the second point above, it is important to realise that (as Allen Holub observes in AH95) functions which exist only to set or return the value of a member variable are also out of order. However, this does not make it wrong for a member function to return information about the state of an object. For example, a *CommsPortManager* class might be legitimately defined as follows (assuming the *CommsPort* class is suitably defined):

```
class CommsPortManager
{
public:
  unsigned int GetCommsPortCount() const;
// Number of ports available
  CommsPort GetCommsPort(unsigned int
portNumber);
private:
  enum { MAX_PORTS = 4 };
  CommsPort ports[MAX_PORTS];
};
```

Whereas the following would definitely be wrong:

```
class CommsPortManager
{
public:
  CommsPort* GetPortArray(); //  etc
};
```

In the second case there is no way to implement *GetPointArray()* if the implementation is changed to use a linked list.

Examples

I will now present two examples. These attempt to demonstrate how the use of data which is not *private*, can cause suffering in the development process.

Example 1: Implementation Flexibility

Suppose you are developing components for a large CAD product, including classes to represent various shapes, and one of the classes is *Circle*. One method of defining a circle is to store it's centre point and radius (other possibilities are storing the centre point and a point through which the circumference passes, three points through which the circumference passes, or a bounding box). It seems reasonable that clients of *Circle* may wish to both query and update the radius. Therefore, the radius is a *public* data member, and the class looks like this:

```
class Circle : public Shape
{
public:
    double radius;
    Point centre;  // assume a suitable
Point class exists

    // ... rest of Circle ...
};
```

There is however, a problem with this approach. It is not clear at this stage, if the radius or diameter will be required more often by the client code. Also, it is thought likely, that the wrong decision will lead to performance problems. Therefore, instead of the above, the designer of the *Circle* class designs the following:

```
class Circle : public Shape
{
public:
  double GetRadius() const;
  double GetDiameter() const;

  void SetRadius(double r);

private:
  double radius;
  Point centre;

  // ... rest of Circle ...
};
```

Which has these function definitions in it's implementation file:

```
double Circle::GetRadius() const
{
  return radius
}

double Circle::GetDiameter() const
{
  return (2.0 * radius);
}

void Circle::SetRadius(double r)
{
  radius = r;
}
```

This (and other classes) are developed, tested, and passed on to other teams who are developing components which require the classes based on Shape. A subset of the client code is developed and then profiled. This exercise reveals that it looks like the diameter of the circle is needed more often than the radius and that is would have be better to implement the circle in terms of the diameter. This is not a problem, because the functions *GetRadius()* and *GetDiameter()* are still valid (having been re- implemented). These functions can also be made *inline* if necessary for performance. No changes are required to any of the client code!

Example 2: Putting Functionality Where it Belongs

It isn't just *public* data which causes trouble: *protected* data is bad too. This can be much harder to convince people of. In fact it is one (but not the only) measure of whether or not they have made the switch to the object oriented way of thinking.

Consider the case of classes designed to provide basic text display capabilities, possibly for use in the graphical front end of a text editor. The *TextWindow* class provides generic text display services, while *AppTextWindow* is derived from it, for use in a specific application. One thing it seems reasonable for the *TextWindow* class to manage, is information about the font in use.

Assuming the class *ScrollingWindow* is suitably defined (probably by the graphical environment's API library), definitions of these classes might look as follows. The *TextWindow* class like this:

```
class TextWindow : public ScrollingWindow
{
public:
  // ...
protected:
  unsigned int textHeight;
private:
  // ...
};
```

and the application specific class like this:

```
class AppTextWindow : public TextWindow
{
  // ...
};
```

The intention is that the *TextWindow*, maintains the *textHeight* variable, which stores the height of the current font's text. This variable is *protected* so that, when the mouse is clicked on a window containing text, the object of class *AppTextWindow* class can work out what line of text the mouse was clicked on (I am assuming that the function which processes the mouse click event will be a member of the *AppTextWindow*).

Might there be problems with this? What happens if we want to expand the font information stored in *TextWindow*? Instead of a single *textHeight* member, we might prefer to use a structure, or even make the font a class in it's own right. Further, so far the assumption has been that each line of text will be displayed in the same font, but in the future there might be a requirement to enhance the *TextWindow* class to use a different font for each line. This would make it useful to apply an implementation sharing technique to the fonts.

Any of the above changes will impact on the client code, but this can be avoided by paying attention to the design of the *TextWindow* interface. Given it's role, it should have a member function called (something like) *GetLineFromY()*, as shown below.

```
class TextWindow :
                public ScrollingWindow
{
public:
  unsigned int
  GetLineFromY(unsigned int y) const;
private:
   // ...
};
```

I have made the function *public*, because it provides a service which clients can reasonably expect of the class. There is no good reason to restrict this service to derived classes.

Finally

Modern C++ software is complex, and getting more complex all the time. Developers should use any available method to tame this complexity, and OOD offers one such method. The problem is, there is now a myth that any software written in C++ uses OOD, even if it breaks fundamental object oriented principles. Encapsulation is the most fundamental of these principles. This is why it is so important to ensure that the state of an object is only modified by the methods of that object. After all, how do you test any component if it's state can be influenced by other parties? In short, this illustrates how OOD is used as a buzzword, rather than properly understood. This is the reason why development environments which move from C to C++ often do not get the benefit they should.

References

[GB94] "Object-Oriented Analysis and Design with Applications" by Grady Booch (second edition). Published by Benjamin Cummings ISBN 0-8053-5340-2.

[JR91] "Object-Oriented Modelling and Design" by James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. ISBN 0-13-630054-5.

[KH96] "Some OOD Answers" by Kevlin Henney in Overload 16.

[AH95] "Enough Rope to Shoot Yourself in the Foot" by Allen I. Holub. Published by McGraw-Hill. ISBN 0-07-029689-8.

[JL96] "Large-Scale C++ Software Design" by John Lakos. Published by Addison-Wesley. ISBN 0-201-63362-0.

By Mark Radford
mark@twonine.demon.co.uk

‘There may be trouble ahead’ By Seb Rose

Motivation

In C++ a class is dependent not only on its base classes, but also on any class that it uses. This is not wholly unexpected! We might hope, however that adding a private method or member variable to one of these classes might not affect the client classes. This is unfortunately not the case. All translation units that (directly or indirectly) #include the file that defines the class that has changed will generally be recompiled.

In the real world this can sometimes be a burden too great to bear, and some way of insulating derived classes from changes in the utility classes that they use is required.

There are many techniques used to limit the impact of these dependencies, but they require that the public interfaces provided remain constant. See Lakos 1996 for discussions on many ways to limit dependencies.

In some cases it may be desirable to simulate the changing of a public interface without causing the recompilation of all client code.

A Real World Problem

Your company produces many different devices and a suite of software to control them. The devices have much functionality in common, but have important differences too. Over time more devices will be produced and it is not known what further differences between the devices may emerge.

We specify an interface that all devices will conform to and provide this as an Abstract Base Class that each device can derive from. This allows our software to control all our installed devices through polymorphic pointers.

```
class AbstractDevice
{
public:
  virtual void Start() = 0;
  virtual void Stop() = 0;
  // etc.
};

class RealDevice : public AbstractDevice
{
public:
  virtual void Start();
  virtual void Stop();

private:
  // Implementation details
};
```

We can also provide some common functionality that many of the current devices can use. If the device code accesses the common functionality through an opaque pointer, then unforeseen changes to the implementation of the generic code will not impact the (many) devices that will be deployed.

The pure virtual functions of the interface allow each device to specialise its behaviour as much as it likes. The provision of code that implements functionality that is common to most devices eases development and maintenance. The use of an opaque pointer reduces further the maintenance burden.

It turns out that each device is remarkably similar. Most of the code can be placed into the GenericCode utility class. Everything looks fine.

However, a couple of months later a new device is produced in response to emerging technology. It is very similar to the existing devices, but the generic code is not quite right for this device. An extra call is necessary in the middle of a complex sequence of operations:

```
void GenericCode::DoSomethingComplex()
{
  ComplexStuff();
  // Need to do something new here
  MoreComplexStuff();
}
```

One solution could be to cut and paste the generic code into the new device implementation and customise it. This is not an attractive solution:

```
void OldDevice::DoSomethingComplex()
{
  pGeneric->DoSomethingComplex();
}

void NewDevice::DoSomethingComplex
{
  ComplexStuff();
  SpecialStuff();
  MoreComplexStuff();
}
```

Alternatively we could split the complex sequence of operations into smaller operations. This would allow us to insert the new code in the implementation of the new device, but at the cost of changing the interface to the generic code's opaque pointer:

```
void GenericCode::ComplexStuff()
{
}

void GenericCode::MoreComplexStuff()
{
}

void
CompliantDevice::DoSomethingComplex()
{
  pGeneric->ComplexStuff();
  pGeneric->MoreComplexStuff();
}

void NewDevice::DoSomethingComplex
{
  pGeneric->ComplexStuff();
  SpecialStuff();
  PGeneric->MoreComplexStuff();
}
```

Also, any further changes required for other devices in the future would require the same difficult decisions and costly changes to be made.

A Solution

What we really want to do is provide a facility that allows an optional specialisation to a piece of generic code (that has already been implemented, tested and released) to be added without causing any existing code to be re-released.

I achieve this using named callbacks. Each callback maps to a specialisation that was not envisaged at design time, and cannot easily be inserted using traditional methods without an unacceptably large maintenance burden.

Here I present the skeleton class definitions for a pair of cooperating callback classes. This develops ideas presented by Coplien 1992

I have left out error handling for clarity.

Please substitute STL containers and strings at will.

```
class CallbackServer
{
public:
  CallbackServer();

  // Called by implementation to
  // register callbacks
  int AddCallback(char *pCallbackName);

  // Called by client to signal that
  // this callback is processed
  int UsesCallback(char *pCallbackName);

  // Called by client to initialise
  // pointers used during callback
  void SetupCallback(
         CallbackClient* pTheClient );

  // Called by server to make a callback
  int MakeCallback(
  int callbackId, void* pData );

private:
  // this pointer of client
  CallbackClient*          pTheClient;

  // ID of next callback to be added
  int         nextCallbackId;

  // Names of callbacks
  char* callbackNames[ maxCallbacks ];

  // Flags that indicate whether the
  // client processes this callback
  bool callbackInUse[ maxCallbacks ];
};

class CallbackClient
{
public:
  CallbackClient();
  virtual ~CallbackClient();

  // Forwards the call to the
  // CallbackServer object
  int UsesCallback(char* pCallbackName);

  virtual int CallbackHandler(
  int callbackId, void* pData ) = 0;

private:
  CallbackServer*     pServer;
};
```

The generic code class must be derived publicly from the CallbackServer class and should register callbacks that it may make during its constructor:

```
class GenericCode : public CallbackServer
{
  int someCallbackID;
  int someOtherCallbackID;

  static char const * const
                    sc_someCallbackName;
 static char const * const
               sc_someOtherCallbackName;
   ...
};

GenericCode::GenericCode()
{
 someCallbackID =
  AddCallback(sc_someCallbackName);
 someOtherCallbackID =
  AddCallback(sc_someOtherCallbackName);
}
```

New callbacks can be added as they become necessary, and no changes to the public interface are necessary.

The client will own an opaque pointer to the object that provides the generic code. The client constructor should then call SetupCallback, and signal what callbacks it will process:

```
class Device : AbstractDevice,
               CallbackClient
{
  int someHandledCallbackID;

  static char const * const
            sc_someHandledCallbackName;

  GenericCodeHandle pImplementation;

  virtual int CallbackHandler(
         int callbackId, void* pData );

  ...
};

Device::Device()
{
 pImplementation->SetupCallback( this );

 someHandledCallbackID =
     pImplementation->UsesCallback(
           sc_someHandledCallbackName );
}
```

When the generic code object requires a specialisation it calls MakeCallback, using the ID of a callback that it has registered. The CallbackServer checks to see whether the client has signaled that it processes this callback, and if it has calls the callback handler that the client has provided:

```
int CallbackServer::MakeCallback( int
callbackId, void* pData )
{
  if ( callbackInUse[ callbackId ] )
  {
   return pTheClient(callbackId, pData);
  }
  return 0;
}
```

Finally, the client class (derived from CallbackClient) must also implement a callback handler function. This will be called whenever the GenericCode object makes a callback that the client has signaled that it will process (by calling UsesCallback).

```
// virtual
int Device::CallbackHandler( int
callbackId, void* pData )
{
  if(callbackID==someHandledCallbackID)
  {
    ...
  }
  else
  {
    // Some reserved value to indicate
    // an error
    return 0;
  }
}
```

We can now add specialisations at will to the generic code class. This will not impact on any existing client code, but can immediately be made use of by new code.

There are some obvious problems:

1. The callbacks are resolved by name. The usual problems of misspelling will not be picked up till runtime.

2. The final argument in a MakeCallback call (pData) is a void pointer. What this points to is callback dependent. If a server registers "Name" and a client uses "Name", but the pData points to a different type of data then all sorts of errors will ensue. An ‘argument’ base class and dynamic casting would improve the situation.

3. The return value from a callback is callback specific and is subject to the same consistency problems as 2. It can be improved in the same way.

Conclusion

It is always desirable to minimise dependencies, especially within large projects and designs need to reflect this. In some application domains it may not be possible to specify all future requirements and provision of a mechanism that helps cope with them will prove to be a good investment.

These callback classes allow the controlled, but potentially unsafe, introduction of new channels of communication between objects without affecting objects for which the existing interface is sufficient.

Seb Rose
seb@hoboco.scotborders.co.uk

Lakos 1996: Large Scale C++ Software Design, Addison Wesley 0-201-63362-0

Coplien 1992: Advanced C++ Programming Styles and Idioms, Addison Wesley 0-201-54855-0

editor << letters;

Reference Counting in basic_string

When using the C++ Standard Library implementation supplied with Visual C++ 4.2, the following code generates a memory leak.

```
int main()
{
  string s1("Hello"), s2("World");
  s1 = s2;
}
```

The problem seems to be in the string assignment operator, or rather in the `string::assign` function used to implement it. The relevant code is this:

```
template<class _E, class _TYPE, class _A>
class basic_string {
public:
    typedef basic_string<_E, _TYPE, _A> _Myt;
//...
    _Myt& operator=(const _Myt& _X) {return (assign(_X)); }
    _Myt& assign(const _Myt& _X) {return (assign(_X, 0, npos)); }

    _Myt& assign(const _Myt& _X, size_type _P, size_type _M)
        {if (_X.size() < _P)
            _Xran();
        size_type _N = _X.size() - _P;
        if (_M < _N)
            _N = _M;
        if (this == &_X)
            erase((size_type)(_P + _N)), erase(0, _P);
        else if (0 < _N && _N == _X.size()          // Line A
            && _Refcnt(_X.c_str()) < _FROZEN - 1
            && allocator == _X.allocator)
            {_Ptr = (_E *)_X.c_str();
            _Len = _X.size();
            _Res = _X.capacity();
            ++_Refcnt(_Ptr); }
        else if (_Grow(_N, true))
            {_TYPE::copy(_Ptr, &_X.c_str()[_P], _N);
            _Eos(_N); }
        return (*this); }
//...
};
```

This implementation uses a reference counting technique to optimise string copy operations. The `basic_string` class contains a pointer to a block of storage which contains a reference count, as well as the characters of the string. E.g. s1: pointer p -----> count = 1, text = "Hello" and s2: pointer p -----> count = 1, text = "World"

The assignment s1 = s2 can then be implemented like as: 1) Decrement s1's count. 2) If s1's count is now zero, destroy s1's storage area. 3) Set s1's pointer to point to s2's storage. 4) Increment s2's count.

[This implementation stores (count - 1) rather than count, but the principle is the same.]

The code provided by the library distinguishes two cases (at Line A). In the case where the two character sequences are the same size the `assign` function fails to release the storage for the left hand string. In all other cases, as far as I can see, new storage is allocated and the characters copied. So the efficient case doesn't work and the cases that work aren't efficient!

Has PJP made a mistake, or have I missed something?

Phil Bass
phil_bass@bio-rad.com

VC++4.2 Templates

If you compile the following code with Visual C++ 4.2 what would you expect to happen?

```
struct Base {};

template <class T>
class Derived : virtual public Base {};

namespace Debug
{
    typedef Derived<char> DebugDerived;
}

Derived<char> trace;
```

I bet you didn't predict this:

```
debug.cpp(15) : error C2039: 'Derived<char>' : is not a member of 'Debug'
debug.cpp(15) : error C2935: 'Derived<char>' : template-class-id redefined as a global
function
debug.cpp(15) : warning C4508: 'Derived<char>' : function should return a value; 'void'
return type assumed
```

Global function? But there are no parentheses in the code! The Borland compiler I had to hand made no complaint. Interestingly, the errors go away if you Remove the 'virtual' keyword, or make Derived an ordinary class instead of a template, or remove the namespace.

None of these options was acceptable to me, but I did find a work-around. Instead of

```
    typedef Derived<char> DebugDerived;
```

use

```
    struct DebugDerived : Derived<char> {};
```

Forwarded by Francis Glassborow
Original contributor unknown.
francis@robinton.demon.co.uk

News

European DevWeek 98, 23-27 February, London. C++, Java, VB, & Delphi training devweek@bearpark.co.uk

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

ACCU contact details.

See Overload Issue 22.

New look web site!... www.accu.org

Still being worked on, its facility to search the ACCU book reviews online has been very helpful.

C++ directory

If you've got the time to kill, you'll find some new C++ information here.

www.yahoo.com/Computers/Programming_Languages/C_and_C__

To see the STL information, go to the above link and choose C++ / Class Libraries / Standard Template Library (STL).

Standard Template Library (STL).

Although STL isn't new, it is new ground for many people. Some of the STL's background is given in an interview of Alexander Stepanov.

http://www.metabyte.com/~fbp/stl/StepanovUSA.html

There are no "Learn STL in 21 days" books (yet). Some STL books have been recommended by people who use it.

As an introduction. "STL for C++ programmers" by Leen Ammeraal, published by Wiley (£27.50). The author has an FTP site, ftp://ftp.expa.fnt.hvu.nl/pub/ammeraal, with the latest version of the book's source code in stlcpp.zip.

As a good (but rather technical) reference, "STL Tutorial & Reference Guide" by David Musser, published by Addison-Wesley (£31). David Musser's website has a very useful STL page (www.cs.rpi.edu/~musser/stl.html).

STL tutorials.

Although there are some STL tutorials on the internet, I don't know enough about STL to comment on them (I ordered an STL book in November from a book page and it still hasn't arrived yet - I'd recommend using the dedicated book suppliers that give ACCU members free post - the books actually arrive). Whenever the book arrives, I suppose I'll still be making my way through "The C++ Programming Language" (3e).

The Silicon Graphics web site has some good STL documentation (www.sgi.com/Technology/STL) as well as a public domain implementation of (thread safe) STL that can be downloaded.

Next issue... Software Engineering.

Next month I'll cover Software Engineering web sites.

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* 25 should be submitted to the editor, John Merrells < merrells@netscape.com>, by March 11th.