contents

credits & contacts

Overload Editor:
John Merrells
overload@accu.org

Contributing Editor:
Alan Griffiths
alan@octopull.demon.co.uk

Readers:

Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Chris Lowe
ads@accu.org

Overload is a publication of the ACCU. For details of the ACCU and other ACCU publications and activities, see the ACCU website.

ACCU Website:
http://www.accu.org/

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial - Ruminations on "A little story"

It's now three years since I joined the Overload editorial panel. In those three years I've spent my time working on articles – getting people to write them, shepherding authors through the process of writing them. Recently, I've turned my attention more to writing, but this is my first attempt at writing the editorial. So, what to write about? I turned to `accu-general` for inspiration.

Despite not posting very much on this mailing list, I do lurk on it a lot – one of the things I like about it is the culture that allows one topic of discussion to lead to another. From time to time something is said that prompts an original short post to lead to a large thread covering a much broader area of subject matter, or even ending up on a totally different topic to the one that started it.

The thread entitled "A little story..." caught my attention. As I write this, my email client contains 122 posts on this thread! It started with a short tale about a chance encounter with a programmer who had given up reading books because he believed he knew how to do his job, and had no need for furthering his knowledge. It led to (among other things), a discussion of the different interests of "techies" and managers. During this thread, the "shiny new hammer" expression – that refers to the desire to use the latest feature or technique learned, regardless of whether or not it is appropriate – came up, as it has in previous threads. As was the case in the "A little story..." thread, often mentioned within close proximity of the "shiny new hammer" expression, is *C++ template meta-programming*. Fair enough, I don't recall the latter being mentioned directly as an example of the former on the "A little story..." thread, but it certainly has in `accu-general` threads in the past.

C++ template meta-programming – which I'm going to call *TMP* for short – is indeed greeted with enthusiasm by some, but there are those who regard it as something not suitable for practical use. A couple of conferences ago, having admitted to using TMP in production code, I was involved in a debate with someone who argued the case of it being unsuitable for production use on the grounds that most developers would not understand the technique, and therefore it would prejudice the chances of the software being maintainable in the long term. My counter-argument was that the TMP approach was actually the *best* chance for maintainability. I chose to use a TMP approach believing it to offer the best of the available sets of tradeoffs – a decision I stand by to this day!

Now, this editorial is not about C++ TMP per se – that is the subject of technical articles within Overload, not a subject for the editorial. However, I think it is reasonable to assert that it is an example of an advanced approach – and here, *advanced* is the critical word. It is the use and/or abuse of *advanced* approaches and techniques in general that is often the substance of (sometimes heated) debate, and here I want to use TMP as a topical example of an advanced approach.

What I want to talk about centres around the interaction between the commercial and managerial forces acting on development, and the use of advanced techniques. I hope to be able to argue, using TMP as an example, that the use of advanced techniques is not in tension with the commercial interests, and therefore is not in tension with the interests of managers.

Solutions to some problems have *naturally occurring* complexity. That is, a certain amount of complexity is inherent in any solution anyone might come up with – and it is the failure to recognise this that perpetuates its own set of problems and leads to trouble in software development in general. Complexity exists and it can't be ignored or got rid of, so the only choice is to live with it and do what we can to tame it. How can we go about doing this when designing software? Three options spring immediately to mind: ignore the complexity, produce lots of code to handle it, or use advanced techniques (associated with advanced features of the implementation language) to tame it. The first of the three options has, unfortunately, a large following. I think it can be safely put in the rubbish bin without further discussion. The second option (write lots of code) is quite common, and may be necessary, depending on the nature of the complexity. In passing, this reminds me of an `accu-general` thread from a few years ago (I *think* it was 1999, but that's without checking). It was posted by someone telling the tale of how his management had banned the use of C++ templates – management believed that if developers weren't allowed to use advanced language features, they wouldn't write complicated code. Management's ears remained closed to any discussion despite the developers being quite happy with templates. The code base became more and more complex because of the bloat caused by having to repeat code.

So far I have talked about complexity without defining what I actually mean. This is important – is a piece of software complex because it uses complicated algorithms, because it contains lots of classes, because it requires the integration of components developed independently by different parties, for some other reason, or because of some combination of these? However, there is a type of complexity that arises where there is more than one class having between them commonality and variability that are well defined. The variability can involve pretty much any part of a class's implementation – constants, types, data structures, operations and algorithms (actually, the variability can even extend to the interfaces, but I'm in serious danger of digressing).

Having got that part straight, let's get back to approaches to addressing complexity. Writing lots of code may be necessary for some types of complexity, but for the one I've just described it leads to repetition of code. Optically the code may look different because its implementation contains different types and different operations, but on closer scrutiny it can be seen that the structure is the same. Therefore there is a large area of commonality that requires maintenance and testing – or putting it another way, the same thing is being maintained and tested more than once!

By contrast, if the same code had been generated from a meta-program, the meta-code would express a clear separation of concerns between the commonality and variability. Therefore, the two could be maintained independently, and at least some of the testing done separately. This sounds good, doesn't it, but is there a downside? Well, there may be, but it's a matter of perspective. The downside that may be perceived is that TMP uses templates, regarded by some as an advanced language feature, and one to be used by experts rather than ordinary programmers. Further, TMP uses advanced techniques on top of an advanced language feature, and is therefore highly complicated.

So far I have focused on the technical topics of discussion, but now it's time to put our project management hats on, and I think this is a good point at which to insert a short digression – I want to briefly make a general comment on something else that came up in the "A little story..." thread, namely the interests of project managers. Some people reading this will know that my interests lie mainly in designing and implementing software – i.e. I'm a developer (and I have in the past been called a "techie", but "techie" is a characterisation I'm not at all keen on). However, on a recent project, I found my role somewhat extended by force. I think the following description using metaphors is fair: if project management can be likened to a swimming pool, then I had to dip my foot in the shallow end. Now, previously I thought I was aware of project management issues and considerations through having read about it and talked/listened to other people. I was in for a surprise – the first hard lesson I learned was that project management looks very different when you're actually doing the job! Having done just a small amount of project management was a serious education, and has given me a broader perspective on software development. OK, digression over, let's get back on topic...

Project managers have to balance several forces, but two of the principal ones are the minimisation of risk and keeping costs under control. Note that the latter is not a case of actually minimising cost – that's different, but the point is, when the costs are added up, the total must be within budget. Risks however, must always be minimised. Risk minimisation and cost control are not independent forces – there is interplay between them. Now before bringing project management into the discussion, I was talking about handling a certain type of complexity either by writing and replicating lots of code, or by using C++ language features to absorb it by using a TMP approach. It's time for these two separate topics to meet up.

If the approach of writing – and replicating – lots of code has been used, then each piece of (what is effectively the same) code must be maintained. This increases the risk, because there is a risk that code will get missed in maintenance, and this risk must be addressed. One way to address it is ensure that maintenance work is properly reviewed, but this takes people's time and pushes up the cost. Further, each piece of code must be tested, and testing more than one piece of code costs more than testing just one. This is an ongoing cycle for as long as the software is being maintained. If instead the TMP approach is used, only one piece of code needs to be tested and maintained. It may seem that other risks creep in, namely that there is a burden placed on the knowledge of the developers that write and maintain the code. Whether or not this is really a problem is debatable, but one thing is certainly true – lack of good developers always increases the risk dramatically! The TMP approach actually has a big plus point that is far from obvious, but reduces both risk and cost – the commonality and variability is *documented* without a word ever being written!

Software developers have a responsibility to apply the use of advanced techniques for the benefit of projects, and not for the sake of using them – if we expect managers to take us seriously, we must conduct ourselves professionally, and wielding a "shiny new hammer" has no place in *professional* software development. Managers have a responsibility to make sure that the software developers working on the projects they manage are competent professionals, and to understand that in the hands of competent professionals, advanced techniques are of benefit to their projects. Unfortunately, while the industry has few software developers who can be described as competent professionals, it is sad that it also has few managers who fit that description.

Mark Radford
mark@twonine.demon.co.uk

Copy Deadlines

All articles intended for publication in *Overload 60* should be submitted to the editor by March 1st 2004, and for *Overload 61* by May 1st 2004.

Letters to the Editor(s)

Regarding “A more flexible container” by Rich Sposato, Overload 58 (December 2003)

Although it is certainly interesting to consider alternative implementations of standard containers, I think Rich has chosen the wrong solution to the problem.

What he want to do is to extract all members of a collection satisfying a given condition. In his driving example, he has a set of Employee records, and he wants to extract those elements of the container that matches a given surname, say. If one only ever wants to deal with sets of such records, Rich’s solution is probably the best one can do. But if one later wants, as Rich does in the article, to perform a similar operation on another type of container, the limitations of his approach become clear.

Instead of modifying each container by defining a new one built upon the standard containers, it is much better to have a generic algorithm working with any container (standard or not).

So it seems to me, a better, or at least more generic, solution would be to define an algorithm working roughly like this:

```
result = filter(collection,comparator);
  // filter elements of a container
```

where `result` and `collection` are containers of the same kind and `comparator` is a function, or functor, taking an element and returning a Boolean value.

A template version would look something like this (in pseudo-C++ to show the structure more clearly)

```
template<class Coll, class Comp>
Coll filter(Coll coll, Comp cmp) {
  Coll res(0);  // create empty result set
  for (iterator it=coll.begin();
       it != coll.end();
       ++it)
    if (cmp(it))
      res.insert(it);
  return res;
}
```

This will then work for any container having a standard iterator interface and an `insert()` member function as well as for any `cmp` having an `operator()` defined taking a container element of the right type as argument and returning something which can be interpreted as a Boolean.

All the extra work now goes into defining the comparator function or functor, which is probably precisely where the effort should be concentrated anyway.

While this solution is simpler and more generic than Rich’s, it is not purely object oriented. In fact, it wouldn’t be possible to carry it over directly to Java or C#. Instead, it is an example of what one might call a “functional programming design pattern”. The filter algorithm above is a C++ analogue of a so-called “higher order function” in functional programming (FP), i.e. a function taking another function as argument. Actually, very often OOP design patterns can be translated into FP higher order functions.

The particular higher order function used here, `filter`, exists in all major FP languages (Lisp, Haskell, ML) and in some other multi-paradigm ones (Perl, Python).

What this illustrates is the idea that, however powerful a concept, OOP just isn’t the right solution in all circumstances. Luckily, C++ is a multi-paradigm language, supporting not just OOP and C-style procedural programming, but also FP-style programming as above. That being said, C++ has a very, very strong support for efficient OOP code, allowing most compilers to make very efficient assembler code. In contrast, the FP-aspects, although introduced with the STL, have received less attention, and not all compilers will generate efficient code in these cases. This, however, may be compensated by the optimisations made for the standard containers

On the other hand, such an FP-like solution is easier to maintain, the code is concentrated in one place instead of being duplicated in all containers. Moreover, the FP-like solution above is more generic. Suppose, for instance, that you defined an iterator interface for files, with the “elements” being the individual lines. The filter-algorithm can then be turned into a utility like UNIX’s `grep`, extracting lines meeting a specific criterion.

Whether to put something into a member function or not is a difficult question to answer in general. All sorts of issues may be involved: design principles, maintainability, and efficiency etc. My own pragmatic rule of thumb is, if the same code is copied with very little change in many classes, it is probably worth considering putting it outside, either in a special class or as an algorithm.

Frank Antonsen
frankantonsen@netscape.net

Rich Sposato’s Reply

My article had two purposes. One was to create associative containers that allow searching using a type other than the key type. The other goal was to remove the unnecessary objects used to search through the standard associative containers. Creating a generic algorithm to find all elements of any container type that satisfy a predicate was not my intention. Indeed, such an algorithm is missing from the STL, but I will get to that later. First, let us look at the associative containers and the generic search algorithms.

The STL’s stand-alone search (`binary_search`, `lower_bound`, `upper_bound`, and `equal_range`) algorithms provide logarithmic complexity only for random-access iterators, and linear complexity for all other iterator types. The standard associative containers are typically implemented using red-black binary trees, and with their bi-directional iterators, they are not served well by these functions. I find it ironic that associative containers were designed for logarithmic complexity but their iterators only provide linear complexity with the STL algorithm functions, while three of the sequential containers (`vector`, `string`, and `deque`) have a linear internal arrangement, but their iterators provide logarithmic complexity with the same STL algorithms. The STL’s stand-alone `count` and `find` functions provide linear complexity for all iterator types. So that irony does not apply to these functions.

These associative containers have their own `find`, `count`, `lower_bound`, `upper_bound`, and `equal_range` member functions, because as members, they can use the internal structure of the containers to provide greater efficiency than linear complexity. The `set`’s operations require only logarithmic complexity. These functions should be used instead of the generic

More is Less

by Thaddaeus Frogley

When it comes to optimisation the established wisdom amongst "serious" programmers these days seems to be:

1 Don't do it
2 (For experts only) Don't do it yet [1].

Fortunately established good practice tends to avoid "pessimisation". Sometimes, however, it is necessary to go one step further, and actually perform some optimisations, based of course on careful measurements. In this article I will show two methods for streamlining object creation semantics, preventing the compiler from generating code that is not needed, starting with established good practice, and going on to a slightly unconventional technique that can be used when dealing with less conventional situations. I will also be describing an effective technique for optimising routines where value types are created on the stack as temporary storage during repeated calculations.

Using initialiser lists

Construction should always result in an object that is in a valid state – i.e. is initialised. Use of types that require explicit (multi stage) initialisation is more error prone than ones that do not, and should therefore be avoided. It is also generally advisable to provide a default constructor for value types as value types should generally be usable within a system in the same way as you might use built in types such as `int`, `float`, and `char`.

The best way to initialise / construct an object in C++ is via the initialiser list syntax[2]. From the perspective of high performance computing, not using the initialiser list syntax can and will result in double intialisation – more code will be generated and executed, resulting in a larger, slower executable – potentially more than double for many layers of aggregate types.

For example, starting from the ground up:

```
struct A {
  inline A() {
    i = 0;
  }
  //...
  int i;
};
```

Does what it looks like it does, because `i` is a built in type (`int`) and built in types do not have default initialisation.

```
struct B {
  inline B() {
    a.i = 1;
  }
  //...
  A a;
};
```

We now have double intialisation of the integer `i`, because on entry to the body of the constructor, all the members and base classes have already been constructed. Thus the code that is executed is as follows[1]:

```
  B b;
0040102B mov    dword ptr [b],0
00401032 mov    dword ptr [b],1
```

For readers unfamiliar with the instruction set used here, the instructions above set a location in memory to 0 (zero) *and then set the same location in memory to 1 (one)*. By providing a specific constructor for the type A, and using it in the initialiser

[continued on next page]

stand-alone functions by the same names. (See item #44 in Scott Meyer's book: "Effective STL".)

Frank Antonsen's example for filtering elements from a container is not the most generic solution for C++. His `filter` function requires the collection to have a constructor that accepts a single integer parameter; a constraint not met by five STL containers. This example is not truly container-independent code. (See item #2 in Scott Meyer's book: "Effective STL".) By acting on a container rather than an iterator, the filter will not work with arrays - a requirement of the STL algorithms. This points out another irony: algorithms that act only on iterators are more likely to provide "container-independent" code than algorithms that act on containers. As Raoul Gough eloquently explained in his article ("Choosing Template Parameters" also in Overload issue #58), picking the correct template parameter types can make all the difference between a flexible and inflexible class or function.

Using the `std::copy` function as an example, we can make a generic filter named `copy_if` that uses a predicate. The filtering function will accept an iterator type as a template parameter, and use the `typename` keyword instead of the `class` keyword within the template specification. Since `copy_if` will always have linear complexity no matter which type of iterator or container uses it, developers might be better off using `std::set::equal_range` and then calling `std::copy`. The `copy_if` algorithm fulfills all the abilities of Frank's `filter` function with fewer constraints.

The `copy` function's signature is:

```
template< typename InputIter,
          typename OutputIter >
OutputIter copy(InputIter first,
                const InputIter& last,
                OutputIter output );
```

The `copy_if` function would have a signature of:

```
template< typename InputIter,
          typename OutputIter,
          typename Predicate >
OutputIter copy_if(InputIter first,
                   const InputIter& last,
                   OutputIter output,
                   Predicate pred );
```

Alas, the STL has no `copy_if` function, although it has `copy` and `copy_backward`. This omission becomes more obvious when considering that other functions such as `find`, `count`, `remove`, and `remove_copy` have predicated versions named `find_if`, `count_if`, `remove_if`, and `remove_copy_if`. The implementation for `copy_if` is trivial, and in a time-honoured tradition, I shall leave that as an exercise to the reader.

Other languages may have filter functions that act as Frank described them, but my article was not about making generic filters for C++ or any other language.

Rich Sposato
rds@richsposato.com

[continued from previous page]
list of the constructor B the unnecessary memory write operation can be avoided:

```
struct A {
  //...
  inline A(int a)
    : i(a) { }
  //...
};
struct B {
  inline B()
    : a(1) { }
  //...
};
```

By specifically initialising the member a we avoid assignment to it in the body of the constructor and the code generated by the compiler is as follows:

```
  B b;
0040102B mov    dword ptr [b],1
```

Unfortunately using initialiser lists is not always appropriate or desirable. There are times when setting the initial state of an object is non trivial and cannot be achieved within an initialiser list, or to do so would result in complex and difficult to maintain code.

Using the "uninitialised" constructor

The solution to this problem – of wanting to perform initialisation within the body of the constructor without paying the runtime cost of default initialisation – relies on the functionality provided by the members or base classes in the same way as the initialiser lists shown above. In this case though it is necessary for the component parts of the class to provide a mechanism for constructing an instance that is not initialised. This is where we need to write code that very deliberately does nothing at all:

```
struct UnInitialised {} uninitialised;
                     // See footnote 2
struct A {
  //...
  inline A(const UnInitialised&) { }
  //...
};
```

Now any class that uses instances of type A is free to defer initialisation to the body of the constructor, like so:

```
struct B {
  inline B()
    : a(uninitialised) {
    a.i = 1;
  }
  //...
};
```

This, as expected, generates machine code that is exactly the same as the version that used the initialiser list directly:

```
  B b;
0040102B mov    dword ptr [b],1
```

This use of function overloading is a technique known as Tags (or Type Tags), and sometimes Discriminators, and is a degenerate case of of the traits technique where the trait type has no associated behaviour and no type parameters[3]. There are many examples of the traits technique throughout the standard library: `allocator<>`, `char_traits<>`, etc. Most directly similar to this is the use of `std::nothrow`.

In this case the type of the parameter is selecting a constructor implementation that does not initialise the object, allowing the programmer who uses that type to deliberately select a course of action that would not normally be desirable.

Even as an optimisation technique leaving an object uninitialised can be counterproductive if doing so may have a negative impact on the implementation of the rest of the class. I would recommend that this facility only be provided for value classes, by which I mean classes that represents a value or set of values, such as a vector or matrix, and would typically be used in the same way as any of the built in numerical types (`int`, `float`, etc).

As a technique for optimising routines where a value type is created on the stack as temporary storage during repeated calculations, this is extremely effective, and can be shown to be more efficient than using a `static` variable. When a function level `static` variable is used the compiler must generate both the code to construct it, and a test to see if it is already constructed. For operations executed many thousands or millions of times in succession the extra memory accesses and code branches can have a surprisingly negative effect on the performance of the processor level cache, and the time it takes for the operations to complete.

Dealing with (illegal) unions

I was recently given the task of optimising a math library that used unions to provide a flexible interface. While this was not standard conforming C++[3] it worked on the target compilers, with the exception of the construction behaviour. It was found that the constructors for each of the members of the union where executed, causing the memory to be initialised many times over, something that became a performance issue for the project. By using the "uninitialised" constructor technique, I was able to get this unwanted behaviour under control. Some before and after comparisons are shown in Figure 1 and Figure 2.

Figure 1 shows the source code and compiler output for a constructor for a 4x3 matrix class that contains an illegal union. The union contains two structs, one with a 3x3 matrix and a 3D vector, one with four 3D vectors. The 3x3 matrix is itself made up of three 3D vectors. In this particular constructor the four vectors are being initialised on construction via the initialiser list, but the matrix /

1 The compiler output examples shown in the first half of the article were compiled using MS VC++ .NET, targeting Intel/Win32, using a debug build, with inlining turned on. For an example that trivial an optimised build will eliminate the example class completely. In testing GCC exhibited equivalent code generation.

The compiler output shown in Figures 1 and 2 were compiled by MS VC++ .NET 2003 targeting XBOX (with SSE instruction set disabled). A MIPS version was also compiled using CodeWarrior for PS2 R3.5, and was also shown to have equivalent code generation.

Note also that it is important that the *uninitialised* constructors be inline, and that the `inline` keyword is not being ignored by the compiler, otherwise any gains seen in not initialising the member twice could be minimal compared to the overhead of a function call.

2 An alternative to using an instance of an empty `struct` is to use an `enum`:

```
enum UnInitialised {
  uninitialised
};
```

Both techniques have advantages and disadvantages. Where an empty `struct` or `class` is used an instance of that type must also be provided in at least one translation unit. Where an `enum` is used one must be aware of its silent conversion to an `int`, and any risks this may introduce.

3 A union containing non trivial types – including any type with a constructor – is ill-formed, and creating an instance of such a union leads to undefined behaviour. C++ Standard, 9.5/1.

vector pair are not explicitly initialised, and so their default constructors are being called. By specifically triggering the "uninitialised" constructor for the matrix / vector pair (shown in the right hand column) the redundant code generation is suppressed. Note that the disassembly shown in this and the following example were generated by a compiler configured to perform full optimization.

In Figure 2, we see an even more dramatic improvement, in the implementation of a copy constructor. In this example the initialiser list was not in the first place used at all, and so before the assignment in the body of the constructor was executed all the elements of the union are first default initialised. In the improved version, shown in the right hand column, the copy is implemented via the rotation (matrix) and translation (vector) copy constructors, and the unwanted default constructors (from the second elements of the union) are suppressed via the use of the uninitialised constructor.

Thaddaeus Frogley
t.frogley@ntlworld.com

This article is also available at :
http://thad.notagoth.org/more_is_less/

References and Further Reading

[1] Michael Jackson, 1975, *Principles of Program Design*
[2] Scott Meyers, 1997, *Effective C++ (Item 12)*
[3] Hubert Matthews, 2003 (`accu-general`)
Andrei Alexandrescu, *Modern C++ Design*
Czarnecki & Eisenecker, *Generative Programming*
Jim Coplien, *Multi-Paradigm Design for C++*

With Thanks To

The Overload team (especially Phil Bass for some invaluable feedback), Tobias Sicheritz, and the `accu-general` mailing list (especially Ric Parkin, Alan Stokes, Hubert Matthews, David Nash, Paul Grenyer, Jon Jagger, & Kevlin Henney) and everyone from HuSi who gave me feedback on this article.

Before

```
template
M_Matrix4x3Template::M_Matrix4x3Template(
  const M_Vector3Template &i,
  const M_Vector3Template &j,
  const M_Vector3Template &k,
  const M_Vector3Template &l )
    : x(i)
    , y(j)
    , z(k)
    , t(l)
{
00343358  push  ebp
00343359  mov   ebp,esp
0034335B  push  ecx
0034335C  push  ebx
0034335D  push  esi
0034335E  push  edi
0034335F  mov   ebx,ecx
00343361  call  M_Matrix3Template
                        ::getIdentity (134E5h)
00343366  fld   dword ptr [eax]
00343368  fstp  dword ptr [ebx]
0034336A  lea   ecx,[ebx+0Ch]
0034336D  fld   dword ptr [eax+4]
00343370  mov   esi,dword ptr [i]
00343373  fstp  dword ptr [ebp-4]
00343376  lea   edx,[ebx+18h]
00343379  fld   dword ptr [ebp-4]
0034337C  mov   edi,ebx
0034337E  fstp  dword ptr [ebx+4]
00343381  fld   dword ptr [eax+8]
00343384  fstp  dword ptr [ebp-4]
00343387  fld   dword ptr [ebp-4]
0034338A  fstp  dword ptr [ebx+8]
0034338D  fld   dword ptr [eax+0Ch]
00343390  fstp  dword ptr [ebp-4]
00343393  fld   dword ptr [ebp-4]
00343396  fstp  dword ptr [ecx]
00343398  fld   dword ptr [eax+10h]
0034339B  fstp  dword ptr [ebp-4]
0034339E  fld   dword ptr [ebp-4]
003433A1  fstp  dword ptr [ebx+10h]
003433A4  fld   dword ptr [eax+14h]
003433A7  fstp  dword ptr [ebp-4]
003433AA  fld   dword ptr [ebp-4]
003433AD  fstp  dword ptr [ebx+14h]
003433B0  fld   dword ptr [eax+18h]
003433B3  fstp  dword ptr [ebp-4]
003433B6  fld   dword ptr [ebp-4]
003433B9  fstp  dword ptr [edx]
003433BB  fld   dword ptr [eax+1Ch]
003433BE  fstp  dword ptr [ebp-4]
003433C1  fld   dword ptr [ebp-4]
003433C4  fstp  dword ptr [ebx+1Ch]
003433C7  fld   dword ptr [eax+20h]
003433CA  lea   eax,[ebx+24h]
003433CD  fstp  dword ptr [ebp-4]
003433D0  fld   dword ptr [ebp-4]
003433D3  fstp  dword ptr [ebx+20h]
003433D6  fldz
003433D8  fstp  dword ptr [eax]
003433DA  fldz
003433DC  fstp  dword ptr [eax+4]
003433DF  fldz
003433E1  fstp  dword ptr [eax+8]
003433E4  movs  dword ptr [edi], dword ptr [esi]
003433E5  movs  dword ptr [edi], dword ptr [esi]
003433E6  movs  dword ptr [edi], dword ptr [esi]
003433E7  mov   esi,dword ptr [j]
003433EA  mov   edi,ecx
003433EC  movs  dword ptr [edi], dword ptr [esi]
003433ED  movs  dword ptr [edi], dword ptr [esi]
003433EE  movs  dword ptr [edi],dword ptr [esi]
003433EF  mov   esi,dword ptr [k]
003433F2  mov   edi,edx
003433F4  movs  dword ptr [edi],dword ptr [esi]
003433F5  movs  dword ptr [edi],dword ptr [esi]
003433F6  movs  dword ptr [edi],dword ptr [esi]
003433F7  mov   esi,dword ptr [l]
003433FA  mov   edi,eax
003433FC  movs  dword ptr [edi],dword ptr [esi]
003433FD  movs  dword ptr [edi],dword ptr [esi]
003433FE  movs  dword ptr [edi],dword ptr [esi]
003433FF  pop   edi
00343400  pop   esi
}
00343401  mov   eax,ebx
00343403  pop   ebx
00343404  leave
00343405  ret   10h
```

After

```
template
M_Matrix4x3Template::M_Matrix4x3Template(
  const M_Vector3Template &i,
  const M_Vector3Template &j,
  const M_Vector3Template &k,
  const M_Vector3Template &l )
    : rotation(uninitialized)
    , translation(uninitialized)
    , x(i)
    , y(j)
    , z(k)
    , t(l)
{
00343358  push   esi
00343359  mov    esi,dword ptr [esp+8]
0034335D  push   edi
0034335E  mov    eax,ecx
00343360  mov    edi,eax
00343362  movs   dword ptr [edi],dword ptr [esi]
00343363  movs   dword ptr [edi],dword ptr [esi]
00343364  movs   dword ptr [edi],dword ptr [esi]
00343365  mov    esi,dword ptr [esp+10h]
00343369  lea    edi,[eax+0Ch]
0034336C  movs   dword ptr [edi],dword ptr [esi]
0034336D  movs   dword ptr [edi],dword ptr [esi]
0034336E  movs   dword ptr [edi],dword ptr [esi]
0034336F  mov    esi,dword ptr [esp+14h]
00343373  lea    edi,[eax+18h]
00343376  movs   dword ptr [edi],dword ptr [esi]
00343377  movs   dword ptr [edi],dword ptr [esi]
00343378  movs   dword ptr [edi],dword ptr [esi]
00343379  mov    esi,dword ptr [esp+18h]
0034337D  lea    edi,[eax+24h]
00343380  movs   dword ptr [edi],dword ptr [esi]
00343381  movs   dword ptr [edi],dword ptr [esi]
00343382  movs   dword ptr [edi],dword ptr [esi]
00343383  pop    edi
00343384  pop    esi
}
00343385  ret    10h
```

For full-sized asm listing see:
http://thad.notagoth.org/more_is_less/figure1.html

Figure 1: Constructor

Before

```
// copy constructor
template
M_Matrix4x3Template::M_Matrix4x3Template(
  const M_Matrix4x3Template &m )
{
000514C3  push       ebp
000514C4  mov        ebp,esp
000514C6  push       ecx
000514C7  push       esi
000514C8  mov        esi,ecx
000514CA  call       M_Matrix3Template
                      ::getIdentity (134E5h)
000514CF  fld        dword ptr [eax]
000514D1  fstp       dword ptr [esi]
000514D3  fld        dword ptr [eax+4]
000514D6  fstp       dword ptr [ebp-4]
000514D9  fld        dword ptr [ebp-4]
000514DC  fstp       dword ptr [esi+4]
000514DF  fld        dword ptr [eax+8]
000514E2  fstp       dword ptr [ebp-4]
000514E5  fld        dword ptr [ebp-4]
000514E8  fstp       dword ptr [esi+8]
000514EB  fld        dword ptr [eax+0Ch]
000514EE  fstp       dword ptr [ebp-4]
000514F1  fld        dword ptr [ebp-4]
000514F4  fstp       dword ptr [esi+0Ch]
000514F7  fld        dword ptr [eax+10h]
000514FA  fstp       dword ptr [ebp-4]
000514FD  fld        dword ptr [ebp-4]
00051500  fstp       dword ptr [esi+10h]
00051503  fld        dword ptr [eax+14h]
00051506  fstp       dword ptr [ebp-4]
00051509  fld        dword ptr [ebp-4]
0005150C  fstp       dword ptr [esi+14h]
0005150F  fld        dword ptr [eax+18h]
00051512  fstp       dword ptr [ebp-4]
00051515  fld        dword ptr [ebp-4]
00051518  fstp       dword ptr [esi+18h]
0005151B  fld        dword ptr [eax+1Ch]
0005151E  fstp       dword ptr [ebp-4]
00051521  fld        dword ptr [ebp-4]
00051524  fstp       dword ptr [esi+1Ch]
00051527  fld        dword ptr [eax+20h]
      *this = m;
0005152A  mov        eax,dword ptr [m]
0005152D  fstp       dword ptr [ebp-4]
00051530  fld        dword ptr [ebp-4]
00051533  fstp       dword ptr [esi+20h]
00051536  fldz
00051538  fstp       dword ptr [esi+24h]
0005153B  fldz
0005153D  fstp       dword ptr [esi+28h]
00051540  fldz
00051542  fstp       dword ptr [esi+2Ch]
00051545  fldz
00051547  fstp       dword ptr [esi]
00051549  fldz
0005154B  fstp       dword ptr [esi+4]
0005154E  fldz
00051550  fstp       dword ptr [esi+8]
00051553  fldz
00051555  fstp       dword ptr [esi+0Ch]
00051558  fldz
0005155A  fstp       dword ptr [esi+10h]
0005155D  fldz
0005155F  fstp       dword ptr [esi+14h]
00051562  fldz
00051564  fstp       dword ptr [esi+18h]
00051567  fldz
00051569  fstp       dword ptr [esi+1Ch]
0005156C  fldz
0005156E  fstp       dword ptr [esi+20h]
00051571  fldz
00051573  fstp       dword ptr [esi+24h]
00051576  fldz
00051578  fstp       dword ptr [esi+28h]
0005157B  fldz
0005157D  fstp       dword ptr [esi+2Ch]
00051580  fld        dword ptr [eax]
00051582  fstp       dword ptr [esi]
00051584  fld        dword ptr [eax+4]
00051587  fstp       dword ptr [esi+4]
0005158A  fld        dword ptr [eax+8]
0005158D  fstp       dword ptr [esi+8]
00051590  fld        dword ptr [eax+0Ch]
00051593  fstp       dword ptr [esi+0Ch]
00051596  fld        dword ptr [eax+10h]
00051599  fstp       dword ptr [esi+10h]
0005159C  fld        dword ptr [eax+14h]
0005159F  fstp       dword ptr [esi+14h]
000515A2  fld        dword ptr [eax+18h]
000515A5  fstp       dword ptr [esi+18h]
000515A8  fld        dword ptr [eax+1Ch]
000515AB  fstp       dword ptr [esi+1Ch]
000515AE  fld        dword ptr [eax+20h]
000515B1  fstp       dword ptr [esi+20h]
000515B4  fld        dword ptr [eax+24h]
000515B7  fstp       dword ptr [esi+24h]
000515BA  fld        dword ptr [eax+28h]
000515BD  fstp       dword ptr [m]
000515C0  fld        dword ptr [m]
000515C3  fstp       dword ptr [esi+28h]
000515C6  fld        dword ptr [eax+2Ch]
}
000515C9  mov        eax,esi
000515CB  fstp       dword ptr [m]
000515CE  fld        dword ptr [m]
000515D1  fstp       dword ptr [esi+2Ch]
000515D4  pop        esi
000515D5  leave
000515D6  ret        4
```

After

```
// copy constructor
template
M_Matrix4x3Template::M_Matrix4x3Template(
  const M_Matrix4x3Template &m )
    : rotation(m.rotation)
    , translation(m.translation)
    , x(uninitialized)
    , y(uninitialized)
    , z(uninitialized)
    , t(uninitialized)
{
00048EB0  push   esi
00048EB1  push   edi
00048EB2  mov    eax,ecx
00048EB4  mov    ecx,dword ptr [esp+0Ch]
00048EB8  mov    esi,ecx
00048EBA  mov    edi,eax
00048EBC  movs   dword ptr [edi],dword ptr [esi]
00048EBD  movs   dword ptr [edi],dword ptr [esi]
00048EBE  movs   dword ptr [edi],dword ptr [esi]
00048EBF  lea    esi,[ecx+0Ch]
00048EC2  lea    edi,[eax+0Ch]
00048EC5  movs   dword ptr [edi],dword ptr [esi]
00048EC6  movs   dword ptr [edi],dword ptr [esi]
00048EC7  movs   dword ptr [edi],dword ptr [esi]
00048EC8  lea    esi,[ecx+18h]
00048ECB  lea    edi,[eax+18h]
00048ECE  movs   dword ptr [edi],dword ptr [esi]
00048ECF  movs   dword ptr [edi],dword ptr [esi]
00048ED0  movs   dword ptr [edi],dword ptr [esi]
00048ED1  lea    esi,[ecx+24h]
00048ED4  lea    edi,[eax+24h]
00048ED7  movs   dword ptr [edi],dword ptr [esi]
00048ED8  movs   dword ptr [edi],dword ptr [esi]
00048ED9  movs   dword ptr [edi],dword ptr [esi]
00048EDA  pop    edi
00048EDB  pop    esi
      //*this = m;
}
00048EDC  ret    4
```

For full-sized asm listing see:
http://thad.notagoth.org/more_is_less/figure2.html

Figure 2: Copy Constructor

Why do requirements change?

by Allan Kelly

Stable requirements are the holy grail of software development. (McConnell, 1993)

Once upon a time stable requirements were seen as a pre-requisite for starting a software development project. There may be a few Civil Servants who still believe this, but many in the IT world have given up looking for the Holy Grail of stable requirements[1].

Changing requirements have become an accepted fact of life for software developers, indeed, most of the process and methodology books now come with subtitles like "Embracing change". But how many of us stop and think about why requirements change?

I've been giving some thought to this question for a while now, and I've come up with some reasons why I think requirements change. I'm not saying this is an exhaustive list, but it is a list that makes sense to me based on my own experience and the way I view software development.

Why do we fail to capture requirements?

Perhaps the most obvious reason that requirements change is that we fail to capture them to start with. Someone writes down "black" when they should have written "blue." Everyone makes mistakes from time to time, and a small mistake by a business analyst can easily go unnoticed for months. Sure, we have document reviews to catch this kind of thing, but such mistakes are easily missed in a 100 page tome.

There are lots of opportunities for mistakes in the requirements capture phase, and not all of them are because some people are better than others. At first we need to comprehend the requirements, then we need to capture them and communicate them. Usually this is done with a text document. Mistakes can arise at any point: comprehension, recording or communication.

Any form of communication involves at least two parties: the sender and the receiver. Typically the business analyst will need to send their understanding of the problem to the developer (the receiver.) The important thing to realise is that the content of the message is decided by the receiver, it is they who interpret the communication and decide what it means. No matter how much effort the sender puts into their message they have no means of guaranteeing it is interpreted as they intend.

Now there are two opportunities for error here. We could assume that the receiver knows very little about the problem domain, to compensate we write a lengthy document that discusses all the details necessary. Unfortunately this approach risks overwhelming the receiver with details so they miss some of the important points.

Alternatively, we could assume that our developer knows quite a bit about the problem domain already and just communicate the bare essentials. The trouble now is that we are reliant on the knowledge the developer already holds, any omissions or errors in their knowledge will actually introduce changes which need correcting later on.

More subtly, the developer may have good knowledge about the problem domain with few omissions or errors but this may lead them to use assumptions and mental short-cuts which have worked well in the past but aren't appropriate in this case.

1 That other Holy Grail - "reusable software" - may also be finding a few less devotees, but that is another story.

Developers aren't the only ones who may hold hidden assumptions, the same may be true of the business analyst, or even the end-users and managers who are commissioning the system. Few businesses have a written operating procedure, often the arrival of business analysts will be the first time someone has ever tried to codify what these people are doing.

In any environment there is normally a lot of tacit knowledge which helps people go about their business. Not only is this information rarely codified but it can be difficult to recognise and extract, it is often embedded in the culture and "the way we do things here." As we delve into the process, either through writing a specification or developing code, we will uncover more and more of this knowledge and much of this will lead us to change our understanding of the process.

On occasions people may choose to withhold information which we need to develop software, but often we may fail to recognise that there is information present or that is relevant. Such information may be embedded in the working practices and culture of the people. For example, it may seem unimportant that every new recruit is told the story of how Old Joe managed to flood the basement one day, but in fact they are being warned about the basement and the water supply.

This stuff is notoriously difficult to capture and document. Anyone who has written a pattern will recognise the difficulty in capturing just what the pattern is about and how we use it, much of the detail exists as tacit knowledge inside our heads but putting it down in a form accessible to others can be incredibly difficult.

It is inevitable that we will fail to capture important tacit knowledge when we draw up our system requirements. Successive iterations may expose more and more but some of it will only emerge when we reach testing and system deployment.

The good news is that it is easier to change systems that are rooted in tacit knowledge than those based on explicit information and agreement. Think of the rule handed down through quietly observing ones fellow employees: "First one in boils the kettle". If we buy a timer for the kettle this is easy to change. However, imagine it is explicitly written into everyone's contract, agreed with unions, incorporated in the quality manual and approved by head office. Changing that is going to be a lot more difficult.

So, although it may be more difficult to develop a system when the requirements are tacit, it should be easy to deploy the system. Conversely, where requirements are explicit, in say written procedures, it may actually be more difficult to integrate a new system.

Temporal dimension

Requirements documents are at best a snapshot of the way things stand at the time they are written. However things change, if we start the project on 1 January, spend a month writing documents and head back to our office to develop and test the system for the rest of the year we can be sure things will have changed in the intervening time. Hence requirements documents need to be living documents, we may not want to accept every change that is asked for, however, setting them in stone will miss important changes.

There are few computer systems introduced today that merely automate existing practice. Instead, systems are implemented as part of an attempt to change practices. This means that to some degree the specifications are attempts to describe how things will be. Since none of us – not even management consultants – are blessed with perfect future vision it is inevitable that over time we will see changes that are needed in the proposed process and system specification.

While we have good knowledge about our internal environment and we can make plans for internal changes we have no such knowledge or control about the external environment. Things that happen outside our problem domain can have as much, or even more, influence as internal events on what is required of a new computer system.

It is a cliché to say the pace of change in business is faster than ever before but there is at least a grain of truth in the statement. Events in the market or action by rivals can radically change what we require from a new system. Imagine a book seller who commissioned a new stock control and retail system in the mid 1990s, they may have had the perfect specification for internal requirements but external events will have forced all sorts of changes from internet retailing to new models of revenue generation upon them.

There is a necessity for all requirements documents to be forward looking but this is also a hindrance. Again, making the document longer will make it less well understood, attempting to cover all the bases may result in a system with more bells-and-whistles than are necessary. System development cost and time may escalate and still events may over take the company.

An empirical study

A study by Edberg and Olfman (2001) looked at the motivations behind software change requests at a variety of organisations during the software maintenance phase. Corrective maintenance (i.e. bug fixing) accounted for only 10-15% of work while functional enhancements accounted for over 60% of changes. This 60% was broken down into four categories:

- External changes – changes required to meet some need from outside the organisation, say a changed legal requirement.
- Internal changes – changes required because of company changes such as new products or restructuring.
- Technical changes – required to meet new technical demands.
- Learning – changes resulting from learning by individuals or groups.

Edberg and Olfman suggest that 40% of these changes where primarily the result of learning. By changing software, organisations can pass on the benefits of one group's learning to the whole company – potentially saving money and/or time and improving efficiency.

Interestingly though, users who requested changes often didn't attribute their request to learning, they preferred to cite other internal or external factors as the motivation. It seemed that requesting a change that would save them time, and eventually make the whole company more competitive, wasn't seen as a good enough reason to ask for a change.

Does this mean the world full of self-effacing people? No, it would seem information systems (IS) people have made their dislike of changes very clear:

> *Almost uniformly among users in work groups, there was a strong belief that the IS organization did not want to enhance software and that changes had to be justified in some way other than it would help work activities. The interviewees in IS organizations agreed, frequently commenting that the enhancements required by users were "superfluous" and, in the opinion of IS, not necessary for users to do a good job. There was a consistent conflict between work groups and IS organizations at each case about what constituted a necessary enhancement to software. (Edberg, 2001)*

People learn more

While Edberg and Oldman suggest system changes are the result of learning other researchers (e.g. Ang, 1997) suggests that system development can act as a catalyst for people and organisations to learn about their activities. I'd like to suggest that a natural extension of this process is that the very act of analysing and specifying a computer system will change the problem. How often does someone sit down with a manager or other office worker and enquire into what they do? How often do we attempt to map the processes that occur in our work environment? And how often does someone write a document describing what goes on?

Actions such as these are perfectly normal activities for business analysts writing a specification. However, the very act of doing them will cause people to reflect on what they are doing, why they are doing it and whether things can be done better. True, some work environments may be so oppressive that people keep these insights and ideas to themselves but other companies' activities encourage people to improve their processes.

It's not only the end users who will learn and change as the system develops. The developers tasked with writing the new system will gain insights into the business and the application of technology, which cause them to change their interpretation of the specification.

In fact, in coding it may not be possible to implement all the fanciful promises made by a salesman, or the vague requirements in a specification document. The coding process forces us to face the reality of what is possible and what isn't. Clients may be oversold a solution by a salesman who promises everything (at a very reasonable price), the specification may be beautifully worded to describe how these things will be brought about, but, when it comes to executable code, issues can no longer be fudged.

At this point the reality of constructing a solution may force a change in the specification. These can be among the most difficult changes to bring about since such changes may not be what people want to hear about. However, this highlights the importance of keeping a feedback cycle from developers to customers and continuing a dialogue over the system requirements.

The other second system effect

Fred Brooks said:

> *The second is the most dangerous system a man ever designs. ... The general tendency is to over-design the second system, using all the ideas and frills that were cautiously side-tracked on the first one. (Brooks, 1975)*

Brooks was discussing the tendency of software developers when building systems. However, there is another *second system effect*, this time within the organisation that decides to replace an existing system, which can bring about the same effects.

On the face of it, if a corporation has a working system it is to be congratulated and the story finished. But we often find companies that want to replace their existing systems. Given the reputation of IT projects to over-run budgets and time one wonders why they would want to take this step, but they do.

At one level, writing a second system should be easy. Get a group of developers, give them the existing system and say "Copy it." But things don't work that way. The system is usually redeveloped because it fails to satisfy some need, so the instruction is more like "Copy it and ....".

It's the "and..." bit which is difficult. The first item on this list is the immediate reason for the new system, that which the original system doesn't do. Next on the list will be all the things the original system was supposed to do but never did.

While the existing system was in place things were frozen, no matter how much people wanted things to change it wasn't going to happen. But once development on a new system begins the position is unfrozen, all that pent up frustration with the existing system can be directed as additions to the new one. Then, as people see the new one take shape, the learning process is seeded and more changes will come along. However, once the new system is delivered and deployed things freeze again as the window of opportunity closes.

Resistance is...

Software engineering books are full of suggestions on how to manage changing requirements. Unfortunately many of them look at Barry Boehm's (1988) economic model of software development and note that the later changes occur in the process the more they cost, they therefore conclude that change is bad and needs to be resisted.

If we go down this route we face two serious problems. Firstly we are going to make ourselves unpopular, the software developers and managers will come to be seen as the people who always say "No." Who wants a bunch of uncooperative people around the office?

Secondly, this assumes that the changes that come along after the project reaches some arbitrary cut-off point are worth less than those that came along before the cut-off point. Changes need to be assessed both in terms of the complexity they add and the value they add. Changes that come along later are more disruptive but this doesn't imply they are valueless, only that they must be worth more if they are to be worthwhile implementing.

The argument that we should resist change is based on the naive assumption that we were able to capture all the valuable requirements up-front and therefore, none that come along later are worthwhile. However, as you can see from my arguments I don't believe this is the case.

In fact, I will go further. I think it is quite possible, indeed perhaps probable, that the most worthwhile requirements for the system will only come to light as the system develops. Only as people – both developers and clients – come to understand the new system and how they will use it will the most valuable requirements become apparent.

When we write the initial specification we document the *low hanging fruit*. The specification will include the most obvious requirements, those that were discussed before the project started, those which are already documented and those that people think of in the early stages. Yet as the project proceeds, everyone involved will get a more detailed understanding of what is happening both in the software and the company, potentially revealing even greater value in a system. Consequently, it is necessary to reprioritise our work as we go.

What can we do about this?

The software development community needs to rethink its approach to changing requirements. We need to stop seeing changing requirements as a problem and start to see them as an opportunity. If we can pin down requirements and stop them from changing then two things happen. First, our organisations cease to change – this isn't good in a dynamic business environment. Second, anyone can implement our requirements because they are fixed and known. That anyone could be a competitor company, or it could be an outsource organisation with low costs.

However, when we address the changing requirements the opposite is the case. Our organisations become more flexible and can out-compete the competition because we can adapt to our environment and market more quickly. Secondly, this ability to adapt and change becomes so fundamental to the organisation that it is unthinkable to outsource it and create space between software developers and their customers.

We actively want to reach a position where new system development is generating new ideas for the business, where the specification is no longer focused on the *low hanging fruit* requirements but is addressing the most valuable.

Software development books are full of techniques to make our software development more responsive: shorter development cycles, iterative development, rapid-application development, and so on. Underpinning all of these ideas is the concept of improving the feedback cycle by making it both faster and clearer.

So, my solution to changing requirements is to improve communication between people. That is, all the people involved, the programmers, testers, analysts and customers. And by communication I don't want to see more documents, or more e-mail, I want to see people talking to one another clearly and honestly. This means we have to value the individuals not the process or the technology.

Conclusion

Requirements change, that's a fact of life. Many IT people have adopted a mindset that change is to be resisted, indeed, many IT people have been so successful in training their customers to expect resistance to change that customers have given up. (Hardly surprising then that IT people get bad press.)

If we look beyond the change requests themselves we see that there are good, valid reasons people request change. Potentially, through IT systems, companies can get to know themselves better. Computer systems have a role to play in helping companies change.

In the current debate on agile software development we need to be considering the user perception of software change. What use is agile software development if users have been indoctrinated into rigidity? For agility in software development to mean anything it must be combined with an agile organisation, we cannot view software development as an isolated activity.

Allan Kelly
`allan@allankelly.net`

Bibliography

Ang, K., Thong, J.Y. L. and Yap, C., 1997, IT implementation through the lens of organizational learning: a case study of insuror, *International Conference on Information Systems*, `http://portal.acm.org/toc.cfm?id=353071&coll=portal&dl=ACM&type=proceeding`

Boehm, B., and Pappacio, P.N. (1988) Understanding and controlling software costs, *IEEE Transactions on software engineering*, 14, 1462-77.

Brooks, F. (1975) *The mythical man month: essays on software engineering*, Addison-Wesley.

Edberg, D., and Olfman, L., 2001, Organizational Learning Through the Process of Enhancing Information Systems, *34th Hawaii International Conference on System Sciences*, IEEE, `http://csdl.computer.org/comp/proceedings/hicss/2001/0981/04/09814025.pdf`

McConnell, S. (1993) *Code Complete*, Microsoft Press, Redmond, WA.

C++ as a Safer C

by Sven Rosvall

There are many features in C++ that can be used to enhance the quality of code written with classic C design even if no object oriented techniques are used. This article describes a technique to protect against value overflow and out-of-bounds access of arrays.

This article started with a discussion about how C projects could use features in C++ to improve the quality of the code without having to do any major redesign.

Bounded Integral Types

The built-in integral types in C and C++ are very crude. They map directly to what can be represented in hardware as bytes and words with or without signs. There is no way to say that a number can only have values in the range 1 to 100. The best you can do is to use an unsigned char which typically has a value range from 0 to 255, but this does not provide any checking for overflow.

It is easy to create an integral type that does the range checking as Pascal and Ada do. The implementation of `BoundedInt` in listing 1 shows how this can be done with C++ templates. It takes three parameters. The first two specify the inclusive range of allowed values. The third parameter specifies the underlying type to be used and uses a default type given by the `BoundedIntTraits` class.

The `BoundedIntTraits` class is used to find the smallest built-in type that can hold numbers of the specified range. It uses some meta-programming to figure out which type to use. The implementation of the `BoundedIntTraits` class is shown in listing 2.

The checking is performed here by using the `assert()` macro. Note that this checking only happens in debug builds and not in the release builds to reduce the overhead for this checking. Using inlining and the `assert()` macro removes any overhead in optimised release builds. With a good optimiser the resulting code will be identical to when built-in types are used. Alternatives to `assert()` can of course be used such as throwing an exception or logging a message to a file.

The `BoundedInt` class is only designed to work with value ranges that fit in an `int`. To support wider ranges all methods that take an `int` as a parameter must have overloaded siblings that take a `long`, or even `long long` where supported.

The `operator+=()` member must check that the new value is within the valid range. It also has to check that there is no overflow during addition. The method of detecting overflow is

```
#include <cassert>

template <int Lower,
          int Upper,
          typename INT=typename
          BoundedIntTraits<Lower,
                           Upper>::Type>
class BoundedInt {
public:
  // Default constructor
  BoundedInt()
#ifndef NDEBUG
    : m_initialised(false)
#endif
  {}

  // Conversion constructor
  BoundedInt(int i)
    : m_i(static_cast<INT>(i))
#ifndef NDEBUG
    , m_initialised(true)
#endif
  {
    // Check input value
    assert((Lower<=i) && (i<=Upper));
  }

  // Conversion back to a builtin type
  operator INT() {
    assert(m_initialised);
    return m_i;
  }

  // Assignment operators
  BoundedInt & operator+=(int rhs) {
    assert(m_initialised);
    // Check for overflow
    assert(m_i/2 + rhs/2 + (m_i&rhs&1)
           <= Upper/2);
    assert(Lower/2
           <= m_i/2 + rhs/2 - ((m_i^rhs)&1));
    // Check result value
    assert((Lower<=m_i+rhs) && (m_i+rhs<=Upper));
    // Perform operation
    m_i += rhs;
    return *this;
  }

  // Increment and decrement operators.
  BoundedInt & operator++() {
      assert(m_initialised);
    // Check for overflow
    assert(m_i < Upper);
    // Perform operation
    ++m_i;
    return *this;
  }
  // Other operators ...

private:
  INT m_i;
#ifndef NDEBUG
  bool m_initialised;
#endif
};
```

Listing 1: Definition of `BoundedInt`. Only the plus operator is shown here. The other arithmetic operators follow the same design.

```
#include <climits>

// Compile time assertion:
template <bool condition>
struct StaticAssert;
template <>
struct StaticAssert<true> {};

// Template for finding the smallest
// built-in type that can hold a given
// value range, based on a set of
// conditions.
template< bool sign, bool negbyte,
          bool negshort, bool negint,
          bool sbyte, bool ubyte,
          bool sshort, bool ushort,
          bool sint>
  struct BoundedIntType;

template<>
struct BoundedIntType< true, true, true,
                       true, true, true,
                       true, true, true> {
  typedef signed char Type;
};

template< bool negbyte, bool sbyte,
          bool ubyte>
struct BoundedIntType< true, negbyte,
                       true, true,
                       sbyte, ubyte,
                       true, true,
                       true> {
  typedef signed short Type;
};

template<bool negbyte, bool negshort,
         bool sbyte, bool ubyte,
         bool sshort, bool ushort>
struct BoundedIntType< true, negbyte,
                       negshort, true,
                       sbyte, ubyte, sshort,
                       ushort, true> {
  typedef signed int Type;
};

template <bool sbyte>
struct BoundedIntType< false, true, true,
                       true, sbyte, true,
                       true, true,
                       true> {
  typedef unsigned char Type;
};

template< bool sbyte, bool ubyte,
          bool sshort>
struct BoundedIntType< false, true, true,
                       true, sbyte, ubyte,
                       sshort, true,
                       true> {
  typedef unsigned short Type;
};

template< bool sbyte, bool ubyte,
          bool sshort, bool ushort,
          bool sint>
struct BoundedIntType< false, true, true,
                       true, sbyte, ubyte,
                       sshort, ushort,
                       sint> {
  typedef unsigned int Type;
};

// The traits template provides value
// range information to the
// BoundedIntType to get the smallest
// possible type.
template <int Lower, int Upper>
struct BoundedIntTraits {
  StaticAssert<(Lower <= Upper)> check;
  typedef typename
    BoundedIntType<Lower < 0,
                   Lower >= CHAR_MIN,
                   Lower >= SHRT_MIN,
                   Lower >= INT_MIN,
                   Upper <= CHAR_MAX,
                   Upper <= UCHAR_MAX,
                   Upper <= SHRT_MAX,
                   Upper <= USHRT_MAX,
                   Upper <= INT_MAX>::Type Type;
};
```

Listing 2: Definition of `BoundedIntTraits`. The types `long` and `unsigned long` are not included to keep the listing shorter.

complicated as there is no support for detecting overflow for built-in types in C and C++. The method here scales down all values to manageable sizes in order to do an overflow check. Because of the scaling down, it has to keep track of carry over data from the least significant bits to work properly in edge cases where the value range is close to the value range of the underlying type.

Other arithmetic assignment operators that `BoundedInt` should support are not shown here as they would take too much space. The design of these operators follows the design for the plus operator.

There are no binary arithmetic operators defined. When a `BoundedInt` object is used in a binary arithmetic operation, it will be converted to a built-in integral type before the operation. This means that there is no checking of the results of these operations, unless the result is assigned to a `BoundedInt` object. There is a pitfall here in that overflow cannot be checked for.

```
BoundedInt<-10, INT_MAX> a = 10;
a += INT_MAX;    // Overflow checked
a = a + INT_MAX; // Overflow not checked
```

A default constructor is available in order to mimic the behaviour of built-in types. It does not initialise the value but maintains a flag to indicate that this object does not have a defined value. This flag is checked by member functions that access or modify the value. The `m_initialised` member flag is surrounded by conditional pre-processing directives to avoid overhead in release builds.

The copy constructor and copy assignment operators are not defined as the compiler generated versions are appropriate.

Below are some examples from an imaginary C project implementing a lift control with a single change to use `BoundedInt`:

```
typedef BoundedInt<-4, 17> FloorNumber;
FloorNumber liftPosition = 0;
const FloorNumber myOfficeFloor = 10;

/* go up */
++liftPosition;

/* go up fast */
liftPosition += 4;
printf("The lift is %d floors away.\n",
       abs(liftPosition-myOfficeFloor));
```

`BoundedInt` objects can appear in any arbitrarily complex expression thanks to the conversion operator. Because the conversion operator is inlined the `BoundedInt` object will generate exactly the same code as when using a built-in type.

Bounded Arrays

A `BoundedInt` object can be used as a bounds checked index into arrays. Example:

```
const int SixPackSize = 6;
Bottle myBeers[SixPackSize];
BoundedInt<0, SixPackSize-1> ix;
for( ix = 0 ; ix < SixPackSize ; ++ix ) {
  drink(myBeers[ix]);
}
```

If `ix` for some reason is changed to an invalid value, the `BoundedInt` class will warn about this.

We can take this one step further by creating a class that only allows element access using numbers within the allowed range.

```
template <typename T, size_t Size>
class BoundedArray {
public:
  T& operator[](BoundedInt<0,
                Size-1> ix) {
    return m_data[ix];
  }
public:
  T m_data[Size];
};
```

Note that the member data is public to allow aggregate initialisation. See how this is used below. The member data can be made public without risk for misuse as the data is equally accessible through the index operator as with direct access.

Whenever an element is requested using an index of any built-in integral type, that index is converted to a `BoundedInt` which checks that its value is within the acceptable range.

This template takes two parameters, the type of the elements in the array and a non-type template parameter to indicate the size of the array. The simple example above will work as before with only a small change to the definition of `myBeers`.

```
BoundedArray<Bottle, SixPackSize>
  myBeers;
```

This array can be initialised in the same way as a built-in array:

```
BoundedArray<Bottle, SixPackSize>
  myBeers = { ... };
```

There is no overhead in release builds for this array class. The index operator is inlined and there is no indirect pointer access to the underlying array. Having the size as a template parameter may look like we are causing code bloat if several arrays of different sizes are used. Yes, there will be several instantiations but because all functions are inlined and optimised away there is no extra code that can multiply.

Bounded Pointers

In the same way as for using checked array indices we can create a smart pointer class that makes sure that it points to an element inside the array. It will have to know the base address of the array and the size to do the checking. This information is retrieved from the array class when a pointer is created.

The starting point is an example with built-in pointers:

```
Bottle* p = myBeers;
for( ; p->size != 0 ; ++p ) {
  drink(*p);
}
```

`myBeers` is an array where the last elements members are cleared as a termination condition. We replace the built-in pointer `p` with a smart pointer:

```
BoundedPointer<Bottle>
  p = myBeers;
```

The loop in the example above remains unchanged.

The definition of `BoundedPointer` is shown in listing 3. The array base address, array size and the initialised flag are kept as members only for debug builds to perform the runtime checks. To avoid this overhead in release builds the `m_base`, `m_size` and `m_initialised` members are surrounded with conditional pre-processing directives.

A `BoundedPointer` object can be constructed from built-in arrays and from user defined array types. The constructor for user defined array types takes two parameters (base address and size) and is intended to be called from conversion operators of those array classes. This conversion operator for `BoundedArray` looks like this:

```
template <typename T, size_t Size>
class BoundedArray {
public:
  ...
  operator BoundedPointer<T>() {
    return BoundedPointer<T>(m_data,
                             Size);
  }
};
```

There is also a constructor that takes a `void*` parameter to support assignment from `NULL`. A `T*` parameter cannot be used as it would conflict with the constructor for built-in arrays.

The `BoundedPointer` class supports all the operations that can be used with built-in pointers. There are checks for incrementing and decrementing the pointer to make sure that it does not point outside its array. As with `BoundedInt` there are checks to see that the pointer is initialised when it is used.

All methods are inlined to avoid any overhead in release builds.

Usage

The classes described here are designed to do the bounds checking during unit and system testing when compiled in debug mode. It is important to run as many test cases as possible that exercise all boundary conditions.

In release builds, all you have to do is make sure that the `NDEBUG` macro is defined, inlining is enabled and the optimise level is as high as possible. Then your code will be as efficient as if built-in types were used.

The `BoundedIntTraits` in listing 2 hides the chosen underlying integral type. If the ranges change in the future, there is no need to manually change the underlying type required for the wider range.

Extensions

This article describes the design of a class that wraps an array and adds bounds checking functionality. There are many more possible classes that can be used in this framework for different purposes. Examples include a class that manages dynamically allocated arrays.

A possible extension to the checked pointer is to keep track of whether the array still exists. If the array goes out of scope or is de-allocated the pointer shall be set to an invalid state. This is straight-forward to implement but is outside the scope of this article.

This article does not discuss checked iterators for STL containers as the article was originally intended to motivate C users to adopt C++ to improve their lives. For STL there are already implementations that check validity of the iterators.

Portability

Although the code in this article has been tested with several C++ compilers there are some difficulties using some existing compilers.

If your compiler does not support partial template specialisations you cannot use the traits class `BoundedIntTraits`. You can avoid the `BoundedIntTraits` class by removing it from the template parameter list of `BoundedInt` and replace it with `int`. You will miss the feature where the underlying type of `BoundedInt` is automatically chosen from the specified range and it will be `int` if a type is not specified.

Conclusion

With the strategies shown in this article it is possible to catch various out of bounds conditions during the testing phase at no cost to the released code.

An additional benefit is that the bounds given to `BoundedInt` and the array types document their valid ranges well.

Sven Rosvall
sven-e@lysator.liu.se

Related Reading

Safe and efficient data types in C++ by Nicolas Burrus
`http://www.lrde.epita.fr/dload/20020925-Seminar/burrus0902_datatypes_report.pdf`

Describes classes for compile time type safety when using different integral types. It defines safe operations for a set of integral types. The integral types used here are only bounded by the number of bits used in the internal representation. The description of operations and integral promotion is interesting and can be applied to the classes in this article.

Boost Integer Library
`http://boost.org/libs/integer/index.htm`

Contains some helpful classes for determining types of integers given required number of bits. Also contains other helpful classes that can be useful in implementing a portable bounded integer and pointer library.

Boost array class in the container library
`http://www.boost.org/libs/array/array.html`

A constant size array class. The design goal for this class is to follow the STL principles.

Bounds checking pointers for GCC.
`http://gcc.gnu.org/projects/bp/main.html`

Additions to GCC to add bounds checking to the generated code.

Safe STL
`http://www.horstmann.com/safestl.html`

An implementation of STL that performs various run-time checks on iterators.

CheckedInt: A Policy-Based Range-Checked Integer by Hubert Matthews

Overload issue 58, December 2003

Describes how policy classes can be used to select behaviour when a given range is exceeded.

```
#include <cstddef>
#include <cassert>

template <typename T>
class BoundedPointer {
public:
  // Default constructor
  BoundedPointer()
#ifndef NDEBUG
    : m_initialised(false)
#endif
  {}
// Constructor from a built-in array
  template <size_t Size>
  BoundedPointer(T (&arr)[Size])
    : m_p(arr)
#ifndef NDEBUG
    , m_base(arr), m_size(Size)
    , m_initialised(true)
#endif
  {}
  // Constructor from a user defined array
  BoundedPointer(const T* base, size_t size)
    : m_p(const_cast<T*>(base))
#ifndef NDEBUG
    , m_base(m_p)
    , m_size(size)
    , m_initialised(true)
#endif
  {}
  // Constructor from null
  BoundedPointer(void * value)
    : m_p(static_cast<T *>(value))
#ifndef NDEBUG
    , m_base(m_p), m_size(1)
    , m_initialised(true)
#endif
  {}
  // Dereference operators
  T & operator*() {
    assert(m_initialised);
    assert(m_p != 0);
    return *m_p;
  }
  T * operator->() {
    assert(m_initialised);
    assert(m_p != 0);
    return m_p;
  }
  T & operator[](size_t ix) {
    assert(m_initialised);
    assert(m_p != 0);
    assert(m_p + ix < m_base + m_size);
    return m_p[ix];
  }
  // Pointer arithmetic operations
  ptrdiff_t operator-(BoundedPointer
                         const & rhs) {
    // Check validity of the pointers
    assert(m_initialised);
    assert(rhs.m_initialised);
    assert(m_p != 0);
    assert(rhs.m_p != 0);
    // Ensure both pointers point to same array
    assert(m_base == rhs.m_base);
    return m_p - rhs.m_p;
  }
  BoundedPointer & operator+=(ptrdiff_t rhs) {
    // Check validity of the pointer
    assert(m_initialised);
    assert(m_p != 0);
    m_p += rhs;
    assert(m_base <= m_p && m_p < m_base + m_size);
    return *this;
  }
  BoundedPointer & operator++() {
    // Check validity of the pointer
    assert(m_initialised);
    assert(m_p != 0);
    ++m_p;
    assert(m_p < m_base + m_size);
    return *this;
  }
  // Other arithmetic operators ...
  // Comparison operators
  bool operator==(BoundedPointer const & rhs) {
    // Check validity of the pointers
    assert(m_initialised);
    assert(rhs.m_initialised);
    assert(m_p != 0);
    assert(rhs.m_p != 0);
    // Make sure that both pointers point
    // to the same array
    assert(m_base == rhs.m_base);
    return m_p == rhs.m_p;
  }
  // Other comparison operators ...
private:
  T * m_p;
#ifndef NDEBUG
  T * m_base;
  size_t m_size;
  bool m_initialised;
#endif
};
// Binary arithmetic operators
template <typename T>
inline BoundedPointer<T>
operator+(BoundedPointer<T> lhs, int rhs) {
  return lhs.operator+=(rhs);
}
template <typename T>
inline BoundedPointer<T> operator+(int lhs,
                      BoundedPointer<T> rhs) {
  return rhs.operator+=(lhs);
}
```

Listing 3: Definition of `BoundedPointer`.

Heretical Java #1: Immortality – at a price

by Alan Griffiths

There is a widespread belief that because Java provides "garbage collection" the programmer automatically avoids the memory management problems that plague the users of other languages. This opinion continues to exist despite there being plenty of material that attempts to correct this impression – and the existence of tools to address memory management problems.

The typical Java developer of my experience either doesn't know there is a problem or, more rarely, knows there is a problem but doesn't know how to address it. While it may be that I'm stretching a point to class this as the "orthodox" view I still feel justified in addressing the topic because it is far better not to create a mess than to have to sort one out.

Part of the problem stems from the treatment of the topic in numerous introductory texts. These tend to present this uncritical viewpoint of "garbage collection" – and early beliefs are always the hardest to challenge. I can remember realising that there was something wrong with the way I was taught about the lifecycle of Java objects. The books I had read about Java told me that their life ended when they were destroyed by the garbage collector. For example in *Java in a Nutshell* we have:

> *The technique Java uses to get rid of objects once they are no longer needed is called garbage collection. It is a technique that has been around for years in languages such as Lisp. The Java interpreter knows what objects it has allocated. It can also figure out which variables refer to which objects, and which objects refer to which other objects. Thus, it can figure out when an allocated object is no longer referred to by any other object or variable. When it finds such an object, it knows that it can destroy it safely, and does so. The garbage collector can also detect and destroy "cycles" of objects that refer to each other, but are not referred to by any other objects. [Flanagan]*

Or alternatively in *Exploring Java* we have:

> *Now that we've seen how to create objects, it's time to talk about their destruction. If you're accustomed to programming in C or C++, you've probably spent time hunting down memory leaks in your code. Java takes care of object destruction for you; you don't have to worry about memory leaks, and you can concentrate on more important programming tasks. [Niemeyer/Peck97]*

Object Lifecycles

In all object-oriented systems the design of object lifecycles is important because objects have responsibilities to meet at significant points in their lives. If an object is being destroyed then it must ensure that any resources that it owns are either released or passed on to a new owner. And sure enough, *Exploring Java* continues with:

> *Before a method is removed by garbage collection, its* `finalize()` *method is invoked to give it a last opportunity to clean up its act and free other kinds of resources it may be holding. While the garbage collector can reclaim memory resources, it may not take care of things like closing files and terminating network connections very gracefully or efficiently. That's what the* `finalize()` *method is for. [Niemeyer/Peck97]*

When I first read this it all seemed to make sense, but while working with Java, I became increasingly conscious that reality didn't accord with this view. For example, when working with instances of the AWT Graphics class I learnt very quickly that I needed to call `dispose` – and not to rely on the object to do so when it was destroyed.

I'm clearly not the only one to see that there is a problem. It is common to see advice like "only put debug code in the `finalize` method" (which directly contradicts the last quote). One of the books that tries to address the problem is *Thinking in Java* which says:

> *This is a potential programming pitfall because some programmers, especially C++ programmers, might initially mistake* `finalize()` *for the* destructor *in C++, which is a function that is always called when an object is destroyed. But it is important to distinguish between C++ and Java here, because in C++* objects always get destroyed *(in a bug-free program), whereas in Java objects do not always get garbage-collected. Or, put another way:*
>
> Garbage collection is not destruction. *[Eckel98]*

Over time I accumulated an assorted collection of rules of thumb that dealt with most circumstances. But they lacked conceptual elegance and when new circumstances occurred they required new rules to be worked out carefully.

Then one day I was reading something by Bjarne Stroustrup about the use of garbage collection. Stroustrup wasn't writing about Java (he's the creator of C++) but, despite what you may have heard in some of the Java texts, garbage collection is available to C++ programmers. What Stroustrup said made sense of these rules:

> *Garbage collection can be seen as a way of simulating an infinite memory in a limited memory. With this in mind we can answer a common question: Should a garbage collector call the destructor for an object it recycles? The answer is no, because an object placed on the free store and never deleted is never destroyed. [Stroustrup91]*

This realisation bound all my rules of thumb together as a single idea: in Java objects are immortal – they are never destroyed. All that should happen in garbage collection is that the memory "owned" by the object is recycled. While important for the application as a whole this isn't a significant event in the object's lifecycle – and we shouldn't expect the object to respond in any significant way. (Which is consistent with the difficulty of writing effective `finalize` methods – which Stroustrup also alludes to later in the same passage.)

The idea of an object continuing to exist without its memory may sound a little strange – but objects exist without other resources that make them useful (a Graphics object still exists after its native peer has been released by calling `dispose`). In any case, the rules of garbage collection ensure that a program cannot tell if the object is there or not. While from the point of view of designing my programs I find the idea that the object is going to sit there forever holding onto any resources I haven't told it to release compelling.

The Design of Java's Garbage Collection

One of the important points taken up by the "Patterns" movement of software design is that a "solution" has consequences. With any design decision there are tradeoffs: resolving one problem may make others worse or even introduce new ones. When the designers of Java adopted garbage collection to manage memory they didn't provide a solution to all the resource management problems a developer will ever encounter. Nor did they believe that they had:

> *Garbage collection (GC) is probably the most widely misunderstood feature of the Java platform. GC is typically advertised as removing all memory management responsibility from the application developer. This just isn't the case. On the other hand, some developers bend over backwards trying to please the collector,*

and often wind up doing much more work than is required. A solid understanding of the garbage collection model is essential to writing robust, high-performance software for the Java platform. [JPAppGC]

There are memory management problems that Java's garbage collection based model for object lifetimes doesn't solve, but nothing is harder to fix than a problem people won't believe in. And the orthodox belief that "you don't need to worry about memory leaks" denies the obvious – memory is a finite resource that needs to be managed.

The following will explore the "infinite memory/immortal objects" design model of the object lifecycle and the way in which it helps to deal with the management of resources in a Java program.

The next section "Garbage Collection and the Object Lifecycle" provides necessary detail of how garbage collection operates for discussing the problems that it does solve and those that are left for the programmer to address.

In "Managing Memory" we will be looking at the memory management problems that programmers can still encounter and the solutions to them. (The solutions are easy once you realise that the problems and solutions exist.)

In "Managing Other Resources" we will examine the management of other resources. In many programming languages (and C++ is often cited in the literature) these can be dealt with by the same mechanisms as managing memory. However, in Java, the commonality in the way these problems are addressed is less obvious – garbage collection addresses a lot of memory management issues but leaves other resource management issues to the developer.

Garbage Collection and the Object Lifecycle

In this section I explain what "garbage collection" does for the developer and why it fails to be the "silver bullet" that many think it is. With this knowledge we will be then equipped to tackle the problems of managing both memory and other resources that may be associated with an object.

In abstract terms "garbage collection" is a service provided by the Java runtime environment to reclaim memory from objects that the program is not going to use again. How does it know which objects are not going to be used again? It doesn't – if it knew instantly, and with complete accuracy, which objects are not going to be used then the orthodox view would be valid. But, even in theory, it is not possible to achieve complete accuracy in a useful timeframe. Instead garbage collection follows rules that can quickly identify objects that definitely won't be used again.

Although the details of how the runtime environment works out which objects the program can or cannot access depend upon the implementation of Java, there are certain rules that it must follow. (I cannot give a single reference for these rules as they are spread around the Java Language Specification and the JVM specification and are not always stated explicitly.) These rules tell the runtime environment which objects the program cannot use – the trouble is that they can sometimes indicate that an object is "in use" when the programmer has forgotten all about it and will, in practice, never use it again. The object is dead but still consuming resources – such "zombie objects" can lead to a program consuming more and more memory until it fails (by slowing to a crawl or by crashing).

The rules the collector has to follow are as these:

1. There are some references referred to as the "root set" (I'll get back to this) – objects referenced by these are in use.
2. Any object referenced by a reference in an object that is in use is also in use. (This is obviously recursive.)
3. Don't collect memory from any object that is in use. *There is no requirement to collect memory from objects that are not in use.*
4. Before collecting memory from an object call its `finalize` method – remembering that `finalize` might set a reference somewhere to the object that changes it to be in use.
5. Don't call `finalize` on the same object twice.

The definition of the "root set" is key to the working of the collector and has changed subtly since the early days of Java (as users of the SINGLETON anti-pattern may have discovered). But for the programmer it suffices to assume that it includes any objects whose methods are active on a call stack, those referenced by local variables on a call stack, and by static member variables. (Actually the latter are not really in the root set – they are "in use" indirectly, by way of the corresponding class and classloader objects – they could become unused if the class was not loaded by the default classloader.)

It is significant that there is no requirement to collect memory from objects that are not in use. There is no guarantee that an object's memory will be collected – or that `finalize` will ever be called. There is a deprecated API – `System.runFinalizersOnExit` – that purports to ensure that all `finalizer`s will be called, but this has proved problematic:

This method is inherently unsafe. It may result in `finalizers` being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock. [JDK1.4.1]

Because one can never be sure that a Java object will be finalised or have its memory collected it is never a good idea (as already noted) to put any functional code in a `finalizer`. And, unless its `finalizer` contains functional code, once an object becomes eligible for collection, it will have no further effect on the state of the program – it can be forgotten.

Because comparisons are often drawn with C++ it is worth expanding on this point of difference between the lifecycle of a C++ object and that of a Java object. A typical C++ object has a lifecycle like that of a Java variable of primitive type (like an `int`): it is created where it is declared and no longer exists after the program leaves the scope in which it is declared. In C++ objects are notified when their life comes to an end (a special destructor "method" is called). In C++ programmers use this to free resources when an object goes out of scope[1]. The result of this is that the lifecycle of a C++ object has a clear and predictable beginning (it is created) and a clear and predictable end (it is destroyed). In Java objects also have a clear and predictable beginning but they don't have a clear end: instead the program just stops using them – after which they *may* be finalised, after which their memory *may* be collected.

Instead of dying (like a C++ object) it has attained a form of immortality – but this is at the cost of the programmer needing to ensure that it frees up any resources it might be holding when she finishes with it.

Managing memory

When a programmer needs an object she gets it from somewhere (usually by creating it using `new`), probably stores a reference to

1 Although C++ objects may also be created dynamically like Java objects (that is, by using `new`) these are idiomatically managed by special "smart pointer" classes that ensure that the memory doesn't leak. But, just as there are many Java books that fail to teach idiomatic memory management, there are many C++ books that fail to teach this.

it in a local variable, uses it for something (typically by calling some methods on it), and then she forgets about it. This is a very natural way to behave – we all do it unless we have had a reason to learn to do otherwise. Parents will recognise this as the way young children treat objects in the real world: they pick up a toy (or a piece of cutlery), make use of it for a while and then forget about it. Once a child has forgotten a toy it won't be long before they want to use their hands for something else and the toy will be left unattended. Eventually, a parent acts and, having decided that the object isn't being used will either send a message to the child ("pick that up and put it away!") or collect and deal with it.

Programmers (and children) get an important benefit from having things cleared up for them: it makes their life a lot simpler. Most of the time this is good – it allows them time to concentrate on matters that are important (most Java programmers are trying to solve problems, not to manage memory). The Java programmers I know were obviously children once and learnt some of the same strategies. But there are differences: instead of holding objects in their hands they now have reference variables. Children run out of hands, while programmers can endlessly create more variables, which means that there is no practical limit to the number of forgotten objects that a programmer might be "carrying around".

Java garbage collection is also different to most parents: it will never ask the programmer to "put that away" it simply deals with those objects that have been left lying around. Children don't like being nagged and neither do programmers, so this might seem great – except that putting things away is sometimes necessary. My youngest son recently negotiated a "no nagging" deal for his bedroom and has demonstrated that it becomes unusable in about a week – and he has no idea how to resolve this problem. He tried putting a couple of things away, but that didn't look any better – so he gave up.

Programs can get into the same state as that bedroom if the programmer relies on the garbage collector to magically take care of everything. Java developers who believe that you don't have to worry about memory management cause more problems than does the need to do it "by hand" in C or C++. These programs might function correctly for a while, but they use increasing amounts of memory until they collapse. Once this happens I've seen developers make a token effort at addressing the problem and then give up in the hope that it won't matter.

This doesn't need to be the case: there are idiomatic ways to manage memory in Java – it is ignorance of these idioms that causes problems, not the language. Java's garbage collection is a tool for managing memory – not a substitute. Garbage collection isn't unique to Java and, despite it frequently being cited as an advantage over C++ one may choose to use "garbage collection" in C++ but, by deliberate intent or accident, most developments in these languages don't use it. Each language provides a context and the developer must learn the idioms that work in that context.

In Java the programmer needs to ensure that there are no "live" references to objects that are no longer in use. This doesn't come naturally – the lessons learned in childhood are not automatically transferred into the made-up world of the JVM. There are parallels: in the real world children have to learn to behave responsibly with the objects they come in contact with, in the JVM world programmers have to learn to behave responsibly with the objects they come in contact with. The difference is that in the real world children learn from adults whereas in JVM world programmers usually learn from other programmers. (Science fiction writers such as William Golding [Golding] have speculated on the effect that a lack of adult input might have on children.) Why do these differences arise? In part it is because in the made-up world of the JVM we are freed from the constraints that arise in the real world. In part it is because the rules of the JVM world have been devised for the convenience of the implementers of that world.

Programmers are not stupid (and neither are children) but they are in the business of producing simplified descriptions of things (usually represented in code). Sometimes, however, they come up with descriptions that are too simple to work. One such description is a lifecycle of a Java object that goes: "creation (using `new` or `createInstance`), use (by accessing its methods or member variables) and forget about it (garbage collection will sort it out)". This is similar to a child in the real world: "create a game (by finding some toys), use (play with them) and forget about it (parents will sort them out)". In the real world parents will soon indicate that there is some learning to do, in the JVM world programmers need to discover this for themselves.

How does all this affect us as programmers? Well, the first thing that is clear is that if we have any references to unwanted objects then those objects will lurk around in limbo. This is quite easy to do unintentionally: all we have to do put the object into a collection and forget to remove it, or into a long lived variable and forget to reset it, or...

The answer is simple: ensure that any long-lived references are set to `null` when the object they reference is no longer in use. The problem the programmer has to address is deciding when use of an object is complete.

References that exist in function scope are rarely a problem: unless the code is written in a perverse style (e.g. excessively long methods) then the scope of the reference will be approximately the scope of use for the corresponding object. When this happens the reference will go out of scope in a timely manner and the object will become eligible for collection.

Instance reference members live as long as the instance that contains them – which can be a very long time (e.g. a SINGLETON lives "forever"). The problem for the developer implementing the class is that it is the user of the class that controls its lifetime – and must be relied upon to initiate any action that resets the reference. In many cases this is not worth the effort of solving (an obsolete reference will only hold memory for a single forgotten object) unless the referenced object also holds non-memory resources that must be released in a timely manner. (A subject we'll revisit later.)

It becomes more important to deal with problems when there is a possibility of holding references to multiple objects. For example, it is possible to implement a stack using an array and a "top" index. If references to multiple objects are pushed onto such a stack the objects will be held in memory until the entries in the array are cleared (or the array itself becomes eligible for garbage collection). This implies that the "pop" operation should reset the reference to the old "top" element.

Collections are where the problem becomes severe: for example, there is a long-lived collection buried in the depths of the Swing library that maps events to listeners. User code that adds listeners to Swing objects causes these objects to be added to the collection. In practice user code to remove these objects is rather rare and, in early versions of Swing, this used to lead to the collection becoming progressively larger each time a dialog (for instance) was displayed. More recent versions of Swing addressed this problem by using a special type of container: one that holds *weak references*.

From Mechanism to Method: Further Qualifications

by Kevlin Henney

Qualification is often used as a simple constraint on behavior. For instance, a non-`const` member function cannot be executed on a `const`-qualified object. In the general case, a `const`-qualified member function can be used with both `const`- and non-`const`-qualified objects; the exception is when the function is also overloaded privately as non-`const`. This exception highlights the other common use of qualification to distinguish between two functions and two different outcomes, although normally in a more substitutable fashion (i.e., so that the non-`const` version is effectively a subtype specialization of the `const` version [1]).

Qualification, Separation, and Unification

You can see this kind of substitutability in action with iterator access in the C++ Standard's container requirements: non-`const` container access yields `iterator`s whereas `const` container access yields `const_iterator`s, and, furthermore, an `iterator` may be used where a `const_iterator` is expected. In this case, qualification-based separation leads to `iterator` and `const_iterator` types.

Qualification can also be used as a means for unification. Consider an object that supports both reader locks and writer locks for safe multithreaded access: so long as there are no writers, multiple reader locks can be acquired and the owners can look at the object without changing it (i.e., only const operations); the owner of a writer lock has exclusive read-write access to an object (i.e., both `const` and non-`const` operations). Translating these concepts directly into code suggests the following:

```
class table {
public:
  void lock() const; // acquire lock for reading
  void lock();       // acquire lock for
                     //      reading and writing
  void unlock() const; // release reader lock
  void unlock();       // release writer lock
  ...
};
```

The acquisition and release of the lock now reflects the permissions of the calling context:

[continued on next page]

Weak References

Weak references are a feature introduced with JDK 1.2 with `WeakHashMap` and `WeakReference`. These allow some additional flexibility in garbage collection. A weak reference is recognised by the garbage collector as one that does not prevent an object being collected (but that must be reset should the object be collected). They allow the developer to keep track of an object for as long as there is a use of it somewhere else in the program, but to release it once that use is complete.

The use of `WeakHashMap` in the Swing library is typical of the scenarios where weak references are useful: in the OBSERVER pattern the subject should rarely affect the lifetime of the observers.

Managing other resources

In addition to the need to put objects away, there are also objects that need to be switched off. It took a lot of batteries going flat before my children learnt to switch off torches, walkie-talkies, Gameboys and other toys that contain batteries. Java objects can also contain resources that need to be "switched off" – graphic contexts, file handles, etc. These are objects that need to know when the programmer has done with them. For these objects the programmer needs to develop the discipline needed to supply the required notification.

In the vast majority of cases this is simply a matter of putting a call to a release method where it will be executed on exit from a block of code. There is even a convenient language construct for doing this: the `finally` block. It looks like this:

```
public void repaint() {
  Graphics g = getGraphics(); // Allocate
  if (null != g) {
    try { paint(g); }
    finally { g.dispose(); }  // Release
  }
}
```

In this code the scope rules are used to ensure that the paired operations of allocation and deletion always occur as a pair (and in sequence). This isn't hard – although correct examples are rare in the literature.

As with managing memory the issue becomes problematic when references to the owner of the resource are long lived (i.e. have instance or class scope). If the graphics object in the above example were referenced by a class member and accessed by a number of methods then it could be difficult to determine when use was completed. It may become necessary for the owning object to provide its own equivalent of the `dispose` method and to rely on its user to call it.

The management of resources is at its worst when there are long-lived references to the same resource-owning object in independent parts of the system. If this happens it can be very difficult to release the resource at the right time – without either implementing some convention for communicating between them or electing one to be "the boss" none of these can confidently release the resource. Fortunately, in real code this is rare. (There are options: weak references, proxy objects that hold a use count, etc.)

Conclusion

I hope this has shown that there are memory management issues to be addressed in Java and that these issues can be addressed. As so often in our profession it is the acknowledgement that there is a problem that is the key step to finding a solution.

Alan Griffiths
alan@octopull.demon.co.uk

References

[Stroustrup91] Bjarne Stroustrup, 1991, *The C++ Programming Language (2nd edition)*, Addison Wesley
[Golding] William Golding, *Lord of The Flies*, Faber and Faber
[Flanagan] David Flanagan, 1997, *Java in a Nutshell*
[Niemeyer/Peck97] Patrick Niemeyer & Joshua Peck, 1997, *Exploring Java (2nd edition)*, O'Reilly
[Eckel98] Bruce Eckel, 1998, *Thinking in Java*, Prentice Hall
[JPAppGC] `http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html`

[continued from previous page]

```
void reader_example(const table *target) {
  target->lock(); // acquires lock for reading
  ...
  target->unlock(); // releases reader lock
}
void writer_example(table *target) {
  target->lock();   // acquires lock for
                    //    reading and writing
  ...
  target->unlock(); // releases writer lock
}
```

Such access is made exception safe by introducing an acquisition-release object [2, 3]:

```
template<typename lockee_type>
class locker {
public:
  locker(lockee_type &target) : target(target) {
    target.lock();
  }
  ~locker() {
    target.unlock();
  }
private:
  locker(const locker &);
  locker &operator=(const locker &);
  lockee_type &target;
};
```

What's neat about this design is that only a single `locker` template is needed to cater for both `const` and non-`const` variants:

```
void reader_example(const table *target) {
  locker<const table> guard(*target);
                    // acquires reader lock
  ...
}
void writer_example(table *target) {
  locker<table> guard(*target);
                    // acquires writer lock
  ...
}
```

Code that is written to be both scope constrained and `const` correct will avoid pessimistic locking scenarios, where a lock is acquired for longer than is strictly necessary or the lock acquired is too strong for the required access.

Write-Back Proxies

Consider a slightly different scenario: objects that can be loaded on demand into memory and, when changed, written back. It is safe to assume that non-`const` operations will cause change and `const` operations won't, and that users will perform predominantly `const` or non-`const` operations on a given object in a particular role:

```
class datapoint {
public:
  double upper_bound() const;
  void upper_bound(double);
  double lower_bound() const;
  void lower_bound(double);
  ...
};
```

The only problem is that the objects are independent of your loading and saving framework. They do not contain convenient features that would make this problem trivial to resolve, such as a `dirty` flag to show when they've been modified. A simplistic, verbose, and error-prone approach would be to save or not in response to each call:

```
void view(const datapoint *target) {
  double = target->upper_bound();
                           // no save required
  ...
}
void manipulate(datapoint *target,
                double new_upper_bound) {
  ...
  target->upper_bound(new_upper_bound);
                           // save required
  save(target);            // assume a non-member
                           // save for target
}
```

This code also assumes that the object has been preloaded into memory.

Custom Proxies

The standard solution to such problems is to manage the level of indirection with a proxy object [4, 5]. A custom proxy, written to handle each of the member functions individually, can be written to encapsulate both the lazy loading and the save policy:

```
class datapoint_view {
public:
  double upper_bound() const {
    if(!target)
      load();
    return target->upper_bound();
  }
  void upper_bound(double new_upper_bound) {
    if(!target)
      load();
    target->upper_bound(new_upper_bound);
    save();
  }
  ...
private:
  void load() const;
  void save();
  mutable datapoint *target;  // null when
                              //    not loaded
  persistence_key target_key; // used by load
};
```

Clients now work in terms of `datapoint_view` instead of `datapoint`. Although conventional, proxies and targets do not necessarily have to inherit from a common base class, in this case, unless the `datapoint` author included an interface class for another reason, it may not even be possible to perform such inheritance without further adaptation.

However, although simple in many ways, this design is tediously repetitive. Each function follows a similar flow: load if not yet loaded, forward call, and then optionally a call to save state. It is also hardwired to a single target type. A generic solution would allow arbitrary data types to be managed in the same way. Smart pointers offer the most common idiom for such generic managed indirection.

Smart References

Consider first a simple case for a generic loading-and-saving smart pointer: working with built-in types or user-defined value types that, like built-ins, are manipulated only in terms of operators. The focus of operations is, therefore, on the dereferenced value. Another proxy variant, a smart reference, allows basic reads and writes to be distinguished:

```
template<typename target_type>
class loading_ptr {
public:
  ...
  class reference {
  public:
    explicit reference(const loading_ptr *that)
      : that(that) {}
    operator target_type() const {
      return *that->target;
    }
    reference &operator=(const target_type &rhs) {
      *that->target = rhs;
      that->save();
      return *this;
    }
  private:
    const loading_ptr *that;
  };
  reference operator*() const {
    if(!target)
      load();
    return reference(this);
  }
  ...
private:
  ...
  friend class reference;
  void load() const;
  void save() const;
  mutable target_type *target; // null when
                               //   not loaded
  persistence_key target_key;  // used by load
};
```

A smart reference cannot be a perfect match for a real reference [6], but such a limitation on transparency is true to a greater or lesser extent of any kind of proxy. For instance, the UDC (user-defined conversion) operator means that the result of dereferencing a `loading_ptr` is an rvalue rather than an lvalue. The disadvantage of the UDC as it stands is that it uses up the allotment of one user-defined conversion. A little generic thinking gets round this problem:

```
template<typename target_type>
class loading_ptr {
public:
  ...
  class reference {
  public:
    ...
    template<typename result_type>
    operator result_type() const {
      return *that->target;
    }
    ...
  private:
    loading_ptr *that;
  };
  ...
};
```

The downside is that this permissive conversion can create some fresh new ambiguities: the templated UDC now eagerly matching all possibilities unless explicitly disambiguated. You have to choose between evils according to which least affects your code – I guess that you probably don't need reminding that the world is not a perfect place.

Almost all the overloadable operators can be overloaded to make working with the smart reference reflect the natural use of its target type more accurately. This is fine for built-in types and like-minded value classes, but not for targets with named members: for them, `operator->` must be overloaded.

Bracketing Smart Pointers

But over-eagerness to implement `operator->` will lead you straight into a brick wall:

```
template<typename target_type>
class loading_ptr {
public:
  ...
  target_type *operator->() const {
    if(!target)
      load();
    return target;
  }
  ...
};
```

Sure, the object may have been successfully loaded into memory, but when and how will it be saved? The save needs to occur after the pointer has been returned and a member dereferenced through it – in other words, outside the body of `operator->` over which you have control.

It would appear that you can never have too many proxies: the solution is to return a smart pointer from `loading_ptr`'s `operator->` and have its destructor perform the save:

```
template<typename target_type>
class loading_ptr {
public:
  ...
  class pointer {
  public:
    explicit pointer(const loading_ptr *that)
      : that(that), accessed(false) {
    }
    target_type *operator->() const {
      accessed = true;
      return that->target;
    }
    ~pointer() {
      if(accessed)
        that->save();
    }
  private:
    const loading_ptr *that;
    bool accessed;
  };
```

```
    pointer operator->() const {
      if(!target)
        load();
      return pointer(this);
    }
    ...
  private:
    friend class pointer;
    ...
  };
```

What makes this idiom tick is that `operator->` chains: if the result of `operator->` also supports an `operator->`, this latter operator is called automatically, and so on until the result chain reaches a real pointer. What makes this idiom tock is the binding of a temporary object's lifetime to the end of the surrounding full expression. So, when used in a simple expression, the load is performed in the first `operator->`, which returns a temporary object used to access the members – either function or data – of the underlying target, and the save is performed by the destruction of the temporary object at the end of the expression:

```
  void manipulate(loading_ptr<datapoint> target,
                  double new_upper_bound) {
    ...
    target->upper_bound(new_upper_bound);
    //   ^load if necessary ^save
  }
```

As a point of implementation, note that you have to ensure that only one destructor execution performs the save because you are returning the pointer proxy by copy.

This versatile bracketing technique has acquired various names over time: locking pointer in the context of thread synchronization [7], call wrappers [8], and execute-around pointers [9].

The Depths of Qualification

But before you get too distracted or overwhelmed by the many levels and kinds of proxy you have at your disposal, remember that part of the design requirement was to distinguish between `const` and non-`const` usage. The code currently assumes the non-`const` case (i.e., always save). How can you distinguish a `const` target?

A common, but alas incorrect, solution to this problem is to reflect the qualification of the smart pointer in the qualification of the result:

```
  template<typename target_type>
  class loading_ptr {
  public:
    ...
    pointer operator->() {
      if(!target)
        load();
      return pointer(this);
    }
    const target_type *operator->() const {
      if(!target)
        load();
      return target; // return actual pointer as
                     //   no save needed
    }
    ...
  };
```

There are certainly designs in which you would want such deep qualification, but this isn't one of them. For a composition relationship through a pointer, where an object pointed to by another is considered a part of a whole, qualification should run deeply, just as it does for a data member by value. For association, where the relationship represented is not whole-part, qualification should be shallow, which is the case for many indirection-based relationships. Making it deep arises from confusion – albeit commonplace – over levels of indirection: the qualification of your smart pointer relates to whether or not you can affect the smart pointer itself, not its referand.

Many Roads Lead to Rome

There are two user roles with respect to target usage for the write-back proxy: a read-only role, which we can equate to accessing a const target, and a write-mostly role, which we can equate to accessing a non-`const` target and typically accessing non-`const` members. If deep qualification were the only solution to this kind of problem, iterators to const containers would be stuck on the starting blocks. The good news is that not only is there a solution, there are many solutions, each with a different set of tradeoffs.

Separately Qualified Types

Taking a leaf straight out of the standard library, the iterator model can be used as the inspiration for a pair of class templates:

```
  template<typename target_type>
  class loading_ptr {
  public:
    ...
    class pointer {
      ...
    };
    pointer operator->() const {
      if(!target)
        load();
      return pointer(this);
    }
    ...
  };
  template<typename target_type>
  class const_loading_ptr {
  public:
    const_loading_ptr(
              const loading_ptr<target_type> &);
    ...
    const target_type *operator->() const {
      if(!target)
        load();
      return target;
    }
    ...
  };
```

Note the converting constructor from a `loading_ptr` to a `const_loading_ptr`. This could also be supported by introducing a UDC to a `const_loading_ptr` in the `loading_ptr` template. Either way, a `loading_ptr` is substitutable for a `const_loading_ptr`. Note also that there is potentially a certain amount of duplicated code. A simple alternative that offers both substitutability and factoring of common code is to introduce an inheritance relationship:

```
template<typename target_type>
class const_loading_ptr {
  ...
};
template<typename target_type>
                 class loading_ptr
  : public const_loading_ptr<target_type> {
  ...
};
```

Some members will be fully "specialized" and "overridden" in the derived class (i.e., `operator->` will be provided in `loading_ptr` and will block the one in `const_loading_ptr` from view). This is compile-time polymorphism rather than run-time polymorphism – there are no virtual functions in sight, nor should there be.

Qualified Type Specialization

The `const_loading_ptr` solution is OK except that it leaves us with two different type names. This means that for an arbitrary type `T`, which may or may not be `const` qualified, you cannot simply write `loading_ptr<T>` and expect it to do the right thing (i.e., `loading_ptr<const datapoint>` is not equivalent to `const_loading_ptr<datapoint>`). Template specialization with respect to qualification offers a simplification [10]:

```
// primary class template
template<typename target_type>
class loading_ptr {
public:
  ...
  class pointer {
    ...
  };
  pointer operator->() const {
    if(!target)
      load();
    return pointer(this);
  }
  ...
};
// partial specialization
template<typename target_type>
class loading_ptr<const target_type> {
public:
  loading_ptr(
           const loading_ptr<target_type> &);
  ...
  const target_type *operator->() const {
    if(!target)
      load();
    return target;
  }
  ...
};
```

This means that one template name, `loading_ptr`, covers both cases, with `loading_ptr<datapoint>` corresponding to the primary template definition and `loading_ptr<const datapoint>` corresponding to the partial specialization. Inheritance can again be used to provide substitutability and cure the common code:

```
// forward declare primary class template
template<typename target_type>
class loading_ptr;
// partial specialization
template<typename target_type>
class loading_ptr<const target_type> {
  ...
};
template<typename target_type>
                        class loading_ptr
  : public loading_ptr<const target_type> {
  ...
};
```

It is worth pointing out that not all compilers support specialization with respect to qualification.

Explicit Qualification Check

There are only a few parts of the `loading_ptr` template that need to be different between the `const` and non-`const` variant. Instead of partially specializing the whole class template, why not just do so for the affected functions? This is an elegant and economic idea; unfortunately it also won't work: the C++ Standard currently disallows partial specialization of function templates. You can overload or fully specialize a function template, but alas neither of these options is directly suitable for the member functions in this particular problem.

However, the effect of `const` partial specialization can, to a limited degree, be emulated. Instead of focusing on the `loading_ptr`, focus on the result of `operator->`. Return the pointer proxy in each case and determine only in the destructor whether or not a save is required.

Here is a brute force run-time type checked approach:

```
template<typename target_type>
loading_ptr<target_type>::pointer::~pointer() {
  if(accessed &&
    typeid(target_type *)
               != typeid(const target_type *))
    that->save();
}
```

Because `typeid` ignores top-level qualifiers, the code is phrased in terms of pointers. The trick that allows the `const` discrimination to work is to recall that `const` qualifying something that is already `const` qualified has no effect. However, this approach lacks both elegance and economy. You can eliminate the run-time type check by introducing a predicate:

```
template<typename target_type>
loading_ptr<target_type>::pointer::~pointer() {
  if(accessed && !is_const(that->target))
    that->save();
}
```

Here, it is overloading with respect to `const` that allows the predicate approach to work:

```
template<typename type>
bool is_const(type *) {
  return false;
}
template<typename type>
bool is_const(const type *) {
  return true;
}
```

If these functions are inlined – and your compiler is doing at least a halfway decent job with inlining – there will be no run-time overhead in this approach. Compile-time selection is the territory of traits – a far more elegant approach:

```
template<typename target_type>
loading_ptr<target_type>::pointer::~pointer() {
  if(accessed && !is_const<target_type>::value)
    that->save();
}
```

The following mono-trait class template uses `const` partial specialization to make the distinction:

```
template<typename type>
struct is_const {
  static const bool value = false;
};
template<typename type>
struct is_const<const type> {
  static const bool value = true;
};
```

This approach can also be made to work using `const`-qualified pointers and partial specialization [11] if your compiler does not support direct `const` specialization.

Qualified Double Dispatch

There is a way to have selection without explicit control flow and to simulate the partial specialization of function templates. Double dispatch allows you to select an action on an object externally based on the type of the object. A family of functions performs the selection on your behalf calling back on the object you pass. This is normally described in terms of different classes in a hierarchy and forms the basis of the conventional form of the VISITOR pattern [5]. We can also frame double dispatch in terms of qualification, which is simply a different, more constrained form of subtyping.

Depending on the kind of extensibility you want in your design, you can define your dispatch functions either outside or inside the class. The following code uses class statics to retain some consistency with the previous solutions:

```
template<typename target_type>
class loading_ptr {
public:
  ...
  class pointer {
  public:
    ...
    ~pointer() {
      if(accessed)
        save(that); // select appropriate
                    //    save strategy
    }
  private:
    template<typename save_type>
    static void save(
             loading_ptr<save_type> *that) {
      that->save();
    }
    template<typename save_type>
    static void save(
           loading_ptr<const save_type> *) {
      // do nothing as no save is needed
    }
    loading_ptr *that;
    bool accessed;
  };
  ...
};
```

If overloading with respect to the full `loading_ptr` type does not work for your compiler, restructure the code so that you overload with respect to the target pointer.

Conclusion

Overloading with respect to qualification combines with templates to provide a valuable form of subtyping and specialization. Whilst not all of the techniques demonstrated are within reach of all compilers, there is enough overlap between their applicability to allow you some implementation wriggle room.

The `write-back` proxy allows a number of issues to be explored, although it is not intended to demonstrate all that you would need in such a design. For instance, matters of object lifetime and thread synchronization have been glossed over. There is also an important assumption in the applicability of the core design: if the non-`const` access is not write mostly there will be a lot of wasted saves. However, as C++ is currently defined there is no way that a proxy can both be made generic and distinguish on the qualification of the actual target member accessed, so this is unfortunately a natural constraint.

Kevlin Henney
kevlin@curbralan.com

References

[1] Kevlin Henney. “From Mechanism to Method: Good Qualifications,” *C/C++ Users Journal C++ Experts Forum*, January 2001, www.cuj.com/experts/1901/henney.htm
[2] Bjarne Stroustrup. *C++ Programming Language, 3rd Edition* (Addison-Wesley, 1997).
[3] Kevlin Henney. “C++ Patterns: Executing Around Sequences,” *EuroPLoP 2000*, July 2000, also available from www.curbralan.com
[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns* (Wiley, 1996).
[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).
[6] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, (Addison-Wesley, 1996).
[7] Kevlin Henney. “C++ Advanced Design Issues: Asynchronous C++,” *Visual Tools Developers’ Academy* (Oxford, September 1996).
[8] Bjarne Stroustrup. “Wrapping C++ Member Function Calls,” *C++ Report*, June 2000.
[9] Kevlin Henney. “From Mechanism to Method: Substitutability,” *C++ Report*, May 2000, also available from www.curbralan.com
[10] Kevlin Henney. “From Mechanism to Method: Distinctly Qualified,” *C/C++ Users Journal C++ Experts Forum*, May 2001, www.cuj.com/experts/1905/henney.htm
[11] Boost library website, www.boost.org

This article was originally published on the C/C++ Users Journal C++ Experts Forum in September 2001 at http://www.cuj.com/experts/documents/s=7991/cujcexp1909Henney/
Thanks to Kevlin for allowing us to reprint it.