HackCraft

Templates and Inheritance

The introduction of templates to C++ brought a new paradigm to C++ coding—Generic Programming. This is now a major part of the toolkit of the C++ programmer, the basis of much of the standard library, and something which many of us younger C++ hackers never experienced C++ without. Generic Programming is often discussed in contrast with Object Orientated Programming's concept of inheritance. However a truly multi-paradigm approach prompts us to examine how the two interact.

Contents

10 References

A Note on Efficiency

There is perhaps rather too much made of efficiency in my writings as a whole. This is for a few different reasons. In some cases a technique where the primary or sole practical advantage is one of efficiency is interesting enough to me that I want to write about it, in other cases I address the issue of efficiency in order to allay criticism that a technique (such as RAII) or a technology (such as Unicode) may be less efficient than possible alternatives.

I'm afraid efficiency is going to raise it's ugly head again here, for two reasons. The first is that, yet again, some of the techniques that will be mentioned are of practical value primarily as they relate to efficiency. The second is that with many compilers lacking support for export with templates (Microsoft's VisualC++ to my certain knowledge, and I'm told the same is true with most others) the use of templates entails the use of inline functions, and hence often a optimisation of speed over size (though in some cases, a few of which are significant here, inlining can improve both speed and size, and hence be a "perfect" optimisation, on the other hand in a few cases it can hurt both).

Because of this, it's worth considering the thorny issue of efficiency for a moment. Knuth famously said "premature optimisation is the root of all evil". This saying is well-known among hackers, unfortunately it is, along with Einstein's statement on simplicity, a truism—of course premature optimisation is a Bad Thing, premature anything is a Bad Thing, that's why we call it premature. Knuth's statement does not automatically condemn optimisation, or even early optimisation. there are three cases where it is wise to consider efficiency from the very beginning:

Choice of Architecture

An architecture that allows for caching of responses between communicating parties will be more efficient than one that does not—often to the degree of dictating whether something is a viable approach. Other architectural decisions, that may by necessity have to be made early in the design process, will similarly affect performance. I can't think of a way that this affects this document, but I'm stating it for the sake of completeness.

Efficient Habits

Sometimes two different approaches to the same task may require equal amounts of mental effort when you are coding, or reading the code, but may differ in terms of efficiency. In such a case it can be worth getting into the habit of using the more efficient of the two, even when the difference is slight. As an example if you do not use the returned value there are cases where `++i` would be slightly more efficient than `i++`. There are rare occasions when it would be significantly more efficient. In practice you

may never find a case where these make any difference that isn't neutralised by the compiler's own optimisation—but hey, it's not like you've caused any harm.

Good Choice of Algorithm

In many cases this isn't considered a matter of optimisation at all, although it does have a strong impact on efficiency. No matter how well you optimise it a bubble sort of 1000 items won't compete with any reasonably well-written quicksort. Ignoring strawman cases, the choice of algorithms will generally have a greater effect on the overall efficiency of a piece of code than any other factor; and the ideal algorithm will vary according to quite a few factors, some related to the machine architecture, some related to the datatypes involved, and some related to the application domain. As will be seen some of the techniques we will examine will enable developers to easily change algorithms to suit the application domain and experiment with the performance of each, others will enable to compiler to choose algorithms so that an efficient one for the datatypes involved will be chosen, without requiring the developer to make a conscious decision in this regard.

Another aspect of this is that it helps to write code in such a way that you can switch algorithms easily. This allows you to use a straight-forward approach and replace it with a more efficient, but more complicated, algorithm later if necessary. In the context of this document it's worth noting that templates in general, and the approach taken by the STL in particular, ease this task considerably.

Optimisation to Pre-empt Premature Optimisation

There is more than a slight degree of irony here, but one of the best ways to discourage premature optimisation on the part of a developer is to give that developer heavily optimised tools.

The advantages of code reuse should be so well-known that they do not need to be repeated here, but what if the available components are less efficient than the "hand-written" equivalents? In this case the developer has a dilemma, she can roll her own and gain in performance, or use an available component and gain the advantages in terms of reliability, source readability, and development time. Too often we find ourselves forced to do the former, if only to hedge our bets. Efficient components make reuse a win-win choice, and some of the techniques discussed below can help library developers to build such components.

(This is also comparable to the case of the optimisations performed by the compiler itself, I would be enraged if I had to deal with code where a colleague had repeatedly used `x^=x` instead of `x=0` just because the former was the more efficient idiom on an Intel processor, but I'm perfectly happy for the compiler to make the equivalent optimisation when producing machine code.)

Templates as an Alternative to Inheritance

It's worth beginning by examining cases where templates provide functionality similar to that of inheritance, in some cases so similar that there is a choice of using either a template-based approach or an inheritance-based approach.

Policy Classes

Policy classes provide either a different approach to the same problem based on application-specific criteria, or offer a different implementation of the same task. The difference between the two is more conceptual than concrete, but examining it in each case helps explain how it can be used in practice.

An example of the first case is a class that comparisons of strings being used to sort strings. There are various different ways we can sort strings. The simplest way is a simple comparison of the numerical values of the characters. This is of little value when presenting results to an end-user but is the most efficient when, as is often the case, we need a mechanism for ordering items but it doesn't really matter what that mechanism is, only that the same strings will always be ordered in the same way. Then there are different orders for different languages and/or countries and sometimes different contexts (some countries have a different order in dictionaries than in telephone directories). There are case-sensitive and case-insensitive sort orders. There are case-insensitive sort orders that consider `i` equivalent to `I` and case-insensitive sort orders that consider `i` equivalent to `İ`.

None of these differences in how we determine which string comes before which affects the rest of a sorting operation (how we determine which to compare and how we move strings). Hence it makes little sense to write different sort operations for case-sensitive sorts, case-insensitive sorts, and so on. This becomes even less sensible if, as well as offering a variety of sort criteria, we wish to offer a variety of sort algorithms, deal with a variety of character encodings, deal with different containers and make use of a swap operation that is efficient with a given datatype.

The solution before C++ had templates (and still the solution with C) is to make the operation that compares elements a function and pass a function pointer to the sort function. This provides the necessary flexibility, but requires a function call for each comparison.

If we use a function object, and make the type of that function object a template parameter to the sort, then it becomes easier to inline the call to the comparison operation.

This is how the STL sort functions work.

A variant on this is a class whose, often static, members provide functionality to another class. The STL `char_traits` class is one example, `basic_string` classes and other users

of traits classes can call into `eq(Ch x, Ch y)` to discover if `x` is equal to `y` without knowing how this is implemented. Similarly ATL provides a class to lock and unlock a thread, and another where the namesake member functions are dummy (no operation) functions that are optimised away entirely. This allows ATL to easily create thread-safe versions of a class without incurring overhead in cases where thread-safety is not needed, or is already provided by the COM apartment model.

Without templates virtual functions could be used to provide the same functionality. However since virtual functions are generally implemented through a table of function pointers—or else by some other mechanism that cannot be easily inlined—this can result in a severe overhead in the case of a large sort operation, or undo the benefit of not locking a thread when it isn't necessary.

Tag Classes

Tag classes are classes which identify traits of another class.

Consider the task of advancing an iterator by an arbitrary number of iterations. In the case of a random-access iterator (including a pointer into an array) this can be done in constant time, we simply add the appropriate number to the iterator:

```
template<typename Ran, typename Dist> void advance_random_iter(Ran& it, D
{
    it += offset;
}
```

With an input, or bi-directional iterator we cannot do this, and so we have to use the linear time method of incrementing the iterator a sufficient number of times:

```
template<typename In, typename Dist> void advance_input_iter(In& it, Dist
{
    while (0 < offset--)
        ++it;
}

template<typename Bi, typename Dist> void advance_bidirectional_iter(Bi&
{
    for (; 0 < offset; --offset)
        ++it;
    for (; offset < 0; ++offset)
        --it;
}
```

In each case we have the most efficient way of producing the desired result. We also don't attempt to perform impossible operations (decrementing an input iterator). However we have three separate functions depending on the type of iterator. We could have a virtual advance member on all of our iterators—declared in a base common to all iterators—but this overhead is severe for iterators; which are commonly operated upon inside a very tight loop, the sort of case where we most want to avoid the overhead of a virtual function call. We could use `advance_input_iter()` for all iterators, or `advance_bidirectional_iter` if we might be supplying a negative offset, but if this were carried to its logical conclusion we'd only ever be performing those operations that are common to all iterators (which is pretty much null set) except where a given operation is specifically necessary to a particular job (an approach that would soon have us ignoring any commonality between iterators and losing the advantage of having the concept of "iterator" in the first place).

The solution is to provide iterators with a tag member typedef. In a random access iterator we have the following public typedef:

```
typedef random_access_iterator_tag iterator_category.
```

In an input iterator we have the following public typedef:

```
typedef input_iterator_tag iterator_category.
```

Hence the type of `It::iterator_category` will vary according to the basic category of iterator It is; input, output, forwards-only, bi-directional or random-access. Rather than look at `It::iterator_category` though, we look at `iterator_traits<It>::iterator_category`. The template `iterator_traits` looks like:

```
template<typename Iter>struct iterator_traits{
    typedef typename Iter::iterator_category iterator_category;
    typedef typename Iter::value_type value_type;
    typedef typename Iter::difference_type difference_type;
    typedef typename Iter::pointer pointer;
    typedef typename Iter::reference reference;
};
```

So going through `iterator_traits` is just a roundabout way of getting to the same thing (at compile-time). However there is also a partial specialisation of `T*` and `const T*`:

```
template<typename T>struct iterator_traits<T*>{
    typedef typename random_access_iterator iterator_category;
```

```
    typedef typename T value_type;
    typedef typename ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};

template<typename T>struct iterator_traits<const T*>{
    typedef typename random_access_iterator iterator_category;
    typedef typename T value_type;
    typedef typename ptrdiff_t difference_type;
    typedef const T* pointer;
    typedef const T& reference;
};
```

So going through `iterator_traits` allows us to get the `iterator_category` tag for pointers as well as iterators classes.

Now returning to our decision as to whether to use `advance_random_iter()`, `advance_bidirectional_iter()` or `input_random_iter` in a given case. We can re-create these three functions as overloaded versions of a function `advance_helper()`, which take iterator_category tags as well as the iterator and offset:

```
template<typename Ran, typename Dist>inline void advance_helper(Ran& it,
{
    it += offset;
}

template<typename In, typename Dist>inline void advance_helper(In& it, Di
{
    while(0 < offset--)
        ++it;
}

template<typename Bi, typename Dist>inline void advance_helper(Bi& it, Di
{
    for (; 0 < offset; --offset)
        ++it;
    for (; offset < 0; ++offset)
        --it;
}
```

Then we call one of them from a template function `advance()`, relying on overload resolution to select the correct `advance_helper()` to call:

```
template<typename In, typename Dist>void advance(In& it, Dist offset)
{
    advance_helper(it, offset, iterator_traits<In>::iterator_category());
}
```

The creation of the `iterator_category` object, and its being passed to the `advance_helper()` function, is easily removed by an optimising compiler, so we are left with a function that will advance either an input iterator, a bi-directional iterator or a random-access iterator by the most full-featured and efficient method possible, without the need for either the coder nor a run-time selection mechanism to do any work.

Templates Engaging in Inheritance Relationships

Okay, as an alternative to inheritance relationships templates offer us mechanisms to perform a similar feat of producing different behaviour for the same function call or operator, but performing the necessary lookup at compile-time rather than run-time (and hence sometimes being referred to as “compile-type polymorphism” as opposed to “run-time polymorphism”). The main reason for preferring this is to remove the overhead involved in run-time resolution, particularly in cases were operations are likely to be performed inside a tight loop.

However the classes created from templates are normal classes, and can take have sub- and super-classes just like any other. The interactions between the template mechanism and the inheritance mechanism are largely what one would expect, but there are a few interesting cases. We’ll deal with the least interesting first (watch how I build dramatic tension…).

Templates with a non-template base

The least interesting case is that of templates with a non-template base. For the most part this is done as a simple example of code reuse—there is code that will be used in all of a template's instances and it doesn't depend upon the template parameters, hence it can go into a separate base class. This is akin to the way we search for commonality between classes in any object-orientated design work, with the added practical advantage in the case of templates with today's compilers that it is possible to have non-inline member functions in the base that will be inherited by the template classes (again, this is perfectly possible with template member functions according to the Standard [ISO/IEC 14882] if you use the `export` keyword in the function definition, but unfortunately support for this is not currently great). This can dramatically reduce the size of code produced if there are a lot of different instances of the template class in question.

Share typedefs and enums that related to the template classes' tasks but which do not relate to their template parameters is a particularly common case. For example the std::codecvt template class has a public base that looks like this:

```
class codecvt_base{
public:
    enum result {ok, partial, error, noconv};
};
```

This enum is used by several member functions of the `std::codecvt` template class, and does not relate to the template parameters. Hence it can exist in a separate base class.

Giving a template class a non-template base is sometimes used to counter one of the places where the rules for compile-time polymorphism differs from the rules for run-time polymorphism—with run-time polymorphism all off the types which share a common base can be manipulated through a pointer or reference to that base. This is not the case of types which are instances of the same template; there is no commonality between `vector<int>` and `vector<long>` that can be used at run-time. There are rare occasions where such commonality can be useful, and in this case a template type with a non-template base offers this. Tree-like object models often lend themselves to this combination; they are composed of objects which have child collections of other objects which are themselves part of the tree-structure, but some objects may have restrictions on the type of child objects they can contain.

In this case covariant return types are often useful as well; there is a relaxation of the rule that a virtual function can only be over-ridden by a function with the same signature that allows the overriding function to have a more restrictive return type (a function returning a `shape*` can be over-ridden by a function returning a `circle*` and

so on). The covariant rule is an example of the general rule that a derived class must deliver everything the base class guarantees, but can add its own guarantees.

We can create a node object which holds the necessary information (pointer to parent node, pointer to the next and previous sibling nodes and pointers to the first and last child nodes) and override the functions that return this information in subclasses with covariant functions that restrict the type of node returned. While these subclasses aren't necessarily template classes, they often are.

A variation on this principle is used by the STL for vectors of pointers. First the STL instantiates `vector<void*>`. While `vector<void*>` is a template class, once instantiated it is pretty much the same as a "normal" class, the fact that it was created from a template no longer affects anything.

Then the STL provides a specialisation of `vector<T*>`. This will be used for any vector of pointers, except for a vector of `void*` (since we already have a `vector<void*>` and `void*` is more specific than `T*`).

`vector<T*>` is declared to have a private base of `vector<void*>`. All functions which place a new element into the vector, such as `push_back()`, call the equivalent function on this private base, so internally our `vector<T*>` is using a `vector<void*>` for storage. All functions which return an element from the vector, such as `front()`, perform a `static_cast` on the result of calling the equivalent function on the private base. Since the only way to get a pointer into the `vector<void*>` (apart from deliberately dangerous tricks) is through the interface offered by `vector<T*>` it is safe to staticly cast the `void*` back to `T*` (or the `void*&` back to `T*&`, and so on).

The only work done by the `vector<T*>` itself is static casting to and from `void*`, which is easily inlined (and frequently a null operation at the level of the produced machine code; the binary representation of the `T*` and the `void*` may well be identical). Since the tricky work is being done by the `vector<void*>` any functions that will be large in the produced executable will be shared between `vector<long*>`, `vector<int*>`, `vector<vector<int*>*>` and so on. The result of this can be a considerable saving in executable size and compile time, with no penalty in performance. Without the specialisation `vector<T>` would have done the same job adequately, if in a somewhat bloated executable, so really the separate definition of `vector<T*>` is purely a matter of optimisation—but one that is invisible to the user, and hence not one that is "premature".

Non-Template Classes with a Template Base

I examined the case of `vector<T*>` being derived from `vector<void*>` as an example of a template class derived from a non-template class. Of course I was stretching my point a bit given that `vector<void*>` is created from a template. Deriving from a template based class is generally a matter of code-reuse rather than a publicly accessible "is a" relationship modelled using inheritance. This is reflected in the fact that the inheritance

is private, rather than public. In practice the difference between private inheritance rather than private membership is generally more one of taste than anything else, though it can be beneficial to say "these two classes are in some way similar" which the "is a" relationship between `vector<T*>` and `vector<void*>` indicates—as opposed to merely saying that one class uses the other.

Consider the case of wide character streams that read from or write to UTF-8 streams.

When a wide-character stream, such as `std::winstream` or `std::wostream`, reads or writes it does so through the services provided by a `std::codecvt<wchar_t, char, std::mbstate_t>` object (provided by the locale). It calls on public member functions which call on protected virtual members.

For brevity we'll examine only what happens when an `std::winstream` that operates on UTF-16 code units reads characters from a UTF-8 source.

A class is needed that does this conversion, we'll call it `u8toU16`. Since `std::winstream` wants to use an `std::codecvt<wchar_t, char, std::mbstate_t>` we'll define it thus:

```
class u8toU16 : public std::codecvt<wchar_t, char, std::mbstate_t>{
};
```

`std::codecvt` provides public member functions, the one called by `std::winstream` is:

```
result in(
    mbstate_t& state,
    const char* from,
    const char* from_end,
    const char*& from_next,
    wchar_t* to,
    wchar_t* to_limit,
    wchar_t*& to_next) const;
```

This will convert as many characters as possible starting with that pointed to by `from` up to (but not including) that pointed to by `from_end`, and place them in `to` up to (but not including) `to_limit`. `from_next` and `to_next` are then updated to show how far the call has progressed through these arrays, and a result returned to show why it stopped.

This function may do some housekeeping before or afterwards, but the main thing it does is to call a protected member function:

```
virtual result do_in(
    mbstate_t& state,
    const char* from,
```

```
        const char* from_end,
        const char*& from_next,
        wchar_t* to,
        wchar_t* to_limit,
        wchar_t*& to_next) const;
```

This is the function that does the actual conversion. Lets look at how it might work in our `u8toU16` class:

```
int u8toU16::appendU8(int start, const char* pointer, size_t count){
    /*private implementation function*/
    while(count--){
        char next = *pointer++
        if (next & 0xC0 != 0x80)/*Invalid, and insecure*/
            return 0;
        start = (start < 6) | (next & 0x3F);
    }
    return start;
}

result u8toU16::do_in(mbstate_t& state, const char* from, const char* fro
{
    from_next = from;
    to_next = to;
    while(from_next != from_end && to_next != to_limit){
        char u8char = *from_next;
        if (u8char < 0x80){
            *to_next++ = u8char;
            ++from_next;
        }
        else if (u8char < 0xC2 || u8char > 0xF4){
            return error;
        }
        else if (u8char < 0x800){
            if (from_end - from_next == 1)
                return partial;
            wchar_t res = appendU8(u8char, from_next + 1, 1);
            if (res == 0)// appendU8 found bad octet
                return error;
            *to_next++ = res;
            from_next += 2;
        }
        else if (u8char < 0x10000){
            if (from_end - from_next == 1)
                return partial;
            wchar_t res = appendU8(u8char, from_next + 1, 2)
            if (res < 0x800)// appendU8 found bad octet or other “short f
```

```
                return error;
            if (res & 0xF800 == 0xD800)// surrogate value—invalid
                return error;
            *to_next++ = res;
            from_next += 3;
        }
        else{
            if (from_end - from_next < 3 || to_limit - to_next == 1)
                return partial;
            int res = appendU8(u8char, from_next + 1, 3);/*int must be at
            if (res < 0x10000)// appendU8 found bad octet or other “short
                return error;
            if (res > 0x10FFFF)// invalid
                return error;
            res -= 0x10000;
            *to_next++ = (res >> 10) | 0xD800;
            *to_next++ = (res & 3FF) | 0xDC00;
            from_next += 4;
        }
    }
    return (from_next == from_end) ? ok : partial;
}
```

The above function (which is neither the most efficient or the most readable possible, but somewhere in between) is specialised to the specific case of a UTF-16 stream based on an underlying UTF-8 stream. The class it is derived from is specialised just enough to be used with a `wchar_t` stream based on an underlying UTF-8 stream.

The choice of what is done in the template, and what is done in the derived class gives a good balance that makes the stream classes general enough to be used widely, but specialised enough to deal with any possible character encoding; the compile-time polymorphism of determines the types used, the run-time polymorphism determines the implementation. The use of virtual functions on buffered data, but only there, is a good balance between efficiency and run-time flexibility.

Templates with a Template Base

In my earlier example of a template with a non-template base I cheated in having `std::vector<void*>` my non-template base. The distinction however lies between cases where a template class derives from, or is derived from, another template class, and where a template class derives from a template class which shares one or more template parameters with it.

This allows us to effectively parameterise an entire class hierarchy. Imagine a simple DOM-style XML hierarchy which contains a node class and element, textnode and attribute classes derived from it:

```
class node{/* … */};
class element :public node{/* … */};
class textnode :public node{/* … */};
class attribute :public node{/* … */};
```

Nothing terribly novel here. Now lets say our DOM does not take the W3C `<http://www.w3c.org>` approach (the W3C DOM `<http://www.w3.org/DOM/>` mandates the use of UTF-16) and allows users to parameterise string types. We can then have:

```
template<typename Str> class node{/* … */};
template<typename Str> class element : public node<Str>{/* … */};
template<typename Str> class textnode : public node<Str>{/* … */};
template<typename Str> class attribute : public node<Str>{/* … */};
```

An alternative, and one I would generally lean towards, would be to separately define character, character traits and allocator types, much as `std::basic_string` does:

```
template<
    typename Ch,
    typename Tr = std::char_traits<Ch>,
    typename Al = std::allocator<Ch> >
    class node{
public:
    typedef std::basic_string<Ch, Tr, Al> string_type;
/* … */
};
template<
    typename Ch,
    typename Tr = std::char_traits<Ch>,
```

```
    typename Al = std::allocator<Ch> >
    class element : public node<Ch, Tr, Al>{/* … */};
template<
    typename Ch,
    typename Tr = std::char_traits<Ch>,
    typename Al = std::allocator<Ch> >
    class textnode : public node<Ch, Tr, Al>{/* … */};
template<
    typename Ch,
    typename Tr = std::char_traits<Ch>,
    typename Al = std::allocator<Ch> >
    class attribute : public node<Ch, Tr, Al>{/* … */};
```

This makes particular sense if we are going to make use of either the character type or the allocator type directly, (perhaps even rebinding the allocator to use it to allocate and deallocate child nodes).

In some cases this can lend itself to the use of templates with template parameters:

```
template<
    typename Ch,
    typename Tr = std::char_traits<Ch>,
    typename Al = std::allocator<Ch>,
template attCon<typename, typename> = std::vector>
    class element : public node<Ch, Tr, Al>{
    typedef attCon<attribute<Ch, Tr, Al>, Al::rebind<attribute<Ch, Tr, Al
    attributeCon attributes;
    /* … */
};
```

This enables us to change how attributes are stored while remaining within the hierarchy rooted at `node<Ch, Tr, Al>` and containing `attribute<Ch, Tr, Al>`. However it's probably gilding a lily in this case.

Inheritance Relationships in Tag Classes.

The main focus of inheritance in a design is often around how the virtual functions in the subclass override those in the superclass. Tag classes don't have virtual functions, or any function, since they are intended to be simple to the extent that after the compiler runs they won't "exist" in the produced code. However it is still possible to treat an object of a tag class as an object of the super class.

Looking at the tag classes example above, we have a way to advance input, bi-directional and random-access iterators, but what about forward?

In this case we can treat such iterators as input iterators; the method for advancing them is the same. For other operations though we would want a different mechanism to be used for these classes, so giving them iterator-category classes of `input_iterator_tag` is not a viable solution.

Examining how iterators work, there is an inherent inheritance relationship between the categories of iterators; random-access iterators have all the functionality of bi-directional iterators, which have all the functionality of forwards iterators, which have all the functionality of input iterators (and arguably output iterators). Hence the `random_access_iterator_tag` is derived from the `bidirectional_iterator_tag`, which is derived from the `forward_iterator_tag` which is derived from the `input_iterator_tag`. The practical effect of this is that in the case of a function like `advance()` above the `advance_helper` for input iterators will also be called for forward or bi-directional iterators, however with other functions we are free to specialise behaviour in a different way.

The `forwared_iterator_tag` class isn't derived from the `output_iterator_tag` as well as the `input_iterator_tag`. Strousup described the reasons for this as "obscure and probably invalid" [STROUSTRUP]. If you are interested in just what these obscure reasons are I suppose you may take it up with him! In terms of practical code though there isn't much harm in having just the single inheritance from `input_iterator_tag`—if there were functions specialised for both input and output iterators then you would need a specialisation for forward iterators in order to resolve the conflict about which function should be called. As such using multiple inheritance to have `forward_iterator_tag` derived from both would actually require more coding, not less.

Templates Parameterised by a Base Class

In the last case we were passing template parameters up to a base template. Another way to do this is to have the base class itself as a template parameter.

The `reverse_iterator` class in the STL could conceivably work like this by using a private base. It doesn't, but it could. Instead it uses parameterised members, and as a rule this is easier.

One use of templates parameterised by a base class is to allow you to provide an extension for a range of similar classes, say containers that called a function whenever they were added to.

Templates Parameterised by a Derived Class

And now for our final case where templates and inheritance interact—templates parameterised by a derived class.

```
template<typename T> class base{
    /* … */
};

class derived : private base<derived>{
    /* … */
};
```

If that looks dicey to you, you're not alone. Let's examine what happens when we create a class of type derived before we actually look at why we would do this.

To begin with we have our base template class. It and some of its members depend on `T`, but `T` is not known yet, so those members (whether fields or functions) that depend on `T` are not instantiated until the template is used.

Now, let us consider some of the ways in which a class or function might use a type's name:

1. Being derived from it.
2. Having a member of that type.
3. Having a member of a pointer or reference to that type.
4. Returning that type in a function declaration.
5. Returning a pointer or reference to that type in a function declaration.
6. Returning that type in a function definition.
7. Returning that a pointer or reference to type in a function definition.
8. Finding the sizeof that type.

Now, what do these uses have in common, and how do they differ?

Well, they all need the name to be available to them, but only some need the layout:

```
class someType; /*declare but don't define*/

class someOtherType : public someType{
    /* compile error—need to know the layout */
};

class yetAnotherType{
private:
    someType x; // error
    someType* p; // ok
    someType someFunction(); //ok
    someType* sillyFunction() //ok (but pointless!)
    {
        return 0; /* About the only thing we can directly do here,
                  although we could call on another function that
                  would give us a someType */
    }
};

someType yetAnotherType::someFunction()
{
    /* can't have this function definition,
    also any reasonable contents for this function
    would depend on knowing the layout of someType. */
}
```

So, what happens when the compiler encounters derived above.

First it makes the name derived available. Then it instantiates the layout of `base<derived>`—this requires knowledge of the member variables and virtual functions, as long as none of that depends upon knowledge of derived's layout this will all go okay. Then it will create the layout of derived, and finally it will create derived's member functions, which may include creating member functions for `base<derived>`. So as long as `base<derived>` doesn't contain a derived member variable (and base classes can never contain a member variable of a type derived from themselves) or a virtual function that requires knowledge of derived's layout we can indeed do the dicey-looking piece of inheritance above.

Now, why would we bother?

An interesting thing to note about `base<T>` is that none of it's member functions are created until after the type `T` is. As such the relationship between derived and `base<derived>` is known, and it is possible to safely perform a `static_cast<T*>(this)`, `static_cast<T&>(*this)` etc. within `base<T>`. This normally isn't the case because a base classes functions are normally compiled before any derived classes are.

This gives us an inline-able alternative to virtual members, allows us to know the size of the derived class (if the base class performed some sort of low-level memory allocation task) and other low-level ways to usurp the normal C++ inheritance methods that should generally be avoided but are good to have in our toolkits just in case.

Base classes such as these tend to be private or protected bases, rather than public. Clearly the inheritance relationship isn't going to be part of an class hierarchy, since derived is the only class that will be derived from `base<derived>`, and the purpose of a template like `base<T>` will generally to provide services to derived, rather than to use inheritance to model an "is a" relationship in the application domain. However, public bases of such types are not unknown. In either case it's worth noting that the access levels can be quite strange. Generally the only method a base class has of accessing a derived class is to call a virtual member. Since this virtual member is declared in the base class itself the base class always has access to it. In the case we are examining here though the base is "looking in from the outside", and so can only access public members—even private or protected members that the derived class has inherited from the base are *verbotten*. Because of this it's common for the derived class to make the base class a friend.

For a rich source of examples look at ATL; ATL works through providing templates that are used as bases of the class the user is creating. Many of these classes are parameterised by the derived class. They provide implementation for various COM-related tasks and use the above technique to do so efficiently (run-time performance was a major design goal with ATL).

Because the `base<T>` type is only sensible in the context of being used as a base for type `T` it is common to ensure it is only created and destroyed by the type `T` by making the constructors and destructors protected, or making them private and making `T` a friend.

This sometimes leads to a debate as to whether such classes should be called "abstract classes" or not since they cannot exist independently, but they do not have pure virtual members.

The answer is, yes they are abstract classes—however they do not use the C++ language feature specifically designed to support abstract classes (yet again a question of what a language supports differing from what it supports directly).

A (Slightly) Practical Example

Okay, just when would we want to avoid virtual calls so much.

Well I have an example already, in my example code to WNDPROC Thunks Using the thiscall Calling Convention. Here's the example again:

```
//Our example window class. This is pretty minimal, it just has the funct
//we need to make our point.
class exWinCls{
```

```
        winThunk<exWinCls> m_thunk;//Our thunk object
        const ::HWND m_handle;//Window handle

        //The message handler called by the thunk.
        ::LRESULT wndProc (::HWND hWnd, ::UINT uMsg, ::WPARAM wParam, ::LPARA
            switch(uMsg){
                case WM_DESTROY:
                    ::PostQuitMessage(0);//Goodbye, cruel world.
                    return 0;
                case WM_PAINT:
                    try{
                        raiiPaint rp(hWnd);
                        paint(rp);//The real work is done here.
                    }
                    catch(raiiPaint::error&){
                        /*we won't actually do anything about this,
                        however note that it prevents an ill-advised paint()
                        or EndPaint()*/
                    }
                    return 0;
                default://All other messages have default behaviour.
                    return ::DefWindowProc(hWnd, uMsg, wParam, lParam);
            }
        }
    public:

        //If it was a real general-purpose windows class, and we liked to do
        //in a type-safe way then we might have enums where the API uses cons
        //from define macros. Here we have just one example.
        enum showWindowType{
            hide        =SW_HIDE,
            normal      =SW_SHOWNORMAL,
            showNormal  =normal,
            showMinimized   =SW_SHOWMINIMIZED,
            showMaximized   =SW_SHOWMAXIMIZED,
            maximize    =showMaximized,
            showNoActivate  =SW_SHOWNOACTIVATE,
            show        =SW_SHOW,
            minimize    =SW_MINIMIZE,
            showMinNoActive =SW_SHOWMINNOACTIVE,
            showNA      =SW_SHOWNA,
            restore     =SW_RESTORE,
            showDefault =SW_SHOWDEFAULT,
            forceMinimize   =SW_FORCEMINIMIZE
        };

        //Return the current message handler
        ::WNDPROC windowProcedure() const{
            return reinterpret_cast<::WNDPROC>(::GetWindowLong(m_handle, GWL_
```

```
        }

        //Set the message handler, returning the previous.
        ::WNDPROC windowProcedure(::WNDPROC newProc){
            return reinterpret_cast<::WNDPROC>(::SetWindowLong(m_handle, GWL_
        }

        //We'll make this function virtual, it seems the sort of function tha
        //be suitable for overriding. Although we use (indirectly) the window
        //we don't pass it from the message handler—the advantage in efficien
        //does not sufficiently offset the risk of weird behaviour should an
        //over-riding class pass through the wrong handle.
        virtual void paint(::PAINTSTRUCT& ps){
            typedef std::basic_string<::TCHAR> tstring;
            static const tstring quote =
                TEXT("\"...it has been truly said that hackers have even more
                TEXT("for equipment failures than Yiddish has for obnoxious p
                TEXT("\n\n\t-\u00A0jargon.txt");
            ::RECT rcDraw = clientRect();
            ::OffsetRect(&rcDraw, 5, 5);
            ::InflateRect(&rcDraw, -10, -10);
            ::DrawText(ps.hdc, quote.data(), static_cast<int>(quote.length())
        }
    #pragma warning(push)
    #pragma warning(disable : 4355)
        //Our constructor. Note that we are naughty once again and use the th
        //pointer in the initializer list—not something to do in code that is
        //meant to be even remotely portable.
        exWinCls(LPCTSTR lpClassName,
            LPCTSTR lpWindowName,
            HINSTANCE hInstance)
            :m_handle(::CreateWindow(lpClassName,
                lpWindowName,
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                0, 0,
                hInstance,
                0))
            ,m_thunk(wndProc, this){
            windowProcedure(m_thunk);
        }
    #pragma warning(pop)

        //some simple and straight-forward wrappers.
        bool showWindow(showWindowType type){return ::ShowWindow(m_handle, ty
        bool updateWindow(){return ::UpdateWindow(m_handle) != 0;}
```

```
        ::RECT clientRect() const{
            ::RECT ret;
            ::GetClientRect(m_handle, &ret);
            return ret;
        }

    };
```

One problem with this code is that it isn't very reusable. If we wanted to package this up into a general-purpose library class we could do something like make `virtual void paint(::PAINTSTRUCT& ps)` a pure virtual function. However that increases the runtime cost of a very frequent call.

Do we care? Well, generally we don't, but maybe we would some of the time. Following the principle above that libraries should take greater pains towards efficiency than other code—to avoid users being tempted into rolling their own faster versions, it makes sense that this should be as fast as possible. The alternative is to change class to be parameterised by it's own subclass:

```
    template<typename Derived> class exWinCls{
        winThunk<exWinCls<derived> > m_thunk;
        Derived* down_cast() throw()
        {
            return static_cast<Derived*>(this);
        }
        const Derived* down_cast() const throw()
        {
            return static_cast<const Derived*>(this);
        }

        /* … */

        ::LRESULT wndProc (::HWND hWnd, ::UINT uMsg, ::WPARAM wParam, ::LPARA
            switch(uMsg){
                case WM_DESTROY:
                    ::PostQuitMessage(0);//Goodbye, cruel world.
                    return 0;
                case WM_PAINT:
                    try{
                        raiiPaint rp(hWnd);
                        downcast()->paint(rp);//The real work is done here.
                    }
                    catch(raiiPaint::error&){
                        /*we won't actually do anything about this,
                        however note that it prevents an ill-advised paint()
                        or EndPaint()*/
```

```
                }
                return 0;
            default://All other messages have default behaviour.
                return ::DefWindowProc(hWnd, uMsg, wParam, lParam);
        }
    }

    /* … */

};
```

We now have a choice between giving a default paint function that does little or nothing (most likely calling `::DefWindowProc()`) or of leaving it out and hence forcing a deriving class to implement it (equivalent in some ways to a pure virtual function).

Summary

So we have seen both how template-based “compile-time polymorphism” can compete with traditional run-time polymorphism, and how the two can work together. In general run-time polymorphism remains the best way to model “is a” relationships between classes. Template-based techniques come into their own in expressing algorihms in a type-agnostic fashion. These two approaches differ, but while some people may argue for one over the other (Alexander Stepanov, the inventor of the STL, is quite dismissive of inheritance*) the two can co-exist productively.

Notes

*

See STLport: An Interview with A. Stepanov <http://www.stlport.org/resources/StepanovUSA.html>.

References

ISO/IEC 14882

ISO (International Organization for Standardization), *ISO/IEC 14882:1998. Programming Languages—C++*, (Bound under the title *The C++ Standard*), International Organization for Standardization

STROUSTRUP

Bjarne Stroustrup, *The C++ Programming Language (3rd Edition)*, Addison-Wesley Pub Co, 2000

(See also, the special edition of this book).
