ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 9

August 1995

Editorial:
Sean A. Corfield
13 Derwent Close
Cove
Farnborough
Hants
GU14 0JT
sean@corf.demon.co.uk

Subscriptions:
Membership Secretary
c/o 11 Foxhill Road
Reading
Berks
RG1 5QS
pippa@octopull.demon.co.uk

£3.50

Contents

Editorial

I was planning to write about coding standards and coding style because that is a subject close to my heart, but I've heard so many people complain that coding standards restrict programmer creativity that I began to think about another, equally "religious" discussion that I found myself embroiled in recently.

I'm a great advocate of free speech and freedom of the individual. For some reason, this makes people think that I should be standing against government intervention in the battle over encryption technology. Those people say "the government mustn't be able to read our email and decode it" – the government say that unless they can decode email, criminals will be able to operate with total security. Those people say that even in times of war, the government didn't have the power to, effectively, "open our mail". Well, yes and no. Whose mail do they want to open? Yours? Why, what have you done to make them suspicious? The authorities need the ability to track and monitor criminals – in wartime, much effort was expended decoded the enemy's secret messages, but now we are (supposedly) at peace and the real enemy to society is crime.

Some criminals are very sophisticated and there is suggested legislation (in England at least) that is specifically targeted at certain criminal groups that would like to use electronic data communications as a secure way to operate their business. The Internet has made the possibility of secure, global crime networks a certainty. Unless, of course, governments are allowed to tap in and decode everything. That really bothers some people.

It may bother you; if it does, I expect you are also upset by the thought of carrying national identity cards? Presumably, your passport doesn't count, nor all the other paraphernalia in your wallet or purse and they certainly don't infringe your personal liberties or restrict your freedom of speech.

What about programming standards? Do they bother you, too? Do they restrict your freedom of speech in C++ and prevent you doing what you want? They don't bother me in the least – I have nothing to declare officer!

Sean A. Corfield
sean@corf.demon.co.uk

Software Development in C++

This section contains articles relating to software development in C++ in general terms: development tools, the software process and discussions about the good, the bad and the ugly in C++.

In this issue, Francis Glassborow asks whether it is time to stop pretending that C++ is just "C with knobs on", George Wendle makes **const** sound hard, Kevlin Henney takes a look at what the Standard Template Library might mean for the future and The Harpist contemplates some implementation quirks.

A Better C?

by Francis Glassborow

Eight or ten years ago the statement that C++ could be viewed as 'A Better C' was not unreasonable and for many tasks the use of a C++ compiler to compile C code cleaned up a number of problems. Understanding of the desirability of prototypes and the weaknesses in the preprocessor was important. This was quite independent of the provision of tools for data hiding, encapsulation of behaviour and inheritance.

There was so much good stuff in the early development of C++ that it rapidly escaped from its birth place. This happened not long after C had escaped from its cradle (mainly UNIX environments) and was being implemented by too many who did not properly understand it. Does that matter? Yes, because many who are involved in the development of C++ have less than a perfect grasp of C. Those with a sound grasp of C were pre-occupied with producing an ANSI standard (later to become an ISO one). During the process of that development (of a C Standard), those involved found many insights and a few surprises.

They were not trying to design a language but to standardise what already existed.

On the other side, the adventurous risk takers were designing a new language, C++. For a number of reasons it seemed desirable to bind this language tightly to C. Though C++ was not designed as an Object-Oriented language, many of its resources seemed suitable for use in an OO environment. Pressures arose to improve its support for this paradigm even though it was markedly different from the underlying expectations of C.

C was trying to provide maximum portability so that it could be used on as many hardware systems as possible. Its numerical representation is defined to be a binary one so anyone who has hardware that works in a non-decimal fashion will find implementing C difficult but that is about the only limitation.

The preprocessor was a powerful tool for supporting portable code and the elder statesmen of the language knew how to use it to best effect. Concepts of scope were developed to handle the increasing complexity of programs. Unfortunately the pre-processor was not designed for such complexity.

I don't think that any language designer actually sat down to eliminate the preprocessor. I think that one day several of them realised that various sensible items they had in C++ were making the preprocessor less necessary.

Let us look at one of these; **inline**. If **inline** had been designed to eliminate preprocessor macros it would have been more than a hint to the compiler, it would have been an instruction because that is the way preprocessor macros work. The advantage would have been to provide something that respected scope but there would have been no conflict with the concept of 'unique definition'.

In fact the idea of **inline** was more by way of providing support for access functions etc. The result was that as well as having inline code, programmers might also need a single addressable copy as well. When put like that, solutions present themselves but without explicit specification we finish with confusion such as **inline** functions being **static** by default.

*Francis means **static** in the sense of internal linkage, here, rather than one of **static**'s many other meanings – Ed.*

The process of "development by response to the current problem" is hacking. Everyone of us knows that proper development must start with a specification of the full problem that needs solving.

Another example of the same unfortunate thinking surfaces in the introduction of **const**. Certainly **const**, in the sense of read-only was a step forward to safer programming. **const** as a mechanism for providing manifest constants instead of the traditional C use of **#define** seems like a good idea until you start looking at the consequences. Fix the wasted space by letting compilers optimise away the storage if it is never used, but what if it isn't optimised away? What linkage should they have? Well to fix the linker problems, **const** globals will have to have internal linkage. You see the problem? Using **const** to provide manifest constants is a hack, the very fact that it nearly works, and often works to the satisfaction of the programmer does not stop it being a hack. We need both a read-only qualification of variables and a scope safe mechanism for manifest constants. My current advocacy of using enums for the latter is still a hack, less problematical until you need manifest constants of a specific type, but still a hack.

Values vs. objects

The C programmer who is seduced by the attractive hacks that C++ provides, the safer use of pointers, the pleasures of more intelligent i/o, and so on, is in serious trouble. The language that started out as a sensible development of C has moved radically towards OO support. No longer does assignment return a value, it returns a reference to an object (well it might not in our own user provided versions).

Support for OO suggest all kinds of modifications to the semantics of a language. At least different syntax stands out and ignorant use will often generate a diagnostic ("error" to you and me). Changes in semantics are much subtler and are very bad news. I think that C++ has moved so far down this path that it is doing no one any favours by continuing to talk about its use as 'A Better C'. Like evolution of species, there comes a time when something has evolved to the stage where it can no longer cross breed with the origi-

nal. I suggest that that point has arrived for C and C++. Inexperienced programmers who use C++ for C programming are laying up a wealth of problems for themselves.

The attempt to continue to support C directly in C++ is damaging C++. Many of the concepts (that were well formed and well understood in C, such as scope) have not made the transfer from C to C++ intact. As problems arise the language is being tweaked (hacked) to accommodate them.

C++ has a lot to offer but 'A Better C' it is not.

Francis Glassborow
francis@robinton.demon.co.uk

Quantum Chromo Typology
by George Wendle

Deep in the structure of modern physics lie some weird things called quarks out of which the World as we know it is supposedly constructed. What makes things particularly weird is that these fundamental building blocks don't just come in such kinds as up, down, top, bottom etc., but that each of these kinds come in both normal and anti-form and that each of those come in three flavours (deceptively named red, blue and green). That is all the sub-atomic physics you are going to get from me today.

Deep inside C++ lies a system of basic types out of which all other types are constructed. Before we look at those, let me spend a little time surveying the system as it exists in ancestral C.

Ancestral types

C provides us with a palette of built-in types coupled with rules for deriving types and producing compound types. There are some serious flaws with the C system that derive from its ancestry and minimalist approach. Probably the most outstanding of these is **char** which has a kind of schizophrenic existence. Sometimes it looks and behaves like a byte (I think that should be explicitly **unsigned char**). Sometimes it looks and behaves like storage for a character, that also should be **unsigned**. Finally it is often used as a minimalist integer where reason suggests it might be **signed** by default. C89 allows the implementor to determine the signedness of a **char**. It then proceeds to provide a library where most string functions have plain **char** or **char*** parameters. It finally shoots itself and its users by declaring that in some circumstances (e.g., *strcmp* and *strncmp*) the **char** parameters will be treated as **unsigned** regardless of the way the implementation views **char**. Note that very carefully...the prototype for *strcpy* has two **const char*** parameters but the actual data is compared as if it is **unsigned char**.

There is also a problem with *wchar_t* in C as it is defined via a **typedef** (or possibly a **#define**) to an integral type. This is simply not enough because as long as the type selected will represent the largest character set of any supported locale, it can be any of the builtin integer types. That means that the user does not even know enough of the interface. I am all for hiding implementation details but only if consistent, predictable behaviour is provided.

Because C is a value based language, the use of storage class specifiers such as **const**, **volatile**, **register** and **auto** may or may not provide derived types. I could spend a lot of time discussing this but in the final analysis it is largely a matter of viewpoint.

> ***const*** *and* ***volatile*** *are type-qualifiers in C, not storage class specifiers – Ed.*

Pointers are a different issue. Again, any top level qualification may or may not be a different type in C, but discussing it is a complete waste of effort. However, note that we have the idiosyncrasy that a top level pointer indistinguishably incorporates pointers to many different types – solo types and vectors of all possible lengths. If you have any doubt about this, consider two arrays, one of nine **int**s and one of ten **int**s. Both these arrays may be accessed via an **int***. On the other hand an array of five arrays of nine **int**s cannot be handled via a pointer to an array five arrays of ten **int**s. There are lots of subtle variations hidden behind the facade of pointers.

Of more direct importance are the differences between **int const***, **int***, **int volatile*** and **int const volatile***. Each is a subtly different type which manifests when you consider the rules for passing arguments of these types to parameters. Sure, you can pass the value of a **const int** to an unqualified **int** parameter but you cannot pass an **int const*** to a parameter of type **int***. The only way that this can be described is that the pointers are of different (incompatible) types.

My reason for taking a little time on the C type system is that it is poorly understood by most C

programmers but this poor understanding does not result in too much harm.

C++ typology

The picture changes dramatically when we move to C++. Quite apart from a much stronger type system, there are two vital extras. C++ supports user defined types, and even more significantly it supports two forms of overloading – operator and function.

C++ makes a serious effort to support programmers in designing their own first class types. It gets very close to empowering users to define types that are indistinguishable from the builtin ones. Its major failures are largely in the realm of sequence points and overloading some operators. You cannot provide the same behaviour for your versions of things such as logical-or because there is no mechanism in the language to specify delaying the evaluation of a parameter. C++ specifically prohibits your overloading the conditional operator. I think a good case could have been made for prohibiting overloading the other operators that include a sequence point.

Overloading functions provides another problem – it is the type system that is used to distinguish overloaded functions. This means that the type system is pushed to the forefront in C++ and demands that C++ programmers should at least have a good intuitive grasp of 'type'. In my experience this grasp is missing. The big irritant is what I have nicknamed 'Quantum Chromo Typology'. By this I mean the subtle flavours of types that C++ has produced in order to support overloading (maybe the flavours – colours – were already there, but in C++ it is essential that the programmer recognises them).

When programmers create their own new types by declaring classes, they actually create far more than just a single type. I'm not referring to the infinite regress that pointers generate but something else that includes a builtin set of conversion rules. Consider the following:

```
class T {
    int t;
public:
    T (int i=0) : t(i);
    operator int () { return t; }
}
```

How many types have been created? By my count, at least eight:

```
T
const T
volatile T
const volatile T
T&
const T&
volatile T&
const volatile T&
```

Now consider variables of these eight types and parameters of the same eight types. Which of the variables would be valid arguments for each of the parameters. If you think that is easy, add pointers into the mix and consider which pointers are compatible.

Now you have cleared up those, ask yourself which varieties are distinguishable for the purposes of overloading? Now go back to my minimalist class *T* and consider which of the eight types can be assigned to which others, and which can be created by copy construction from variables of the same or derived types?

Now, when you have got that clearly sorted in your mind consider the following:

```
template <class Q> void f ( ?? );
```

and replace the '??' by each of the variations of *Q*. Are all these variations legal? Of course they are because you should be able to write a generic version of any function. Actually you should be considering at least the eight varieties above and the eight pointers to such. Now for each of those 16 potential template functions which versions of *T* (or pointers to *T*) can be used to instantiate a function?

I make that 12, George: you can't have pointers to references...or are you including references to pointers? – Ed.

For example consider:

```
template <class Q> void f(Q&);
```

Is

```
T t;
T& tr=t;
f(tr);
```

valid code?

By the way what about the following code:

```
volatile T vt;
const T ct=vt;
```

In other words, can I use a copy constructor to create *ct* as a clone of *vt*? Is the answer the same for:

```
const T ct;
volatile T vt=ct;
```

Now consider the following:

```
class path {
    mutable int i;
public:
    path () : i(0) {}
    path (path&) : i(1) {}
    path (const path &) : i(2) {}
    path (const volatile &) : i(3) {}
    path (volatile &) : i(4) {}
    void print() { cout << i; i++; }
    void print() const { cout << i;
i*=3; }
    void print() const volatile
       { cout << i; i*=4; }
    void print() volatile
       { cout << i; i*=5; }
};
void fn(path & p) { p.print(); }
void fn(path const & p) { p.print(); }
void fn(path volatile & p) { p.print();
}
void fn(path const volatile & p)
       { p.print(); }
```

Now write some code that exercises all these functions. When you have your code working comment out any single function (either a member function of class *path* or one of the overloaded versions of *fn*). Predict what will happen to your test code. Now try commenting out a second function.

*That's a bit unfair George! How many of us have compilers that support **mutable**? – Ed.*

Conclusion

What I have attempted to do with this item is give you something to think about – what you might call the ultra-fine structure of the C++ type system. I have largely set you questions rather than attempting to give you answers because I believe that programmers need to experiment to develop a good intuitive grasp of what they are doing when they add qualification to a member function, to a parameter, to a type parameter in a template and so on.

Note that all flavours of a type necessarily share two things: constructors and destructors. All other behaviour can be made different for each variation of cv-qualification. However references are also sub-types (if you don't believe it, look back at the template cases).

Of course no sane programmer would intentionally mess around with creating radically different behaviour for types that are only distinct at the quantum chromo type level. But playing at this level might improve your sensitivity to C++ type problems just as the annual Obfuscated C contest has done much to improve C coding quality.

A challenge

Write a clear explanation of what I have called quantum chromo types (or elsewhere the ultra-fine structure of type).

George Wendle

*Poor old **volatile** always seems to be a second-class citizen when people talk about cv-qualification – everyone talks about **const** member functions but, as George shows, there are four flavours not just two. The committee are still wrestling with the semantics of volatile: they haven't even decided yet whether **T::T(volatile T&)** is a copy constructor or not! – Ed.*

Seduction: The Last? – Applying the STL mindset

by Kevlin Henney

There are two things that immediately strike you about the STL (Standard Template Library):

1. it won't compile, and
2. it's a very powerful way of thinking.

Developed by Alex Stepanov [1] and Meng Lee at Hewlett Packard, the STL is a library of generic components – algorithms, functors, object adaptors, and containers with their iterators – based on thorough operational specifications [2]. Andrew Koenig suggested that it should be put together as a proposal for the C++ standard library, and the rest is becoming history [3].

Library design philosophy

The STL makes heavy use of templates – in some places using features that have been recently standardised but are not yet supported by any compilers – and no use of polymorphism. The library's philosophy is to make algorithm use and design significantly easier. The non-inheritance approach comes as a surprise to many, but this should not be taken to mean that the library components are inflexible. On the contrary, components are heavily parameterised, the difference being that most of the parameterisation is at compile time. It is easy to build an efficient polymorphic container hierarchy using STL containers as the underlying implementation – and this is something I may return to in a future article – but not vice-versa. As such, the

STL constitutes the more fundamental approach to library components.

The emphasis on algorithms endows the STL with a particular flavour, once and for all giving C++ containers and iterators a style of their own. Of late, C++ libraries have been growing their own standard idioms and becoming less like Smalltalk hand-me-downs. The STL takes a new and alarmingly simple approach to complete the picture. Or is that old and alarmingly simple? Choosing the data structure to simplify the algorithm is hardly new advice, but it is this approach more than any other that typifies the library. An important feature of algorithms is their complexity, i.e., their relative performance in terms of the number of elements they operate on. The proposal gives the relative cost of each operational expression. This, as well as the notion of interface, is used to fully define what a type is; although the STL is not strictly object-oriented, it is firmly based on abstract data types.

Containers

The standard first tackles the specification of container interface and behaviour. A number of parameterising types are specified, such as the reference and pointer types used for the containee type, and then a number of operations that the container must support, e.g., default construction, copy construction, equality and size query. These operations are specified in terms of valid expressions along with their expected behaviour and complexity. For instance, the complexity of the equality operation is linear: the time taken to determine equality of two containers is no worse than proportional to the number of contained elements.

Sequences are a specialised form of container. These are required to satisfy the constraints placed on a container in addition to a number of others, such as insertion and erasing of elements. A number of optional operations are also specified:

- *front* and *back*, to query the first and last element;
- *push_front* and *push_back*, to prepend or append a new element;
- *pop_front* and *pop_back*, to drop the first or last element; and
- **operator[]** to access an indexed element.

The library provides three standard sequence classes:

- *vector*, the standard array class, which supports random access and length change at the end in constant time;
- *list*, the standard doubly linked class, which supports general insertion and erasing in constant time;
- *deque*, which supports random access and length change from either end in constant time.

The standard also specifies the requirements, in addition to those for containers, for associative containers. Standard implementations are provided for *set*, *multiset*, *map* and *multimap*. The *map* class is what is sometimes known as a dictionary or an associative array, and the *multi-* classes are bags.

Iterators

What gives the containers an extra dimension and their flexibility with algorithms is the specification of iterators. Iterators have a straightforward pointer-like interface, going against the trend for ever fancier and more all-knowing iterators. Many operations on containers are specified in terms of iterators rather than indices: *find* returns an iterator to the first occurrence of the element to be matched; *insert* and *erase* operate either single iterators or a range specified by two iterators. It is as if they were an abstraction of pointers into a container, in the same way that pointers can be used within a plain old C array.

A benefit of this is that algorithms can be written in a generic way on iterators. They do not tie you down to a particular implementation, e.g., you need not inherit from *VendorSpecificClass*, and they can be used with plain arrays. Both of these points relate to efficiency which, despite C++'s high level features, is still something that must be considered as part of its 'spirit'.

The standard specifies five categories of iterator, depending on the kinds of operation supported:

- *input*, these are quite simple iterators for single pass algorithms and support only equality, increment and dereferencing for reading;
- *output*, these are also single pass, like input iterators, except that only dereference for assignment is supported;

- *forward*, these have both input and output iterator properties, and are useful for multi-pass unidirectional algorithms;
- *bidirectional*, which as the name suggests supports all the features of a forward iterator and also the decrement operator;
- *random access*, this is a generalisation of bidirectional that supports full 'pointer' arithmetic.

General algorithms for sorting, reversal, filtering, etc. are written that require only iterators in their interface, without any explicit mention of a container.

So how are iterators initialised? Just as with pointers into arrays, iterators are valid on only one container at a time. The standard interface for containers provides for return of an iterator at the beginning and just past the end:

```
template<class value_type> // simplified
class list                 //
declaration
{
public:
    ...
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    ...
};
```

By "just past the end" I mean that such an iterator is not legally dereferenceable as part of the container, but is notionally just after the last legal element. This is used as follows:

```
list<int> l;
...
for(list<int>::iterator i = l.begin();
                        i != l.end();
                        ++i)
    *i += x;
```

The past-the-end marker is a useful out-of-band value for denoting conditions like a failed find. The other thing to notice about the declarations is that **const** and non-**const** iteration are treated as separate. A non-**const** iterator dereferences to a modifiable lvalue, or dummy lvalue, whereas a container cannot be modified through a **const** iterator. The use of **const** preserving iterators mirrors the usage for pointers and ensures complete type correctness across the container and its iterators.

Traversables

The requirements for the STL are biased towards containers. Well of course they are! But aren't we missing something? Yes. There is no reason that all iteratable entities need to be size constrained in any container-like way, so there is something more general than a container that could be defined.

The appropriate iterator category for this concept would be the input iterator. For obvious reasons I call this more general concept a traversable. This covers all containers and also includes algorithmic sequences such as random numbers, ranges, filters, arithmetic and geometric progressions, and almost any other read-only transient sequence that you might care to name. This is similar to the idea of generators [4]. However, a generator is an iterator that does not explicitly refer to a sequence; here I make a distinction between the iterator and the iterand, no matter how abstract or simple a sequence it encapsulates.

I am not suggesting that this idea needs to be expressed in the C++ standard, but you may find it a useful design tool. As an example, I believe that Sean's basic outline of a lexer [5] could be moved towards a traversable model with relatively little effort. On the next CVu disk, and then on ACCU's ftp site at Demon [6], you should find my implementation of a fully working non-template prototype of the random number example that I mentioned in passing [7].

On being lazy

A whole raft of lazy containers can be built on the STL foundation, possibly reusing some of the existing predefined classes. Generated sequences, as discussed above, are a form of lazily populated list. So long as you only access from the head forwards, i.e., via an input iterator, you need not be any the wiser that the collection is actually virtual (in the original sense of the word). Evaluation on demand can also be a feature of sized containers. For instance, sparse and growing arrays that automatically fake default values or create elements as needed, like **awk**'s associative arrays.

A sparse array can have a fixed size but with storage allocated only for members that do not contain the default value for the collection. Some neat tricks with proxy classes as reference types ensure that this illusion is smoothly maintained. The underlying implementation is free to use a map class or a sequence of sequences that

capitalises on contiguous groups of non-default elements.

Auto-resizeable arrays simply grow to meet their uppermost referenced member, e.g.

```
grow_vector<int> v;
cout << v.size() << endl; // prints 0
v[100] = 0;
cout << v.size() << endl; // prints 101
```

This class could be built from predefined components, such as a sequence, or sequence of *deque*s, or it might take advantage of a sparse array's properties to avoid unnecessary allocation as a trade off for access speed. I referred to such a type in [8], where the **const** version of **operator[]** would not change the length, throwing an exception instead.

Persisting through space and time

A persistence model might use the STL as its foundation. It has been suggested that this can be achieved by providing a specialised allocator for the container. A feature I have not mentioned so far is that the allocation strategy used by a container is fully parameterisable. The library classes use default template arguments to plug in the standard allocator, which effectively corresponds to the standard **new** and **delete**. At first, you do not seem to gain much, except when you realise that the iterator abstracts the whole process of allocation, what a reference is, what a pointer is, and how to convert a reference to a pointer. By providing your own allocator not only can you easily create your own allocation strategy for a library class, you can also abstract whether or not an object is actually live and present in memory at any point. In other words, the Holy Grail of OO: persistence.

An alternative approach is to provide a lazy container that allows swapping of objects to disk and relatively transparent retrieval but makes persistence an up-front issue and an explicit feature of the container. This would be simpler to implement than the allocator version, but would not necessarily provide the same transparency. Clearly the trade off between using an allocator, a specialised container, or a hybrid of the two must be evaluated, but this illustrates that there is more than one way to go.

By restricting part of a vector's interface, namely leaving out the size changing operations, it is possible to create a custom allocator that uses a given fixed part of memory directly. This could be used for mapping lower level structures from the operating system, C libraries, other languages, or other address spaces into a convenient object, e.g.,

```
vector<unsigned, direct_allocator>
            mapped(length, 0,

direct_allocator(base));
mapped[0] = a; // assigns from base for
               // sizeof(unsigned)
mapped[1] = b; // assigns from base +
               // sizeof(unsigned)
...
```

If the constructor to *direct_allocator* is a converting one, this could be simplified to

```
vector<unsigned, direct_allocator>
             mapped(length, 0, base);
```

In addition to shared memory, further adaptations of allocators suggest file mapped, pipe, or message buffer approaches, depending on what features a particular operating system offers. Some restraint and taste is probably required in this area – as noted by Arthur C Clarke, any form of technology that is sufficiently advanced is virtually indistinguishable from magic.

Wrapping up the file system

The old procedural way of thinking about APIs endures through familiarity, but such complacency can hide a better approach. Remember that once a clean abstraction has been made and committed to code, it need not be made again.

Directories can be viewed as containers of files. The Posix *opendir* and *closedir* functions, on *DIR* structure pointers, and the *dirent* structure, with the entry name *d_name*, already constitute an iteration model. Using this as a foundation it is possible to create a directory container class yielding iterators that dereference to pathnames. These can be viewed as truly single pass containers, so that the *DIR* pointer is owned and handled by the container. More practically they can be implemented as re-entrant objects, so that each iterator has its own managed *DIR* pointer.

A simple version of a directory listing command could be implemented as follows, using the standard copying algorithm with output iterators on *cout* and automatically inserting a newline after every write:

```
int main(int argc, char *argv[])
{
    directory dir(argc > 1
                  ? argv[1] : ".");
    copy(
        dir.begin(), dir.end(),
```

```
ostream_iterator<string>(cout,"\n")
    );
    return 0;
}
```

Should it be possible to erase entries from a directory container? In principle this is easy to provide, but there is a potential loss of symmetry in that directory entries may not be inserted or created in a symmetrical manner. Insertion using iterators from another directory could be interpreted as creation using hard or soft links, as appropriate. Encapsulating pathnames and file types, and hence their creation method, provides for single pathname insertion. A uniform interface for insertion is possible once clearly defined.

Pathname encapsulation is another interesting container candidate. Most systems have some kind of hierarchical or indexed file system, with pathnames reflecting such addressing. A fairly common requirement is to iterate through pathname components: a pathname iterator would automatically extract the path separator token on iteration. In other words,

```
pathname path = "/home/kevlin/bin";
for(pathname::const_iterator name =
                              path.begin();
    name != path.end();
    ++name)
{
    cout << *name << endl;
}
```

Would print out

```
home
kevlin
bin
```

Some additional handling for prefixes could give you relative versus absolute pathname queries, or extended pathname encapsulation, e.g., URLs.

Saving the environment

Operating system or application configuration files are another form of external container that can be plugged into this approach. Windows .INI files can be viewed as a two tier hierarchy of sections and entries accessible by iteration or key lookup. The features available in the Posix <*pwd.h*> header – *getpwnam* and *getpwuid* functions, and **struct** *passwd* – provide the basis for an associative view container.

Should such containers be what are termed singletons [9], i.e., only a single instance can exist per application? As their state is actually held and managed externally the container objects can be considered proxies of this state, and so there is typically no need to complicate the model with a singleton approach.

What about environment variables? There are two kinds of environment we are interested in: the actual current environment, as accessed by *getenv* (ISO C), *environ* (Posix) and *putenv* (common extension); a composed environment for use in executing other processes, as used by the Posix *execle* and *execve* calls.

One solution is to have a singleton instance representing the actual environment, but this necessitates one class for the current environment and another for composed environments as the mechanisms are so different. An alternative is to treat the current environment as an implicit global resource behind the scenes that can be read from or written to, suggesting *refresh* and *apply* members for environment containers.

Conclusion

Looking at some of the ideas above, some other writings and the specifications for a couple of commercial STL libraries – those from ObjectSpace and Modena to be precise – there seems to be a strong convergence in thinking. A lot of other library areas seem to be undergoing the STL treatment independently and in parallel by vendors and individuals alike.

Finally, the seduction I am referring to in the title is in the way of thinking, particularly as regards iterators. It is a powerful tool that, in conjunction with other powerful software engineering concepts you may have accumulated, gives you another solid design framework within which to work. The emphasis here is on the multiplicity of compatible techniques you may use: it would be foolish to think that any of them is the only or the last one.

Kevlin Henney
kevlin@wslint.demon.co.uk

Notes and references

[1] Alex Stepanov is interviewed by Al Stevens in the March 1995 issue of Dr Dobb's Journal

[2] The original definition and implementation of the STL is available from `ftp://butler.hpl.hp.com/stl`

[3] The STL has been incorporated into the draft C++ standard, a copy of

which is available from ftp://research.att.com/dist/c++std/WP

[4] "A Little Smalltalk" by Timothy Budd covers generators in some detail

[5] "So you want to be a cOOmpiler writer? – Part II" by Sean Corfield appeared in Overload 8

[6] ftp://ftp.demon.co.uk/pub/ACCU

[7] The random number example is to be used (hopefully) as the basis for a future magazine article

[8] "Overloading on const and other stories" appeared in Overload 7

[9] "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma, Helm, Johnson & Vlissides (aka the 'Gang of Four'), is a cornucopia that includes the Singleton pattern

Joy Unconfined – reflections on three issues

by The Harpist

I have known Francis for more than half my life because he was the one that corrupted my mind by introducing me to the gentle art of computer programming. He has never been one for staying in the safe central territory of any activity. I have seen code of his get up to just about every trick in the book but almost always with that extra fingerhold on safety. His implementation of Forth on a ZX-Spectrum included much use of self-modifying code, but to the best of my knowledge each instance made no assumptions as to the code's prior state. He always taught us to do anything as long as:

1) we could guarantee that it would work
2) we documented it
3) we were willing to maintain it
4) we were able to justify it as being an effective solution to a problem.

Now look back at his item on 'Polymorphic Objects' in the last issue. It fails criterion one, and does so in a very nasty way that seems to have been missed even by our esteemed editor.

I did say that I had no idea whether it worked or not! – Ed.

Consider the following program based on his code (assume that *xstretch***()** is a polymorphic function that stretches an ellipse (circle) in the *x* dimension, i.e., it works simply with an ellipse but will have to call *change***()** for a circle):

```
int main() {
       Circle c;
       c.xstretch(2);
       (typeid(c)==typeid(Ellipse) ?
              cout << "I've changed" :
              cout << "I'm still a
circle";
       cout << "." < <endl;
       return 0;
}
```

What do you think running this program will display? Think very long and hard. Certainly *c* has polymorphed into an ellipse but how does the compiler know that? Compile the above program with no optimisations and you should get what Francis expected. However, note that there is absolutely no reason for the compiler to expect any form of polymorphic behaviour: *c* is neither a pointer nor a reference and even if I changed the declaration to *Circle***&** *c* = ***new** *Circle***;**, there is still no way that the compiler should expect polymorphic behaviour.

If the above program is to run the way Francis expected, then we would have to cripple optimisers quite unnecessarily so that they were forced to call virtual functions through a virtual function table (or other device for implementing polymorphism) even if the compiler believed it could statically identify the required function. This is not acceptable – indeed we should be doing just the opposite: we should be encouraging implementors to provide static binding of virtual functions whenever it is possible. I believe that all objects should be statically bound to their virtual member functions, in addition references and pointers should also be so bound whenever the compiler can determine that the static and dynamic types must be the same (and possibly at times where it can determine the dynamic type statically even if that is not the static type).

Sorry, Francis, polymorphic objects are another of those seductive ideas that lead to either fatally flawed code or a permanently crippled language.

(To be fair, Francis had actually worked out most of this for himself, before I discussed it with him, but it is nice to be able to correct one's teacher sometimes)

Virtually inline

I cannot remember where I saw this, but I recently read a comment about Symantec's new compiler (well worth a look if your hardware can cope with it) rejecting definitions of virtual functions in the interface of a class (horrible thing to do, defining functions in an interface that is). The grounds being that inline has to be acted on at compile time whereas virtual is the exact reverse, the code has to be selected at execution time. On the surface that appears to be a justification for not accepting virtual inline functions. That is too superficial, and I think that any compiler that rejects such code needs amending (I don't think I would go so far as to claim it is a bug but it is getting pretty close). Strictly speaking **inline** is only a hint to the compiler just like **register** is in C. The compiler can inline code without your suggesting it and it can decide not to inline code that you have marked as such.

On the other hand, as intimated above, just because a function is virtual does not mean that it must be bound dynamically (at execution time).

While it is hard to imagine any circumstance where a compiler could both inline code and use dynamic binding simultaneously, it is certainly desirable for a programmer to indicate that the compiler can inline the code if late binding proves unnecessary.

As I wrote the above I was thinking about the nature of inline functions and remembered that they are currently static functions (i.e., have only file scope visibility) by default. I read somewhere that X3J16 was intending to deprecate the use of **static** at file scope. What are they going to do about implicit uses? Such use also applies to file scope **const** variables.

*The ISO C++ committee have deprecated file scope static because unnamed namepaces provide a 'better' alternative. To be precise, file scope **inline** functions have internal linkage rather than being **static** – the same applies to file scope **const** variables – Ed.*

Following this flow of thought, many programmers are coming to realise that the idiom of using a **const** variable where C traditionally uses a **#define** to provide a manifest constant is flawed. It doesn't work properly inside a class, though this is the primary reason for introducing it in the first place. It has to be defined out of class which is a pain and causes problems when you need the value in class at compile time. For example:

```
class T {
      const int size;
      int array[size];
// etc.
};
```

doesn't work. This has led to the idiom of using enums for such purposes so we have:

```
class T {
      enum{size=100};
      int array[size];
// etc.
};
```

Which is all right as far as it goes, but we do not have typed enums (enums are of course types but we have no control over their underlying storage and conversion properties) and anyway that is not what enums were intended for. What we need is some new form of storage class specifier that simply instructs the compiler to use the value but not provide storage for it so we could write something such as:

```
class T {
      nostore int size=100;
      int array[size];
// etc.
};
```

I guess that isn't the best choice of keyword, but the idea is so simple that, even at this late stage, it could be added to C++ (actually it would be even nicer to add it to C). The problem is that it is so simple, so easy to fix, and so easy to understand that, inevitably, it would result in hours of discussion (lots of people understand it so they can express an opinion – its only seemingly useful things that no one understands that result in no discussion).

Sorry, Harpist, perhaps you'd better go back and read The Casting Vote more closely – the committee already fixed this to allow:

```
class T {
      static const int size =
100;
      int array[size];
// etc.
};
```

Admittedly, you are still required to have a static member definition somewhere but it solves the problem without introducing new keywords in an intuitive manner. And it didn't take hours of discussion, either – Ed.

Namespaces

I tried to understand these by reading the relevant part of the Committee Draft standard. I also found a copy of Metaware High C/C++ available at work which claimed to implement namespaces. I cannot find any other implementation. That leaves me with a problem. I find the text of the CD almost impenetrable and after much struggle I am beginning to suspect two things. First, I do not think that the Metaware namespace matches the one being described in the CD. I am not sure about this but that is my best guess. Second, I understood that one purpose of namespace was to support programmers synthesising their own program namespace from several external namespaces. I am absolutely sure that this was offered as a major facility that namespace would support. Again I could be wrong, but what is in the CD does not seem to offer this facility.

You're right: Metaware implements something different to the draft and you couldn't synthesise namespaces. The latter is now fixed – see The Casting Vote in this issue – but, unfortunately, the committee can do little about Metaware's implementation! – Ed.

When I find text as incomprehensible as that about namespaces in the CD and cannot find any body of experience on which it is based, nor any carefully worded specification of the problem it is intended to tackle, I become deeply suspicious. Who understands this? Did those who voted for it know what they were voting for?

Not entirely, judging from the recent discussion on the committee reflector about namespaces – Ed.

I do not need an implementation of **nostore** to understand what it does and to be certain that it will work consistently: the only cost is a little work to the grammar of the language and a few (probably very few) adjustments to the text of the relevant standards documents. But when it comes to complex proposals such as namespaces I think that nothing less than three working implementations should exist before the proposal goes any further.

The thing that is giving me increasing concern is that X3J16 seems to be pushing hard to get the whole of C++ standardised while there is no compiler in existence (well, publicly available) that supports exception handling, templates and namespaces as described in the CD. Without such, we are all in a position of abstract design. As programmers we all know just how much lies between abstract design and concrete implementation. By the time C++ reaches a standard it is too late – much too late. Version 2 of C++ will not arrive until at least ten years after version 1. By then so much code will have been written that has to tackle any language flaws that nothing less than an entire new language will fix the problem.

Finally

As I read the article from Kevlin Henney in *Overload 8* a thought crossed my mind (they do sometimes) – how do we get commercial library producers to specify their products properly. If I understand Kevlin correctly, he maintains that there is no point in designing a class hierarchy of *Shape* until you know for what purpose it is being designed. I agree, but the implication is that reusable code needs documentation that fully describes its design criteria.

Could we start with MFC? Microsoft have a very specific view of the computer world and what are desirable programs (they run under MSWindows and are written by themselves ;-). Seriously, a library written to support large scale data processing for the Insurance industry will probably be inappropriate for a developer of a stock control application for a small business.

The Harpist

The Draft International C++ Standard

This section contains articles that relate specifically to the standardisation of C++. If you have a proposal or criticism that you would like to air publicly, this is where to send it!

Two different views of the most recent joint standards meeting are given by Francis and myself, and Kevlin takes some pot-shots at some of the inconsistencies in the draft standard library.

Diary of an Observer

by Francis Glassborow

Elsewhere in this issue you will find Sean's report on the technical side of the recent joint meeting of WG21/X3J16. This item is intended to remind readers that Standards, even International ones, are written by people who are not very different from yourselves.

For those that are unfamiliar with the technicalities, X3J16 do not allow a member organisation to vote until they attend their second meeting and they have to attend two out of three meetings to retain voting rights – otherwise they are an observer. This is an excellent rule as it means that those who vote must have more than a passing interest and at least a minimal amount of background. I was attending on behalf of Richfords (who are a London based organisation that, among other things, provides training in C++). I was also part of the BSI's nominated delegation so could also describe myself by the more grandiose title of 'Technical Expert'.

That is enough pre-amble.

The start

We (that is Sean Corfield, Steve Rumsby and I) were due to fly out of Birmingham International at 10.15 am on Saturday, 8th July. Not a problem, you might think. Well it is if your house is effectively roofless (and so needs occupation if at all possible), your wife is in Germany and you don't own a car (I don't drive, so there would be no point in borrowing my wife's). International flights require you check in two hours before scheduled departure time and British Rail seem incapable of running trains to Birmingham International before about 8.15 in the morning – too tight a schedule to risk. My Bridge partner rescued me by driving me there, something much beyond the call of duty.

The flight to Chicago was uneventful though it is hard to classify the reason for the visit for the US immigration control who only know of 'Business' and 'Leisure'. They seemed to swallow hard at the concept of a 'business visit' for which I was not earning anything.

Chicago to San Francisco is further than you might think (unless your geographical knowledge is above average) and we were on another American Airline's Boeing 767 (do you know why seats C, E, G are adjacent?). The final leg of the journey was on a Jetstream 32. Not a plane for the nervous, or first time traveller – small enough to be piloted in a more aggressive fashion and with minimal space (carry-on luggage gets stowed in a pod under the fuselage). While waiting for our departure we collected Beman Dawes, another member of X3J16. We also found ourselves in conversation with another of our fellow passengers. When I expressed some surprise that the airline wanted some proof of identity for a purely internal flight they took great delight in telling us about the latest threat from the 'Uni-bomber' but that is another story.

We arrive

A short taxi journey brought us to the hotel and Beman paid my share as I knew I had nothing smaller than a $20 note. I was shortly to discover that I actually had nothing as my dollars were safely sitting on my kitchen table some 6000 miles away. Having to start on one's contingency fund of travellers cheques on day one is a bit cramping on one's style though probably a good way of minimising expenses.

We had twenty-four hours to socialise, play the tourist etc. before meetings started in earnest. Those who know me will not be surprised that I spent some of that time browsing through the bookshop opposite the hotel. Yes I did buy a couple of books, but not about any aspect of computing.

WG21 meets

Even though the two committees meet in joint session for technical discussion and decisions, WG21 still has some political decisions that are handled at a meeting that starts at 6 pm Sunday. The two main items this time were a sensitive issue of why there had been a two week delay in providing a distributable copy of the working paper for the CD ballot (it wasn't until the following Friday that the WG21 heads of delegation let the Convenor off the hook on that one). The other item was the anticipated votes on the CD. Of those present only the UK was firmly committed to voting 'No'. However all knew that at least four others (France, Netherlands, Australia and New Zealand) were very unlikely to vote 'Yes'. As it was very probable that this would result in SC22 requiring a second CD ballot we discussed the future timetable. It looks as if the most realistic/optimistic timetable will

produce a Draft International Standard about when the committees next visit the UK (July 1997)

Down to business

The next five days were to start at 8 am with breakfast provided by our hosts (endless supplies of orange juice, coffee, pastries and breads).. The joint committees met on Monday at 8.30 to organise the week and tackle the first round of administration. Looking round the room I was struck by one of the changes since I last attended a WG21/X3J16 meeting (London 1992) – the majority of those present had replaced the stacks of paper with some form of portable or laptop. Steve Rumsby's Psion caused something of a stir, particularly when he assured everyone that he had the whole of the working paper on it, and yes it could talk to other machines – more of that later.

By mid-morning we were ready to break into the groups where the real work gets done. This time it was three core groups and five library groups. That about says it all. No more extensions (though some clean-up work is still going on in Core III) and Environment, C compatibility etc. were all demoted to the status of "we'll meet ad hoc if we need to."

I joined Core I, or was it Core II (numbers don't really matter) with Josée Lajoie in charge. Josée is Canada's regular head of delegation, an employee of IBM working out of their Toronto Labs and a French Canadian. She is one of those quietly impressive people, a smile is rarely far away and she has immense tolerance for the misunderstandings of others. How many of the English speakers among you could comfortably handle meetings and technical issues in French? Until seven years ago Josée's English was no more than that which she had to learn as a second language. The priority issue for our group was to write a formal and acceptable description of the 'One Definition Rule'. I am not going into that here as that is Sean's domain. However it took all of Monday and part of Tuesday despite the excellent preparatory work by Jerry Schwarz.

...and John Max Skaller – Ed.

The rest of Tuesday was spent on easier, more tractable minor points. We still had to shelve a ream of work for next time. Tuesday evening was drafting time to try to get sensible words agreed upon (actually this is a never ending process). I managed to get lost and missed it.

Actually, Tuesday evening is when the WGs try to draft their formal proposals, Wednesday evening is when the drafting committee meet to draft the formal motions. The difference is subtle enough to be unimportant for anyone except members of the drafting committee – Ed.

Wednesday and Thursday morning was spent in full session while each group reported back and we tried to decide what we would actually formally vote on. This may seem like duplicated effort but it isn't. Sometimes a group comes up with a bright idea that has hidden implications. We don't try to fix such problems in real time but we do have to decide that we have a problem.

I was eating my lunches in the hotel as that allowed me to put it on the bill that was already on Richfords account. Various other delegates ate there as well. We'd had little difficulty with getting separate checks until Wednesday when one of the staff informed us that there was no way that he could keep more than two checks open on a table. He eventually recanted when we demonstrated that he had an alternative – we would all sit at different tables (thereby filling all of them), be served and then move to where we wanted to be (sowing even more confusion). The English don't confine their stirring to standards issues!

A round of applause

Our proposed solution to the ODR met with unanimous approval in the straw vote, something so rare on a major issue that it gained a spontaneous round of applause. (For the record it received similar treatment in the formal votes on Friday). There is still some polishing and I guess someone is going to come up with a corner case we hadn't thought of but I think it is now essentially complete.

A Caribbean meal

On Thursday evening Sean and I were entertained by Reg Charney and his wife to an excellent dinner at a Caribbean restaurant in the next town round the headland. Reg is one of US members and a staunch supporter of ACCU. He is also an example of the kind of committee member that many of you do not expect. He is one half of a partnership working in computing

but not as a C++ specialist. All told about one third of the active membership of WG21/X3J16 are either individuals or represent small businesses (with less than a dozen employees).

It was a fine evening, with pleasant conversation, enjoyable food and finished off with a quiet stroll by the beach. Then it was back to work for Sean and I as we discussed the UK votes with Steve.

I can highly recommend said restaurant: El Cocodrilla's in Pacific Grove. They do great alligator tails... – Ed.

While there were few surprises at the final full session on Friday when we voted on over thirty motions there was still more to come. After the end of the WG21/X3J16 meeting the US TAG (X3J16 members representing companies domiciled in the US) had to decide their vote for the CD. A routine matter, you might think. Nearly three hours later they had to resort to preparing for a letter ballot because they had lost their quorum. I guess that the US will vote 'yes' (and I am not sure that it isn't in the interests of the future of C++ that they do so) but it is worth noting that, contrary to some opinions, the issues is not entirely cut and dried.

Editing

As soon as the US TAG was over it was down to work for those of us who were still around. The results of the motions had to be incorporated into the WP. Some motions are very casual, requiring such things as 'include wording to the effect'. This means that the editor (Andy Koenig) is responsible for getting it right. At other times exact wording has been provided, but it is wrong. Such circumstances require what the Americans call wordsmithing. Then there is the general effort to improve the WP by wordsmithing to provide more accurate expression of what we mean (so as to reduce the need to say after the WP becomes an IS 'a close and careful reading of the Standard reveals that ...' i.e., we meant '...' but didn't say it).

Some wordsmithing is just tedious, some is hard work. I spent at least two hours trying to get the paragraph on qualified name lookup in a namespace to say what was meant. The original from Bjarne Stroustrup was fine as an informal statement but would have provided the language lawyers with a field day. Throughout Friday evening and all day Saturday a small band slaved away to get as much done as possible so that Andy would have a fighting chance of doing some of his employer's work over the next three months. It all has to be done using the arcane magic of **troff** and careful collation of work back into the master document. I wish I had had a camera to record the variety of equipment pressed into service. At one extreme we had Steve's Psion and at the other we had a fairly old Sun SPARC station (with a non-functioning floppy disk drive) – both these machines had to go through Sean's PowerBook so that material could be transferred via floppy to and from Andy's laptop. Hardware experts may realise that some pretty clever things were happening. We had our moment of panic when someone's machine reported detecting a virus.

Proving once again that an Apple Mac is a truly 'open' system! And I couldn't catch the virus which led to everyone getting me to format PC disks and transfer files to and from the infected machine! – Ed.

The return

The three of us departed for Monterey airport at 7 am on Sunday to find it fog-bound. American Eagle couldn't get a plane in to take us to San Francisco so at 8.30 they dispatched us by taxi on a 120 mile journey to catch our 10.38 flight from San Francisco to Chicago. I think that it would be only possible to do it on Sunday with a taxi driver who completely ignored the US speed limit of 55 mph. Fortunately the US doesn't have any problem with checking people's baggage in twenty minutes, even if it is being checked through to the UK. Sean's experience with international travel helped – I would never have considered checking baggage at the 1st class desk with an ordinary coach class ticket.

And finally

It somehow reflects on British Rail that the train I caught from Birmingham International to Oxford was twenty-five minutes late. The true significance of this is that it was a connecting train to Gatwick. I hope no one on it had a plane to catch.

To summarise, a hard but instructive week and made pleasant by both the quality of the company and sense of purpose and friendship. I have said it before but it is still worth repeating – standards work is a very effective way of getting to understand the language better.

Francis Glassborow
francis@robinton.demon.co.uk

The Casting Vote

by Sean A. Corfield

I'm writing this on the flight from San Francisco to Chicago after the most recent C++ meeting in Monterey, CA. It's been an eventful week – see Francis' *Diary of an Observer* – and many problems with the draft have been resolved. In *Overload 8*, I indicated that the ANSI public review had begun and that other countries would also be soliciting comments. Some of those comments were available in Monterey, but I think many more are yet to come: the public review period has been extended due to an unexpected one month slip in the ballot process. This will give people more time to read the draft and comment on it – keep those comments coming in!

Two years and counting...

We've just about reached the point now where 'all' we have left to do is resolve the 'small' issues that keep cropping up. There are no major extensions on the table, no major library additions planned and no major language changes predicted. After a period of rapid and widespread change, the draft standard is finally stabilising. Whilst that may give C++ programmers (and their managers) cause for rejoicing, it doesn't mean the committee's work is nearly done! The flow of small issues means that it will probably take us until 1997 to arrive at a draft standard that is precise enough to submit as a Draft International Standard (see previous *Casting Vote* columns for details of the ballot process).

That means that the UK meeting in July '97 may well be the one at which we know whether or not we will be on the brink of an official ISO C++ Standard.

Monterey was the first meeting since the Extensions WG was disbanded. Some of the former EWG members joined the Library WG (including Bjarne Stroustrup) and the rest joined the pool of Core WGs. I spent Monterey with Core III which looked at templates, exceptions and namespaces – Core III is the "not-the-Extensions WG" – but probably the most important step forward was taken by Core I at this meeting.

Just one definition!

The biggest definitional hole in the draft has now been filled: the committee adopted wording that specifies what has become known as the One Definition Rule. The essence of this rule is that it is OK to have two definitions of something in different translation units if those definitions are 'the same'. For the purposes of the ODR, 'the same' means the token sequence is the same and the name binding of those tokens is the same in each translation unit. Whilst most of the effects of the ODR are 'obvious' and common sense, there are a couple of 'gotchas'. My understanding is that an **inline** member function that calls a (**static**) **inline** function will violate the ODR if defined in more than one translation unit:

```
// file.h
inline int max(int a, int b)
{
       return a > b ? a : b;
}
class A
{
public:
       // ...
       int biggest() const
       { return max(x, y); }
private:
       int    x, y;
};
```

The member function *big***()** has external linkage (because it is a member function) but it calls *max***()** which has internal linkage and is therefore considered 'different' in each translation unit that includes *file.h*. I may be mistaken – I am writing this after hearing the discussion of the ODR proposal but before seeing the actual wording in the working paper.

To specialise or not to specialise

The closest thing to an extension that was added in Monterey was a clarification of the syntax for declaring and defining specialisations of templates. Now that partial specialisations have been adopted (see *The Casting Vote* in *Overload 7*), full specialisations were the 'odd one out' in the template world because they didn't start with the keyword **template**. In addition, **static** data members could only be specialised as definitions because the syntax did not allow you to distinguish between specialised declarations and definitions. This has been addressed by requiring specialisations to be declared (and defined) with the prefix **template<>**.

```
template<class T, class U> class A;
```

```
      // primary template
template<class V> class A<V*,int>;
      // partial specialisation of A
template<> class A<void*,int>;
      // full specialisation of A
A<int*,int*>* app;
      // use primary template:
      // T==int*, U==int*
A<int*,int>*  api;
      // use partial specialisation:
      // V==int*
A<void*,int>* avi;
      // use full specialisation
```

If you don't like this, blame me because it was my proposal and I've been lobbying for it for quite some time!

Are you pointing at me?

One of the template classes in the draft standard library which has attracted quite a few comments is the *auto_ptr* class, which allows you to wrap pointers so that they become exception-safe (or, at least, exception-saf*er*). One of the members of *auto_ptr* is **operator–>** and I have had some mail from people who've tried this class and found it doesn't compile – see *Q&A* in this issue. The committee previously decided that the return type of **operator–>** should not be checked inside the declaration of a template so you could have *auto_ptr*<**int**> and not get a compile-time error for **int* operator–>** unless you tried to use it. I proposed that this relaxation be extended, because it is perfectly reasonable to call the operator explicitly as a function:

```
X x;
T* p = x.operator->();
```

This is valid even if *T* has no members. It's valid because you are not trying to dereference the type returned by **operator–>**. The committee accepted my proposal and two paragraphs of the draft standard were removed as a result – definitely a step in the right direction!

Related to this, and part of the above proposal, the standard iterators in the draft library are now required to support *i->m* if it makes sense to do so. That will hopefully tidy up a lot of code that currently has to use **(*****i*****).***m* instead. I ended up editing the changes into the appropriate library clause and it made me realise just how much attention that section of the draft still needs: we're getting a lot of comments about the language clauses but it would be really helpful if you all tried to read the library and comment on that!

Except for destruction...

Over the last few meetings, the committee fixed a lot of the holes concerning exception-safety, by adding **try**/**catch** blocks around *mem-initializers* (well, around whole function bodies, in fact), providing *auto_ptr* and tightening up the rules about *exception-specifications*. This still left one particularly thorny problem: when an exception is thrown, the stack unwinds and destructors are called – if one of those destructors throws an exception, the program terminates (it actually calls *terminate***()** which can be overridden). Quite a few people have called for some mechanism that allows a destructor to ask "can I throw an exception?" A proposal from Germany provided the solution: add a function, called *uncaught_exception***()**, that returns **true** if an exception has been thrown but not yet caught (i.e., during stack unwinding). This provides the bare minimum necessary for robust handling of exceptions during destruction.

Synthesis

Core III also tidied up an important flaw in the semantics of namespaces. One of the benefits claimed for namespaces was that you could synthesise a new namespace from several others:

```
// standard namespace to be used by all
// programs written within ACME
namespace ACME {
  // open the standard library
namespace:
  using namespace std;
  // open Rogue Wave's library
namespace:
  using namespace RogueWave;
  // open version 3 of ACME's 'K'
library:
  using namespace KLibV3;
}
```

The intent was that ACME's programs could then include the appropriate headers and just say:

```
using namespace ACME;
```

This worked, but there are times when you don't want to open the whole namespace, you only want to pull parts of it out without getting (potentially) everything. The obvious way to do that is with an explicitly qualified name without worrying which namespace the declaration really inhabits:

```
ACME::initialiseKLib();
ACME::list<ACME::widget> widgets;
```

Unfortunately, this didn't work! Bjarne Stroustrup proposed a change to allow qualified

name lookup to 'tunnel' through *using-directive*s which fixes this problem. The committee accepted the proposal so namespaces now fulfil their initial promises. It's taken a long time to get clarification on the meaning of namespaces and, even now, a lot of people still don't really understand how they work. At least we now have a stable base, that seems to work, on which to build.

Small is beautiful?

There were a large number of 'small' issues resolved in Monterey. Each WG has a list of outstanding issues for the clauses for which they are responsible. Those lists typically have fifty to a hundred issues active with WG members working hard to suggest resolutions, draft WP changes and get the committee to accept them. This process is generally fairly successful and will be the pattern of work for the committee for the next few years. Not all the resolutions are entirely sensible and here are two from Monterey that I think are somewhat dubious:

Boolean arithmetic?

Assignment operators now allow the left hand operand to be **bool** which allows you to write:

```
bool b = true;
b *= 42;
```

This wasn't universally popular with the committee with a quarter voting against, but it falls naturally out of the existing rules for **bool**, unfortunately, because those weren't strict enough in the first place.

First class rights for unions!

A union can no longer have members with reference type. It was argued that you can't do much with such things so we should ban them. This motion was particularly unfortunate, in that we have already voted on it and defeated it. It succeeded this time because two of the National Bodies that strongly objected were not represented at Monterey.

I should point out that the remaining small issue resolutions were reasonable and included such things as:

- sequence points in mem-initialisers – to ensure that the initialisers are evaluated in strict sequence,
- multiple **extern** "C" definitions are now ill-formed,
- clarification of many issues regarding linkage, templates and the library's handling of exceptions.

Name injection revisited

In *Overload 7* I hinted that the committee were trying to restrict name injection to make it less surprising. In fact, at Monterey, there was a groundswell of support for removing the feature altogether but a couple of things stopped us. Consider the following code based on an example in Barton & Nackman:

```
template<typename T>
struct Comparable
{
friend bool operator==(const T&, const
T&);
};
template<typename U>
struct Array
: struct Comparable< Array<T> >
{ ... };
```

Or consider this, simpler, example:

```
template<typename T>
class basic_complex
{
friend basic_complex<T>
       operator+(const
basic_complex<T>&,
              const basic_complex<T>&);
// ...
};
basic_complex<double> z = 1.0;
z = 2 + z;
```

The last line requires a conversion on the left hand side of the + which means that **operator+** must be a non-member function. It also means that **operator+** cannot be a global template operator because conversions are not allowed there either (because of type deduction). So we must use a non-member, non-template operator which can be declared as needed for any sort of *basic_complex*. Only name injection allows us to do this.

At the moment then, it seems that name injection must live on. Steve Rumsby (maintainer of the UK C++ information web site) suggested three rules that might make name injection better behaved:

1. inject into the namespace of the template definition, not the namespace of the use,
2. defer injection to the end of a full expression (i.e., where temporaries are destroyed),
3. if name injection occurs, reconsider the expression and if any names have changed

their meaning, the expression has undefined behaviour.

These suggested rules are currently being discussed by the committee so we shall probably see a proposal on paper for voting at Tokyo. Incidentally, rule 1 only works since we changed the operator lookup rules in Austin.

What about that Barton & Nackman code? It factors out the name injection into a template base class so that any other class can be 'Comparable' – i.e., have an appropriate non-member, non-template equality operator – simply by deriving from *Comparable*. Neat? Clever? Obscure? I think we can expect to see much more of this sort of thing as programmers become more comfortable with OO design and flexible ways of using templates and inheritance – Barton & Nackman makes a good read on those grounds.

The future

For the committee, the future holds several more meetings at which we will continue to deal with small issues. For the C++ community, the future should hold an increasingly stable draft standard and compilers that conform more closely. Remember: two years and counting!

Sean A. Corfield
sean@corf.demon.co.uk

Uncontained – oddities and oversights in the standard library

by Kevlin Henney

The STL (see *Seduction: The Last?* elsewhere in this issue) is now part of the draft standard library, but how much of the rest of the standard library could be considered a part of the STL? Unfortunately not as much as might or should be. Whilst the *basic_string* template class has certainly been moved towards the STL model, other areas of the library remain sadly unaffected.

It is worth recognising that the library working group has finite resources; it is unreasonable to expect the whole of the library's style to change at the flick of a switch. However, most of the library has been invented by the committee – raising questions from critics about its maturity – and it seems surprising that consistency among the invented components may also become an issue.

Valerie and friends

The maths library includes containers such as *valarray* that cater for a more numeric view of computation. Although a more recent invention than the original *string* class, the nomenclature and style of these classes has not been made STL-like. It is trivial to show that vector computation classes can satisfy the standard container requirements. It would certainly simplify a developer's understanding of the library if a – for want of a better word – 'standard' approach were taken. Thus we might consider that in its current state the library is not wholly compatible with itself.

For instance, rather than *length* I would have expected the *valarray* class to have *size* and *empty* members for querying capacity.[1] There are also no iterator functions or types defined for it; a convenience that would allow easier integration with the algorithms library. Simply because FORTRAN fails to provide useful non-numeric operations on its arrays does not mean that a newly designed C++ library has to repeat its mistakes.

Sure, the results and operations I have described as missing can be deduced or handled by different means, but that's not the point of a standard library – I expect standard interfaces. The *valarray* class is not badly designed: it just doesn't fit.

Bits in pieces

The *bitstring* class, discussed in some detail in [1], was retired in preference to the **bool** specialisation of the STL *vector* class, the unspecialised version of which also saw off the *dynarray* and *ptrdynarray* classes. I have no problem with this except it appears that many of the bit specific operations present in *bitstring*, such as left and right shift, did not turn up in *vector*<**bool**>. Whether deliberate or by oversight, this does not seem an entirely fair exchange.

With the loss of *bitstring* the *bits* class, again discussed in [1], gained a couple of characters to become the *bitset* class. It acquired some STL

[1] Not to be confused with capacity which is a member of some containers already – Ed.

wisdom in the naming and rearrangement of some of its members, but lost out on having any of the iterators which its bit-unwise relative, *vector*<**bool**>, did rather well out of. In case you have any doubts: yes, it is possible to have references into and iterators over a bit sequence. You can't use traditional references and pointers so you effectively define your own lvalue dummy type to represent the target and bit offset. This is a common C++ idiom and an application of the PROXY pattern that is used in many libraries, the standard one included. You will find the technique fully described in [2] and [3].

The deficiencies in *bitset* appear more pronounced when you look at the draft standard document. The clause on containers starts off with the basic requirements for a container, i.e., what constitutes a container, and is followed immediately by the non-conforming definition of the *bitset* class.

Fixed opinions

With the exception of *bitset*, all the containers in the library have variable runtime size. While this is certainly the most flexible and the most common requirement, for some critical applications either behavioural specification or efficiency considerations can constrain the cardinality of a size at compile-time. A fixed size vector effectively behaves like a traditional C array with the added advantage of a glossy interface. On the downside, its type includes its size; a *fixed_vector*<**int,** *10*> could not be passed to something expecting a *fixed_vector*<**int,** *20*>. For the applications that genuinely need this type of class it is unlikely to be a problem.

They could always define a member template conversion operator or template constructor – Ed.

Where the allocator mechanism is defaulted but overridable for other containers, fixed size containers do not require an allocator to handle storage for their contained elements. Part of the reason for using a fixed size container is to eliminate this additional level of indirection, and consequently the exact memory requirements for a fixed size container are known at compile time. The memory used by the fixed container would be that of its immediate context: using the same storage as any enclosing structure, or on the stack if it is declared as a local variable, or on the heap if it is allocated by **new** or memory mapped if placement operator **new** is used.

The following is a light sketch of such a class. The remaining members can be filled out easily if you are familiar with the standard vector class:

```
template<class value_type, size_t
length>
fixed_vector
{
public: // types
    typedef value_type     value_type;
    typedef size_t         size_type;
    typedef ptrdiff_t
difference_type;

    typedef value_type*    pointer;
    typedef const
value_type*const_pointer;
    typedef value_type&    reference;
    typedef const value_type&

const_reference;

    typedef pointer        iterator;
    typedef const_pointer
const_iterator;
    ...
public: // capacity
    bool           empty()    const
       { return length == 0; }
    size_type      size()     const
       { return length; }
    size_type      max_size() const
       { return length; }
    ...
public: // iteration
    iterator       begin()
       { return base; }
    const_iterator begin()    const
       { return base; }
    iterator       end()
       { return base + length; }
    const_iterator end()      const
       { return base + length; }
    ...
public: // access (not checked for
        // exceptions)
    reference operator[](size_t index)
       { return base[index]; }
    ...
private: // state
    value_type base[length > 0
                    ? length : 1];
};
```

In terms of speed efficiency, objects of this class will fly like the wind. Code size, however, could become an issue if not handled carefully. I said that each different size constitutes a different type: this implies that a class is instantiated for every new size. Fortunately most of the members can be inlined, and other techniques – such as using **private** base classes that operate on **void*** – can be used to reduce any possible weight gain.

If general fixed size containers were present in the standard, the *bitset* class could be killed off in favour of the **bool** specialisation of a

fixed_vector. Other modifications would have to be made to the basic requirements for containers to allow fixed size containers, e.g., the mandated *swap* member must swap the state of current object with the operand in constant time, according to the standard, but will take linear time for a fixed size container.

Hashes to ashes

Where are the standard hash tables? Next to linked lists, the mainstay of any library or book of data structures is, without a doubt, the hash table. The STL already provides the associative containers *map* and *multimap*, and the auto-associative containers *set* and *multiset*. However, the ordered iterator access requirements on these imply that the implementation is in terms of a sorted tree structure, which has logarithmic lookup time, rather than a hashed implementation, which has nearly constant lookup time.

The solution is not simply to weaken the requirements for associative containers, but to provide an additional set of requirements based on hashing associative containers. A library implementation would provide *hash_set*, *hash_multiset*, *hash_map* and *hash_multimap* classes.

Apparently hash tables were included in Alex Stepanov and Meng Lee's original STL implementation, but not – for some reason – in the original proposal. Javier Barreiro, Bob Fraley and David Musser made a proposal for their addition to the STL, but in the race for draft release the gate had already been closed on large changes. Many hope that the hash table model will become at least a de jure if not initially a de facto standard [4].

This issue was raised again in Monterey and the committee reaffirmed its position that the library must gain no more weight – Ed.

Kevlin Henney
kevlin@wslint.demon.co.uk

Notes and references

[1] "The Draft Standard C++ Library", reviewed in CVu 7(3), was an unfortunately premature look at the C++ standard library by P J Plauger.

[2] "Advanced C++ Programming Styles and Idioms" by James O Coplien looks at proxies for overloading the subscript operator.

[3] "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides describes and applies the proxy pattern.

[4] Documentation and implementations of the hash table model are also available from `ftp://butler.hpl.hp.com/stl`

C++ Techniques

This section will look at specific C++ programming techniques, useful classes and problems (and, hopefully, solutions) that developers encounter.

Ulrich Eisenecker's series on multiple inheritance continues, Roger Lever follows up his campaign for real inheritance and Peter Wippell shows how RTTI solved his problem.

Multiple inheritance in C++ – part II

by Ulrich W. Eisenecker

In the first part of this series, I showed why multiple inheritance is sometimes necessary and introduced some simple examples. In this article, I introduce virtual base classes with an example based on combinatorial maths.

Combinations

The following example, which is adapted from [EIS91], produces a complete series of combinations in lexical order. What does that mean? Well, suppose you want to crack a combination lock. A typical lock of this kind has three rings, each with numbers from zero to nine. A systematic approach to open the lock would be to start with combination 0-0-0, then change to 0-0-1, then to 0-0-2, and so on. The lock will be open by the time you reach 9-9-9 (hopefully, a long time before). A different example for such a series of combinations would be to generate all possible outcomes of a lottery with numbered balls, for instance the "Lotto 6 of 49", which is very popular in Germany. Of course, with 13,983,816 combinations this would be very tedious work, even for a computer.

Four different situations should be distinguished. To illustrate them, one can think of an urn filled with uniquely coloured or numbered balls. The first situation is when the balls are taken out of the urn, the numbers are noted in order, and each ball is returned immediately. Formula (1) computes the number of possibilities when the balls are replaced and their order is important. Formula (2) applies when the balls are replaced, but their order is not considered. Formula (3) is to be used when the balls are not replaced, but their order is important. Finally, formula (4) is used when the balls are not replaced and their order is of not important. The number of balls is denoted by n and the number of draws by k.

$$(1)\quad n^k$$

$$(2)\quad \binom{n+k-1}{k} \quad \frac{(n+k-1)!}{k!(n-1)!}$$

$$(3)\quad \frac{n!}{(n-k)!}$$

provided $1 \leq k \leq n$

$$(4)\quad \binom{n}{k} \quad \frac{n!}{k!(n-k)!}$$

Generating combinations

But the aim is not to compute these formulas. Rather we are interested in generating all possible combinations under the described conditions. Lexical order means, that

- an order relation is defined and, that
- according to this order the resulting draws are strictly sorted from low to high.

That is easy to achieve, when balls are replaced and their order is important. It is simply to count in a numeral system based on n from the lowest number up to the maximal number which can be represented in this numeral system with k digits. Here is a complete example for three balls and drawing two balls each time, which results in nine different combinations:

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

What happens if the order in which the balls are drawn is not important? Again the answer is simple: all combinations which consist of the same balls as a previous combination are deleted. The example from above looks now like this:

```
0 0
0 1
0 2
1 0 deleted
1 1
1 2
2 0 deleted
2 1 deleted
2 2
```

If you look at which of the combinations were deleted, you can see that the numbers of an individual valid sequence are in lexical order and those of an invalid sequence are not! So a simple test that the elements of a combination are in ascending order is sufficient to detect combinations which must be deleted.

Now we return to our original example and delete combinations where a ball occurs more than once in a single sequence. This is an easy task:

```
0 0 deleted
```

```
0 1
0 2
1 0
1 1 deleted
1 2
2 0
2 1
2 2 deleted
```

Finally we have the situation of the "Lotto 6 of 49", where it does not matter in which order the numbers were taken out and where it is impossible to draw the same number more than once. It is only important to have the right numbers. This leaves us with the following:

```
0 0 deleted
0 1
0 2
1 0 deleted
1 1 deleted
1 2
2 0 deleted
2 1 deleted
2 2 deleted
```

Now we can go on to design the classes we need for the four different combination generators, of course using multiple inheritance.

Implementing the generators

The root class is called *AllCombs*, which is abbreviated from "all combinations with duplicates and importance of order". Its public services are its constructor, which allocates memory for the combinations and initialises it, *reset*, which initialises the array of numbers in the combination, *n* and *k* which return the corresponding values, with which *AllCombs* was initialised. The value of a combination at a given position between 0 and *k*-1 is returned by *valueAt*. The **operator**<< is defined for printing the actual combination. The most important method is *nextCombination*, which generates the next valid combination in lexical order. When there is no next valid combination, *FALSE* is returned and *TRUE* otherwise. The destructor is necessary to return the memory occupied by combination. There are several protected data members. Among them *combination*, which is a pointer to an array holding the combination. The values of *n* and *k* are held by *n_* and *k_* respectively, *firstTake* indicates that the urn has just been initialised, and *cursor* is used as an internal marker for speeding up the generation of the next combination.

```
class AllCombs
{
public:
    AllCombs(unsigned N, unsigned K);
    void reset();
    unsigned n();
    unsigned k();
    unsigned valueAt(unsigned index);
    virtual BOOL nextCombination();
    virtual ~AllCombs();
friend ostream& operator<<(ostream&,
                           AllCombs&);
protected:
    unsigned n_, k_, cursor;
    BOOL firstTake;
    unsigned* combination;
};
```

NoOrd, which is derived public from *AllCombs*, generates combinations without ordering being important. It has its own constructor which only calls the constructor of *AllCombs*. Of course *nextCombination* must be overridden and a protected method *needsSorting* is necessary which checks whether a given combination is in itself lexically sorted and therefore a valid combination.

NoDup is very similar to *NoOrd*, except for the method *duplicates*, which checks for repeated values.

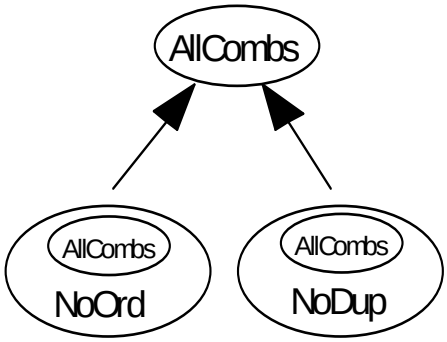


Fig 1: AllCombs, NoOrd and NoDup

To derive the class *NoOrdNoDup* multiple inheritance is used. In *nextCombination* the inherited tests *needsSorting* from *NoOrd* and *duplicates* from *NoDup* serve to check if a combination is valid.

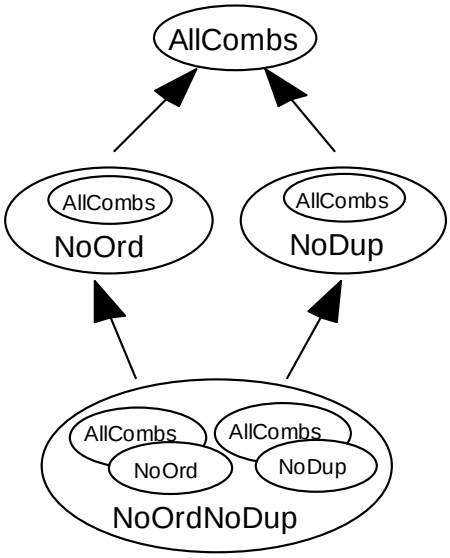


Fig. 2: A problematic NoOrdNoDup

After examining figures 1 and 2 thoroughly it does not come as a surprise that the compiler will complain about *AllCombs* being an ambiguous base class of *NoOrdNoDup* and about *n* and *k* being ambiguous members. There is a similar problem as with the *TwinPhone* (in the first article), but now propagated through the inheritance hierarchy. Therefore a mechanism is needed to eliminate unwanted duplicates of common ancestors. Unfortunately this cannot be controlled when deriving *NoOrdNoDup*. Rather it is necessary to make *NoOrd* and *NoDup* use *virtual inheritance* by deriving them **virtual** from *AllCombs*. The correct declarations for *NoDup* and *NoOrd* to enable elimination of duplicate members are:

```
class NoOrd: virtual public AllCombs
{ ... };
class NoDup: virtual public AllCombs
{ ... };
```

The declaration of *NoOrdNoDup* is not affected.

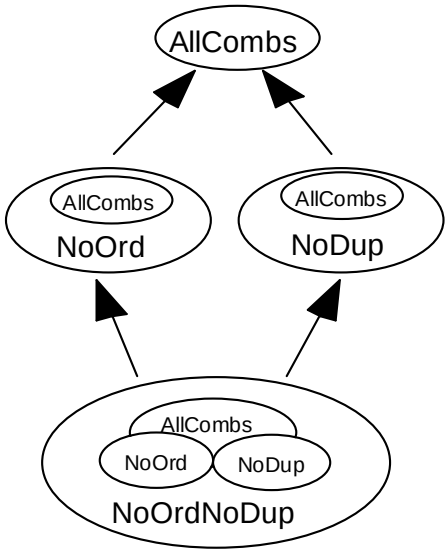


Fig. 3: A well designed NoOrdNoDup

One important peculiarity concerning initialisation has to be mentioned. A common base (a **virtual** base class) is initialised only once, regardless of how often any of its constructors are called. This is done automatically if a default constructor for this class exists. If not, the constructor must be called explicitly from the constructor of the most-derived class. Any additional direct or indirect calls to constructors of this common base class are ignored (cf. [STR93], pp. 580ff).

Since in the example of combination generators only

```
AllCombs::AllCombs(unsigned n,
                   unsigned k);
```

exists, the initialisation of *AllCombs* can not be done automatically by calling a default constructor. From the point of view of good design, there is no need to introduce a public default constructor, which can only cause harmful behaviour if called accidentally for an instance of *AllCombs*. Therefore that constructor must be called explicitly from the constructor of *NoOrdNoDup*. Any further indirect calls of *AllCombs* constructor via the constructors of *NoOrd* and *NoDup*, which must themselves be called by *NoOrdNoDup*'s constructor, are ignored.

A riddle

To demonstrate the usefulness of the combination generators here is a riddle to solve.

The Riddler blackmails Gotham City. He reveals that there is a bomb in the foundations of the town hall which has been activated recently. The

bomb is stuck in solid concrete, so it can not be removed. The mayor must pay a large sum of money to the Riddler, so that he will stop the bomb. But there is another chance – to disable the bomb. The correct combination of three ciphers must be entered in an electronic lock in the cellar, which controls the bomb. The riddle was like this:

"The first number is at the second number's position in the fraction of the square root of adding the first and the second number. The third number is at the second number's position in the fraction of the second number's root. You need two additional hints: none of the three numbers will occur twice and the numbers are in ascending order! Be careful, you have only one attempt! If you are wrong, it will be the last error in your life. It seems better for you to pay. Har har har ..."

The mayor calls Batman for help. Batman reflects and starts to program his computer. After half an hour the computer prints out three numbers, which Batman presses immediately on the bomb's lock. Of course, the detonator stops running and Gotham City does not need to pay the riddler.

How can we do this too and find the right numbers? Try to design the solution yourself before you study and run the program, the *main* routine of which is listed over the next page.

Next article

The focus of the next article will be mainly on multiple inheritance and design. The circumstances under which multiple inheritance can be used and what to be aware of will be discussed in depth.

References

[EIS91] Eisenecker, Von Urnen und Kugeln. Ziehungsgenerator für Kombinationen und Variationen. In: c't 11/91, S. 172-178

[STR93] Stroustrup, The C++ Programming Language, 2e. Addison-Wesley, reprinted with corrections, 1993

Can't find it? – I

If you're wondering where part III of my cOOmpiler writer series is, don't worry, it'll be back in *Overload 10*. With the committee meeting and an overdue product release, I was unable to devote as much time to the column as I would have liked and had to postpone it. With my forthcoming job change – from Development Group Manager at Programming Research to my own company, Object Consultancy Services – I will also have to change the direction of the column slightly but I still intend to examine various problems that beset compiler writers and developers alike!

Sean A. Corfield
Technical Director, OCS
ocs@corf.demon.co.uk

Can't find it? – II

Here are some useful URLs for information about C++:

`http://www.maths.warwick.ac.uk/c++`
: The official UK C++ web site maintained by Steve Rumsby. There are links from here to lots of other useful resources, including the Virtual C++ Library and various BSI and ISO standardisation resources.

`http://www.cygnus.com/~mrs/wp-draft/index.html`
: A browsable version of the latest draft C++ standard made available by Mike Stump of Cygnus – the GNU software support folks.

`http://metro.turnpike.net/S/scorf/cplusext.html`
: One of my pages that provides examples of the extensions the committee have added to C++ since the ARM was published. The pages accessible from here are still under construction and change fairly regularly. This, and all my other pages, will shortly move to `up-town.turnpike.net/~scorf/` (and may, in fact, have moved by the time this issue is delivered). You'll probably need Netscape to browse this page because it uses tables!

`http://www.cs.rpi.edu/~musser/stl.html`
: Lots of information about the Standard Template Library including example programs, documentation, history and philosophy – an essential read if you are interested in STL.

`http://bach.cis.temple.edu/accu`
: Alex Yuriev's excellent Association of C & C++ Users home page!

If you know of other useful URLs, please let me know and I will include them in a future issue of *Overload*.

```
// nThNumberOfFraction
//     extract the nth digit of the fractional part of the floating point value passed
unsigned nThNumberOfFraction(
       double       f,
       unsigned     n
)
{
       for (int posn = 0; posn < n; ++posn)
       {
              f = 10.0 * f - 10.0 * unsigned(f);
       }
       return unsigned(f);
}

int main()
{
       // 3 digit lock, 0-9 on each:
       NoOrdNoDup   urn(10,3);

       while (urn.nextCombination())
```

```
        {
                // the first of The Riddler's conditions:
            if ( (urn.valueAt(0) == nThNumberOfFraction(sqrt(urn.valueAt(0) +
                                                        urn.valueAt(1)),
                                                    urn.valueAt(1))) &&

                // the second of The Riddler's conditions:
                (urn.valueAt(2) == nThNumberOfFraction(sqrt(urn.valueAt(1)),
                                                        urn.valueAt(1))) )

                // nextCombination ensures the values are in lexical order
            {
                cout << "To disable the bomb, press " << urn << '.' << endl;
            }
        }
}
```

Ulrich W. Eisenecker

eisenecker@dbag.ulm.DaimlerBenz.COM

The complete code will be on a forthcoming CVu disk and then on Demon for anonymous ftp – Ed.

On not mixing it...again

by Roger Lever

My original intention in "On not mixing it" [1] was to question the use of the mixin programming style. To constructively criticise, I provided an alternative implementation, but unfortunately I explained the rationale rather inadequately as The Harpist indirectly noted [2]. I will redress that balance now – or at least make a better attempt! To understand why I question the use of mixin programming, I shall make a case for 'proper' inheritance and the design choices made for that approach. So, starting with design...

Analysis, design & abstraction

Before we start to design a solution we need to fully understand the problem. We build a model of the problem domain by using abstractions. This model is then refined and used to solve the particular problem. It may *sound* simple but it isn't. This process is not C++ specific but it is well worth investing some time in it. There are many good books devoted to this area but my personal favourite is Booch's book [3]. *Overload 8* also touched on this very important subject [4].

Classes are used in C++ to represent the fundamental concepts of the problem domain or "reality" being modelled. To quote Stroustrup [5]:

> *"A well designed system will contain classes supporting logically separate views of the system. For example:*
>
> 1. *classes representing user-level concepts (e.g. cars & trucks)*
> 2. *classes representing generalisations of the user-level concepts (vehicles)*
> 3. *classes representing hardware resources (e.g. a memory management class)*
> 4. *classes representing system resources (e.g. output streams)*
> 5. *classes used to implement other classes (e.g. lists, queues...)*
> 6. *built in data types and control structures"*

Naturally to go from analysis and design to delivered executable will incorporate many decisions regarding these classes, however, I want to focus on inheritance. This is the mechanism that C++ uses to generalise concepts and mirror them in the solution domain and in the process reuse code. An important point is that concepts do not need a physical counterpart: concepts can and do include abstract items such as events and roles.

Inheritance & reuse

Reuse *used* to be synonymous with inheritance – no need to write new code, derive a new class and hack that. This was of course a major step forward from the previous mechanism for reuse, namely copy-paste-and-hack.

Problems emerged from this approach which led to rules-of-thumb about what inheritance was and how and when to use it. This body of wisdom encapsulated simple rules of thumb such as *IsA* and *HasA*. However, we need to emphasise some vital points for proper inheritance:

- Context – specific problem domain and requirement
- Perspective – viewpoint of the user, designer and implementor. Context and perspective are everything. Understanding this enables us to produce better abstraction models since we know what the model is doing, why and for whom.

Context & perspective are everything

Let's explore the vehicle-car-wheel trio. Using Stroustrup's terms, car and wheel are user-level concepts and vehicle the generalisation. What are we assuming? That the concepts are related, which may be true or it may not. How do we decide if these concepts are related? By understanding the context and the perspective being used.

1. What is a vehicle and what does it do?
2. What is a car and what does it do?
3. What is a wheel and what does it do?

Generally, we would accept the above trio as being related and would automatically assume the missing details such as :

- vehicle – types of transport (car, truck, bus...)
- car – the family-sized vehicle that gets us from A to B
- wheel – keeps a vehicle rolling along from A to B

But what if we applied a different context? For a fairground ride the vehicle might generalise the rides available, including a kiddie car which is very different from a normal car. Wheels in this instance are redundant or decorative – such "cars" tend to be fixed to a roundabout or run on rails. We could rework this example in any number of scenarios but the key is knowing what behaviour or services we expect from the concepts in the abstract model and the perspective.

Perspective is important since it qualifies to whom the abstraction is relevant and helps avoid confusion based on similar vocabulary. For example, using the concept 'queue' – what is that and what does it do? From each perspective it may be something very different:

- User – a queue of cars on a production line, or a traffic jam
- Designer – an operating system service to handle system requests
- Implementor – data structure with FIFO semantics

What we should expect is governed by requirement, specification and perspective. The user's expectation will be based on user-level concepts such as *car* and what services a car would offer the user. The designer's expectations will generalise these from those original concepts and cover a set of services or behaviours that are appropriate to the abstract model. Cargill [6] describes this as "Concentrate common abstractions in a base class". The designer will also need to consider the solution domain and design for that as well, such as printers and screen output. A basic definition, to horrify language lawyers, of interface and implementation:

- Interface – publicly available services or behaviours (what it does)
- Implementation – private mechanism used to implement the interface (how it does it)

The implementor's expectation would be in terms of data structures and algorithms. The expectation would be that the implementation would be private and users of the class would use the public interface to access the services offered. The division and roles are somewhat arbitrary, for example Murray [7] states:

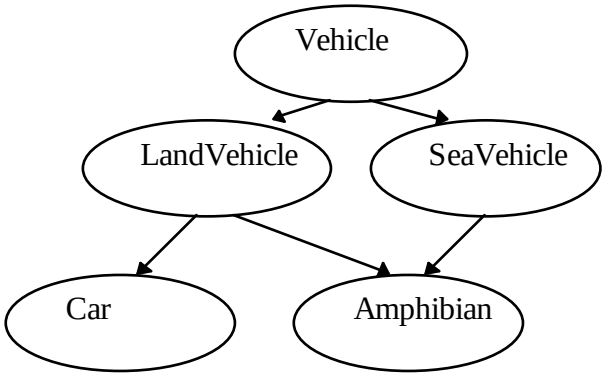
- Designing the abstraction and designing the implementation should be two separate, but related activities
- What is *not* in the abstraction is as important as what is *in* the abstraction (Mural's emphasis). However, it is useful to separate the concerns to simplify the complexity. Also as a matter of principle we want to maintain the separation of the interface from the implementation as much as is reasonably possible.

It follows from the logically separate views of the system, that each view could have its own inheritance hierarchy. So a car manufacturing application could be composed of three distinct hierarchies, reflecting the problem domain (car), the designer domain (vehicle, output...) and the implementor domain (list, queue...).

In each of these views the inheritance hierarchy needs to pass certain litmus tests such as:

1. A class should describe a set of objects [6]
2. A Derived (car) IsA Base (vehicle)
3. A Derived (car) is a subtype of the type Base (vehicle)
4. A Derived (car) is substitutable for a Base (vehicle)

This last rule is known as contravariance [8] and is the *real* test for proper inheritance. Multiple inheritance does not change the intent of, or remove, any litmus tests. This hierarchy is fine:



Since an *Amphibian IsA LandVehicle* and *IsA SeaVehicle*, using multiple inheritance instead of single inheritance does not change the criteria applied – it simply adds another 'and' clause.

If we used this hierarchy instead of (*Vehicle –> Car*) it would require substantial changes: a) *Vehicle* would need to be declared **virtual** b) the most derived class must initialise the virtual base

class. This is an important point in terms of designing for inheritance.

Interfaces and implementations

Using multiple inheritance still models the problem domain and still passes the substitution test. The problem domain is distinct from the solution domain, but in practice we need to combine all of the classes into a cohesive solution to the problem. In particular we need to map the solution hierarchies of the designer and implementor to the user's problem domain hierarchy. This is where the mixin style enters the equation offering an apparently simple solution.

The user-classes need a mechanism to operate within the solution domain. The mixin approach is to use inheritance to provide 'car' with additional, public interface, services such as the ability to save state to disk, send output to the printer or display information to the screen. However, this form of inheritance is not in keeping with substitutability and is used only as a reuse mechanism for the implementation. Cargill [6] notes for single inheritance:

- Recognise inheritance for implementation;
- use a private base class or (preferably) a member object

Inheritance supplies both an interface and implementation depending on how it is used. For example, an ABC (Abstract Base Class) composed entirely of pure **virtual** functions supplies only the interface and no implementation. Private inheritance supplies the implementation but no interface. To use inheritance as a design tool, rather than a reuse mechanism we need to apply the criteria of substitutability.

Traps in applying inheritance

Inheritance should not be used for specialisation or subsets. Specialisation is too vague or as Cline [8] states "A major source of confusion and design errors...Forget specialisation and learn about substitutability". The use of inheritance with subsets came up in *Overload 8* and Kevlin Henney [9] is correct to say "...the problem is poorly stated". The context is everything. What services the abstract model of *Ellipse* supports will determine if one of these statements is false:

1. Every *Ellipse* can be resized asymmetrically
2. *Circle IsA Ellipse*
3. A *Circle* cannot be resized asymmetrically.

If one statement is false then we have the following options:

- Recognise that public inheritance is not appropriate
- Change *Ellipse* to not include asymmetric resizing

Of course if we were desperate we could forget public inheritance, use private inheritance and override the asymmetric resizing functions, but that may end up as a surprise for someone later.

Mixin alternatives

I have taken some time in explaining design issues and inheritance to explain my position on the mixin programming style. In *Overload 7* [1] I presented one alternative, which also used the key feature that inheritance supplies – polymorphism. The mechanism shown had polymorphic behaviour both vertically *Record->ExtendedRecord* and horizontally *Device->Printer|Screen|Disk.*

Peter Wippell [10] states "Surely the stream library is complicated enough without introducing another layer of classes!". The intent was to add that layer as another hardware abstraction layer. Previously, particularly with printers and floppy disk drives, I have found that the stream is fine but the device at the other end is not:

1. Printer not connected, out of paper, offline, paper jammed...
2. No disk in drive, write protected, disk full...

So to have both polymorphic selection and device safety I put that layer in. In retrospect there is a problem with the design. It is likely that a *Record* will know where the output is going in terms of the screen, disk or printer and will format the output differently for each device. So perhaps the horizontal polymorphism (can I patent that phrase?!) is redundant or overkill. Thankyou for that point of detail on *strstream* – which I believe is deprecated now? (If so, it will stop me from using it).

> *Yes, strstream is deprecated in favour of stringstream – Ed.*

This mechanism was designed as an alternative to the mixin style. A number of other approaches are possible dependent on the requirement for polymorphic behaviour and the expectation of

deriving further classes. It is not intended to represent *the* way to add in these mechanisms, it is just a catalyst for looking at the issue in a different way.

Summary

The crux of the issue is: how do disparate classes communicate and collaborate to solve the problem at hand. This is a subject of active study and work and is certainly something which will affect everyone with an interest in OOP. However, for now, inheritance should be used as a design tool and not as a mechanism for reuse of the implementation.

Roger Lever
rnl16616@ggr.co.uk

References

[1] *Overload 7*, "On not mixing it", Roger Lever

[2] *Overload 8*, "Having Multiple Personalities", The Harpist

[3] Benjamin/Cummings, "Object-Oriented Analysis and Design with Applications", Grady Booch

[4] *Overload 8*, "OOA – The Shlaer-Mellor Approach", David Davies

[5] Addison Wesley, "The C++ Programming Language Second Edition", Bjarne Stroustrup, Chapter 12, Section 12.2

[6] Addison-Wesley, "C++ Programming Style", Tom Cargill

[7] Addison-Wesley, "C++ Strategies and Tactics", Robert B. Murray

[8] Addison-Wesley, "C++ FAQ Frequently Asked Questions", Marshall Cline & Greg Lomow

[9] *Overload 8*, "Circle & Ellipse – Vicious Circles", Kevlin Henney

[10] *Overload 8*, "A "too-many-objects" lesson", Peter Wippell

Another "too-many-objects" lesson

by Peter Wippell

I have taken some rather large editorial liberties with this article since it is essentially an update of last issue's article – I hope Peter doesn't mind too much! – Ed.

In *Overload 8*, I showed how to write records polymorphically to a generalised device stream. I stated that there did not appear to be an easier way to establish whether a stream was a printer than to "invent" a *Printer* class derived from *ofstream*. After consulting the Borland help files, I found a more direct way!

An enquiry function, *isPrinter***(***ostream***&)** employs Run Time Type Identification to find out if the *streambuf* of the device in question is a *filebuf*. If it is, it can call the *filebuf* member function, *fd***()**, and identify the printer from its predefined MS-DOS file descriptor:

```
bool isPrinter(ostream& os)
{
  // note: condition is a declaration of
  // pfb and is true if the dynamic_cast
  // succeeds
  // i.e., returns a non-null pointer:
  if (filebuf* pfb =

dynamic_cast<filebuf*>(os.rdbuf()))
  {
    // it is a filebuf, is it the
printer?
    return PRN_file_handle == pfb->fd();
  }
  else
  {
    return false;
  }
}
```

I have supplied the complete code in case anyone wants to improve it.

Peter Wippell

The code will be on a forthcoming CVu disk – Ed.

editor << letters;

Dear Editor,

I'm converting a large DOS application which uses a proprietary database (CTree from Faircom) to C++, Windows and Client/Server (Watcom, MS SQL and Oracle targets). We've built an enquiry version in PowerBuilder (using a class library called PowerClass which we are very pleased with). We've also done other work in VB, so Windows knowledge is improving.

I'd like to find C++ class libraries for screen forms handling (MFC etc. may be OK for that) and "Data Windows".

I also need consultancy to give me the necessary training, advice and knowledge, and possibly contract programmers too.

I seem to be finding it very difficult to find the right people to get connected with. A C++ training course for C programmers is easy to find – but they seem to want to spend four days teaching me how to draw a circle etc. and one day showing me the database stuff, and I'd like it the other way round! – but there may be a reason for that.

It may also indicate that C++ programming for database is not done these days and people use PowerBuilder or whatever; I've got a legacy in C which I would prefer to carry forwards to the new product – if we rewrite it in PowerBuilder, say, it will need massive testing and all our clients will insist on full dual running when they upgrade, if we just build screen and database handlers, and keep the old code for data validation, transaction processing, etc., then the dual run should be achievable by a short pilot run.

Perhaps I should be building this product with a view to selling it to fill a hole in the market?

Any ideas?

Kristen Baker-Munton
IPSS Limited
Bentons West
Bildeston
IPSWICH, IP7 7JR
Tel 01449 741777 Fax 740202
kristen@ipss.com

> *This seems to be a common problem: I see many companies trying to make the move to C++ for various reasons and appropriate training and consultancy is hard to find – the operative word being "appropriate".*
>
> *I'm sure that database work is being done in C++, so where is the training? If anyone can help Kristen, please get in contact!*

Dear Sean,

You invited comments on namespaces, so here are mine. Please bear in mind I'm only starting out with C++.

I would have thought that when searching for a variable the search should have gone:

```
Local => (Namespace if specified) =>
Global
```

From my understanding of your examples, which may be limited, it seems if a global variable and a namespace variable exist and you then use the namespace and attempt to access the duplicated name the result is ambiguous. Why is this? Surely if you have specified a namespace it should have precedence over the global settings.

I do agree with finding of local variables first if they exist, but I really think that namespaces should be searched before globals.

I'm not so sure about the usage of two different namespaces with the same variable names, what happens if you have

```
namespace A {
  int j;
}

namespace B {
  using namespace A;
  int j;  // Or perhaps even worse long
j,
          // or int *j
}
```

What happens in this situation?

Regards,

Barry Dorrans
BarryD@phonelink.com

> *You're not alone in expecting the more locally specified "using namespace X;" to be searched prior to the global scope!*
>
> *Despite appearances however, the using-directive is not a declaration of any sort: it just says "perform the usual name lookup, but if you get to file scope, also look in this namespace".*
>
> *I'll write up a more detailed examination of namespaces in Overload 10 to try to dispel the confusion that currently surrounds them.*

Hi Sean,

Just scanned through *Overload 8* and, well, I'll stick my neck out and assert a mistake in the article by/about the authors of the Ellemtel standards. (Didn't Feynman say something along the lines of "we won't find out where we're wrong unless we stick our neck out"?)

On page 43, in the first 'code' box the following fragment is shown:

```
char a[]="abc";
a[1] = 'x'; // undefined behaviour [sic]
```

I reckon that this example ought to be using not a **char** *array* but a **char** *pointer*. It was my belief that an *array* definition (as above) actually allocates space for the correct number (here, 4) of **char**s, and that this space is read-writable.

On the other hand, the similar-looking *pointer* definition:

```
char* pc="abc";
```

does *not* allocate memory for the four characters that would make up the quoted string, and a write such as:

```
pc[1]='x';
```

would be undefined behaviour.

I wouldn't bother pointing this out were it not for the fact that a lot of people (me included) read ACCU publications in order to learn and improve; this process is hindered by subtle mistakes.

Yours (in a particularly pedantic frame of mind),

Fazl Rahman
fazl@hadronic.demon.co.uk

*You're absolutely right, Fazl! I should have spotted this when I edited the article. For the point that Mats and Erik were making, both examples (with and without **const**) should have used pointers – just goes to show that even 'old hands' are fallible!*

Dear Sean,

Dave Midgley complains (Letters, April issue) that C++ code is always littered with *getAttribute***()** member functions, and suggests that it would be an improvement to C++ if member variables could be made private for writing while public for reading. In my view this is a tempting but bad suggestion! The significance of data abstraction extends beyond preventing the encapsulated data from being changed by functions outside the class. The public member function that returns the data is the class's interface to the rest of the world. The function may currently just return the value of a variable without doing anything else; the variable's name may be implied by the function's name; the internal storage type of the variable may be known. But things may change one day. The use of the member function ensures that the interface to the class remains the same, so that internal changes within the class don't affect the rest of the world.

Peter Arnold
peter.arnold@iccs.sil.org

I agree – the classic example usually given is Point – whose coordinates may be polar or rectangular:

```
class Point
{
public:
  double  getX() const;
  double  getY() const;
  double  getR() const;
  double  getTheta() const;
//...
};
```

Written in this way, you have a choice about representation and can change it later on without needing to change any client source code.

Exercise for the reader: how would you write Point so that you could change the representation without needing to recompile any client code, just relink?

Questions & Answers

Got a C++ problem? Not sure whether it's you or the compiler? Send it in and *Overload* will try to sort you out!

Sean,

I was using the *auto_ptr* class *[from the draft standard library]* in BC++ when I noticed:

```
template<class T> class A
{
public:
       T* operator->() const;
};
int main()
{
       A<int> a;
       return 0;
}
```

did not compile. Using a user defined class in *main* instead of **int** is fine. Is this a bug in BC++?

Do you know the relevant reference in the ARM which specifies whether this is legal or not?

Joseph Borkoles
jborkole@jpmorgan.com

*The ARM specified that the return type of **operator->** must be a pointer to a class. If you write a pointer-like template class then you have the problem that you need two versions: one with **operator->** that can only be instantiated with class types, and one without that operator that can be used with builtin types. The committee decided that this was not really acceptable and decided that checking the return type of **operator->** should be delayed until the point of use – for template members only! So, BC++ isn't really wrong, it just hasn't caught up with the draft standard yet.*

Interviews

Overload is always glad to feature "virtual" interviews with well-known names from the C++ world. If you want to see an interview with someone – especially if you're willing to conduct the interview – please let the editor know!

In this issue, Roger Lever interviews the author of *Taming C++*.

Interview with Jiri Soukup
by Roger Lever

Having read a good book called Taming C++ by Jiri Soukup (Addison Wesley) I thought it would be interesting to ask the author a few high level questions about C++ for the ACCU. Jiri Soukup was happy to oblige:

Why did you write 'Taming C++'?

I wanted to cover two subjects that are almost entirely missing in the existing literature:

1. How to implement large C++ projects without introducing a confused network of mutually interacting classes (spaghetti++).
2. Practical insight into what is involved in implementing persistent objects in C++.

In terms of OOPL and 'IT Solutions' how would you position C++ and Smalltalk? There was an interesting comparison in Taming C++...

This is a nice way to get into a big controversy.

Personally, I definitely favour C++, even for prototyping. The typelessness of Smalltalk opens the gates to numerous errors and, in my opinion, quickly hacked code is not the best strategy even when designing a prototype. Recently, I was involved in a big project which was prototyped in Smalltalk, and implemented in a different language. Due to the typelessness of Smalltalk, it was difficult to understand the prototype (you had to depend entirely on variable names), and the first implementation had serious performance problems caused by the different concepts of the two languages. C++ does have problems with allocation and pointers, which are often quoted in favour of Smalltalk; these can easily be prevented when using the techniques shown in "Taming C++".

Two small comments:

One of the alternate titles originally proposed for "Taming C++" was "Designing Large Projects in C++". That title would better emphasise that the book provides ideas on how to keep complex C++ architectures under control.

Also the shelving term (which is what publishers call the word(s) printed on the upper left corner of the back cover, whose purpose is to help bookstores place the book correctly with other related titles, is "Programming Languages/C++", which has caused the book to be placed with C++ textbooks, not books on OO design and OO methodologies.

What are essential characteristics of quality C++ programs?

- Clarity; one should be able to understand the program from reading the code. Implementing relations as objects may help in this goal.
- Automatic protection against pointer and/or allocation errors which cause the most dangerous situations in C and C++.

- Persistent data handled as a "system feature", without having custom coded functions such as *saveGuts***()** in every class.
- Deep inheritance hierarchies, multiple inheritance, and virtual functions should be used only when really needed – as little as possible.
- Quality programs avoid special language features and smart tricks. If such tricks are used, they are flagged in the code and properly commented so that even a non-expert can understand the code.

What are your thoughts about reuse in C++?

The key to reuse is in the communication between all designers participating in the project.

In a large project I managed some years ago, our team of 10 people met every day for about an hour and discussed the progress daily. No duplication of code or algorithm was permitted, under the threat of being fired.

What advice would you offer to beginner C++ programmers?

Get Bjarne's book (The C++ Programming Language), forget all other books that try to "explain" the language, and start coding. If you can, find a friend who knows C++ and is willing to answer questions. Return to other books later, when you start to use more advanced features of the language. Keep two different compilers on your computer – for example Borland C++ and Watcom C++. Often, when one compiler's error message does not give you enough clues, the other one will.

What advice would you offer to intermediate C++ programmers?

Keep reading and learning, but do not forget that more than 50% of software cost is in maintenance. Keep your programs simple and easy to read – I just cannot overemphasise that point. In C++, there are so many features and tricks, it is easy to produce totally unreadable code that's impossible to maintain.

What are 5 of your 'Golden Rules' of programming?

1. A program reflects the state of the mind of its creator. Confusion creates confused programs. Avoid coding on those days when things don't work your way.
2. When you reach the point that you don't understand your own code, add comments or redesign it immediately. If it is difficult to understand its logic now, it will be a nightmare to do anything with it later on. If you don't understand it well, nobody else will.
3. Program is a living entity which is never finished. Code with this in your mind. Leave comments, hooks, clues, and explanations to help possible additions. I use an error message even for conditions that "should never happen".
4. An extensive test should be a part of every program right from the beginning.
5. Prototype in the language you plan to use for the final product.

What are the major C++ trends you see developing?

I am somewhat unhappy about several trends which, I believe, will eventually reverse.

In my opinion, STL is a poor choice for the basic class library: it is totally unprotected, and does not address object persistency. I believe that design patterns will replace what we used to call data structures, but we will have to develop methodology for implementing patterns and building libraries of reusable patterns. Also, and that is a totally different trend, I don't like Windows and the general emphasis on graphical interfaces. Programming should not become more difficult just for the ease of displaying pretty pictures.

What do you mean about STL being unprotected?

The worst C (and C++) errors are incorrect pointers or pointers leading to objects that were destroyed without being disconnected from some list. These errors may stay dormant in your code for a long time, and then they suddenly show up – usually by crashing your program. Such errors are often difficult to find; one such horror story is described in Chap.3.2 (p.91) of "Taming C++".

When using STL, you can place an object on a list and then destroy it. When you traverse the list, the program will crash. "Taming C++" shows how to design a class library so that this type of error cannot happen. There are commercial libraries that protect against pointer errors.

Have you started to favour a particular approach to Design Patterns?

It is too early to talk about different approaches – the entire field is in flux. However, I think that the next steps will be more concern about the low level implementation issues, not just about abstract patterns that apply to the high, architectural level. The central step for the improvement in both reuse and maintainability will be to make patterns visible in the final code, even after the original coder and designer are done (and perhaps long gone).

What impact would these major trends have on development in C++?

Perhaps, C++ objects could be automatically persistent. The compiler has all the information to implement this efficiently.

Also, I believe, with time various C++ code generators will become more accepted.

Libraries of ready-to-use patterns will be soon available.

What are you currently working on?

A book that will be probably called "Implementing Class Patterns" and it will show how to use patterns when actually coding programs, and how to design libraries of selected pattern implementations. The difference from "Taming C++" will be that all the code is based entirely on C++ templates; a code generator is not required except, perhaps, for reducing the code effort.

Thankyou for your time and effort.

Thank you, you posed interesting questions.

Roger Lever
rnl16616@ggr.co.uk

Some of Jiri's remarks, particularly about STL, I disagree with and I'm sure that several other readers may have something to say on this subject too? – Ed.

Books and Journals

Overload would like to set up a small book review panel, consisting of experienced C++ developers to write in-depth reviews of C++-specific books. Please contact the editor if you are interested or want more details.

Design Patterns

reviewed by Sean A. Corfield

Title:	Design Patterns – Elements of Reusable Object-Oriented Software
Authors:	Gamma, Helm, Johnson, Vlissides
Publisher:	Addison-Wesley, 1994
ISBN:	0-201-63361-2
Price:	£28.95
Format:	hardback, 400 pages.

Patterns

Patterns are probably the hottest topic in OO at the moment and this, the "Gang of four" pattern catalogue, the most widely praised. So what's all the fuss about?

At its core, this book has a catalogue of twenty-three "design patterns". The patterns are design-level templates for creating solutions to common problems. And that's it, really.

"That's it?" you say. Well, yes and no. What makes this book so special is simply that no-one has taken the trouble to distil this problem commonality, categorise it and write it up in a form that programmers and designers can actually understand.

A roadmap

The book is in three sections (despite the claim of "two main parts" in the preface). The first section attempts to explain what patterns are and how you use them. The third section looks at where we are now and where we might be going. The catalogue of patterns makes up the central, and largest, section of the book (270 pages).

It is probably worth quoting a line from the Preface: "A word of warning and encouragement: Don't worry if don't understand this book completely on the first reading. We didn't understand it all on the first writing!". My first reaction was one of disappointment because the

pattern descriptions were just that: descriptions. They didn't seem generic enough and the code fragments given were often for specific examples. So I put the book down for a couple of weeks and then started reading it again.

Hard work

This book makes you work! Hard! On subsequent readings I began to appreciate this more and more because I began to see two things: patterns that I had unknowingly already applied and patterns that I could have applied. The former gives you insights that help you solve other, similar problems more quickly. The latter tends to make you curse, because if you had applied the pattern, the end result would have been more elegant and more flexible!

Whilst the amount of applied thinking that the book requires is unusually high compared to the norm these days, the authors have provided plenty of hints and tips on how to best use the material in the book. They provide several suggestions for ways to read the book as well as how to use it to solve particular problems. The latter section (§1.6) is particularly helpful as it takes you through various parts of the design process, pointing out how various patterns fit in to different scenarios that you might be trying to solve.

Organisation

For a catalogue, the book is extremely readable because the authors have adopted a clear and consistent method for documenting the patterns. Each pattern is explained by stating what it is intended to do, why you might want to do that and when you can. The components and interactions behind each pattern are then explained with a mixture of prose, OMT, Booch and examples, before moving on to the "how". Finally, example code is given in C++ or Smalltalk (or both) and some real world uses are mentioned.

This means you can quickly establish whether the pattern is useful or interesting, and as you read further you get more detail and more hints on how to apply it to your own problem.

Dipping in

I've found that the most instructive way to read the book is just to open it up randomly, flick back to the start of whatever pattern you're in and just start reading. You probably won't put it down until you've read several patterns. Over time, you'll absorb more and more of them – some are more intuitive than others. One thing that struck me was the variety of application domains from which the authors have drawn their examples: FACADE (compiler), CHAIN OF RESPONSIBILITY (help system), STATE (TCP communication), VISITOR (inventory / pricing). Of course, the usual graphics and text editor examples are also present.

Code fragments

Don't expect pages of C++ template-based source code – you'll be disappointed like I initially was. The code used to illustrate the patterns is mainly C++ with some Smalltalk but this is a book about design rather than about programming. If you want the generic pattern in code, you'll have to think hard and understand the pattern so that you can apply it to your own code or derive the template-based solution if one exists (it doesn't always).

Conclusion

You probably don't need me to tell you, but this is a very rewarding book. Buy it and dip into it a few times and you'll find yourself coming back to it time and time again. The reward comes from the "lightbulb effect" as patterns start to suggest themselves when you're designing systems later on.

Sean A. Corfield
sean@corf.demon.co.uk

Product Reviews

What development tools do you use? Do you want to review them for *Overload*?

UTAH – a short product report

by Francis Glassborow

When the telephone has not been ringing (over 40 times today) I have been testing a product from ViewSoft Inc called UTAH. I have been using version 1.1 of the product for MS Windows 3.1. Let me start by saying that I like the product so that my critical statements below are because I want it to be better.

The product is a tool for designing and developing GUI based products. It has a nice feel to it and I found it easy to use. Certainly inserting application specific code was clean and simple. After you have your design complete, with some facility for emulating the result, generating code and files for the compiler of choice was only a couple of mouse clicks away. I did not have to choose between Borland 4.0 and Visual C++ 1.5 until I was finished. When I selected my compiler UTAH went away and generated project files etc.

Unfortunately, at this stage, UTAH makes unwarranted assumptions about where its libraries are and where you will have installed your compiler. As my systems never have anything in the default location I had to patch the generated project file when difficulties manifested.

ViewSoft have plans to provide versions of UTAH for other platforms, though they only support Borland and Microsoft development tools on Windows platforms. When I asked about other compilers for the same platforms they said that they had surveyed the field and too few of their potential customer base used compilers such as Watcom and Symantec. Problems with things such as name mangling algorithms cause difficulty when you try to use another compiler (or so they believe – time has not allowed me to test this).

I think they have the wrong target. A large proportion of those using Microsoft or Borland compilers will be quite happy with the AFX builders that come with those products and are not realistic customers for UTAH. It is those that want to transfer across compilers, and even more across platforms that are most interested in products like UTAH. It was nice to be able to delay the choice between Borland and Microsoft compilation tools. I would have been even more impressed had I been able to take a product developed with UTAH on a MSWindows machine and port it directly for compilation on an OS/2 platform.

It is those that want to work with multiple compilers or multiple platforms that have most to gain from tools like UTAH. As long as the UK distributor keeps me informed I'll let you know how the product develops. In the meantime, if you would like to know more or want to evaluate the product contact Professional Software Ltd on 01753 810 845 who are the UK distributors.

Francis Glassborow
francis@robinton.demon.co.uk

S-CASE

reviewed by Sean A. Corfield

Product:	S-CASE
Company:	MultiQuest Corporation, USA
Release:	2.0
Platforms:	MS-Windows, Macintosh, Sun SPARC, HP 9000
Cost:	From $495
Contact:	72531.2510@compuserve.com (708) 397 9930 tel (708) 397 9931 fax

What is it?

As you might infer from its name, S-CASE is a design tool. Specifically, it is an OO design tool using the Booch notation that generates C++ source code. I bought S-CASE after seeing a comparative review in C++ Report, January 1995. The alternatives were Rational's Rose and Together/C++ (see *Overload 6*, page 39) at $1995 and $995 respectively and they were simply too expensive for me to consider as a personal purchase.

What does it do?

S-CASE allows you to design your software graphically by specifying the different relationships between objects. You can optionally describe "scenarios" showing events and messages passing between the objects. Finally and, for most people, more importantly, you can generate C++ code which you then edit to flesh out the methods. Although it cannot take existing C++ code and reverse engineer it, once you have gen-

erated code, you can edit that and the code generator stays in synch, with some limitations.

S-CASE is clearly aimed at multi-user development and supports "projects" with check-in/-out. Each user can check-out part, or all, of a project and work on that part, although this relies on some sort of network file sharing with record locking to provide the necessary security (e.g., NFS). Once checked-out, diagrams can be edited and code generated and/or modified. For anyone who has worked with a source code control system, such as **rcs**, this is second nature. The projects are organised in hierarchies and classes can be 'linked' between sub-projects so that different views of the design can be maintained. This is ideal for designing a system composed of related hierarchies of objects, e.g., a parser will have hierarchies of *Type* classes, *Expression* classes and *Statement* classes that are related by use. S-CASE makes it easy to work in different views while it coordinates the design-level and code-level changes across all views.

Getting started

Installation is straightforward but the licensing is a bit of a nuisance. S-CASE operates in 'demo' mode until you obtain a licence key from MultiQuest. If you have email, this is a relatively painless process but otherwise involves a transatlantic phone call. However, I found the technical support, by both email and phone, to be courteous and efficient so I can't really complain. The manual is well organised, starting with installation (for each platform) and configuration, leading through the online tutorial and then on to the project manager, the class diagrams and finally the code generation subsystem.

The online tutorial is enough to allow you to do useful work with the product but familiarity with Booch's book is necessary to cope with the subtleties of some parts of the notation. Having said that, I had produced several pages of annotated design documentation and code for a small project I had chosen within a day.

Code generation

Initial code generation is simply a matter of selecting some classes from a diagram and telling S-CASE to generate headers and source files. The annotations used in the specification dialogs are written into the code as comments. You can then edit the bodies of the methods to complete the functionality and S-CASE will retain your code during the next phase of code generation.

It does this by embedding special comments in the generated code which you must not remove, although you can generate 'clean' code at any time that does not contain these comments. Because of this strategy, you must be careful not to change the interface of, or relationships between, classes. That means: don't add methods or data members, and don't change the inheritance structure! This 'trains' you to work with the diagrams and so you tend to think more carefully about such changes.

The only problems I encountered were with instantiated classes. Parameterised classes translate to templates as expected, but I could not find an easy way to persuade S-CASE to generate sensible references to instantiated classes (uses of templates). The documentation is somewhat sparse in this area so I suspect this is a fairly recent addition that will improve over time.

Annotations and other information

Specifications of data members and methods are entered through a hierarchy of dialog boxes. This takes a bit of getting used to because you can only view one member or method at a time at the most detailed level. You can always generate the code and look at that since it contains all the information entered in these dialogs but it would be more convenient to have a scrolling list of the information in the dialogs.

Booch-style 'notes' can be added freestyle to the diagrams which allows the class and object diagrams to be used as standalone documentation. Unfortunately, these notes do not get written through to the generated code.

Support for the methodology

S-CASE supports class diagrams and object diagrams with the full range of icons for classes, relationships and messages. S-CASE does not support the other Booch diagrams, such as state transition and interaction diagrams. Given the code generation facilities, the lack of the latter diagrams does not seem, to me at least, any great loss.

CASE for the unCASEwise

I've never been a great fan of CASE tools or formal methodologies but after using S-CASE to document an existing C++ project, I soon found

that 'obvious' design flaws were present in the project. These flaws are 'obvious' once you have a notation other than code to work with and I am keen to use S-CASE for future development where possible. That's not to say that CASE tools stop you making such design errors, but they are likely to be apparent earlier in the process, and probably easier to correct (because inheritance and other relationships can be changed with a few mouse clicks and the code regenerated).[2]

Download a demo

S-CASE can be downloaded from `ftp://ftp.netcom.com/pub/sh/showcase` – "try before you buy!" which is what I did.

Summary

For the price, S-CASE provides a reasonable level of functionality that will be suitable for many C++ shops wanting to take their first steps along the OO CASE path. Working with the diagrams is quick and intuitive but the specification dialogs are a bit clumsy – I hope these will be improved in future releases.

Sean A. Corfield
sean@corf.demon.co.uk

[2] Possibly followed by some tedious editing to get the code to recompile. :-)

Credits

Founding Editor

Mike Toms
miketoms@calladin.demon.co.uk

Managing Editor

Sean A. Corfield
13 Derwent Close, Cove
Farnborough, Hants, GU14 0JT
sean@corf.demon.co.uk

Production Editor

Alan Lenton
yeti@feddev.demon.co.uk

Advertising

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

Subscriptions

Dr Pippa Hennessy
c/o 11 Foxhill Road
Reading, Berks, RG1 5QS
pippa@octopull.demon.co.uk

Distribution

Mark Radford
mradford@devel.ds.ccngroup.com

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 10* (October) must be submitted to the editor by September 4th.