ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 7

April 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 3

The AGM 3

Future directions 3

Submissions 3

The Overload disk 3

Software Development in C++ 4

C++ compilers – mainly for OS/2 4

No such thing as a free lunch 7

Subsidising lunch? – a reply 10

Operators – an overloaded menace 12

Overloading on const and other stories 15

operator= and const – a reply 18

The Draft International C++ Standard 18

The Casting Vote 18

C++ Techniques 22

Wait for me! – copying and assignment 23

Related objects 24

Related addendum 28

Multiple inheritance in C++ – part I 28

On not mixing it... 32

editor << letters; 35

Questions & Answers 40

++puzzle; 41

Books and Journals 42

Coming soon! 42

The C++ Report 42

News & Product Releases 42

Programming Research to distribute TestView 42

NoBUG 43

Editorial

Thankyou to everyone who has contributed to *Overload 7* – I have received more material than I can publish so please keep it up! The success of *Overload* depends on each and every one of you.

The AGM

Yesterday (as I write this) was my first attendance at an ACCU AGM. It was good to put faces to so many contributors and committee members and was a lot of fun. The programming competition was let down somewhat by the failure of several teams to attend but Acorn and IBM put on a good show with some completely over the top hardware (Acorn) and some very entertaining MPEGs (IBM). My favourite moment was Acorn's team saying that if they could just get their program to compile, it would work wonderfully! "Yah boo sucks!" to the other vendors – I hope you redeem yourselves next year!

Future directions

The most common question I was asked at the AGM was "Where is *Overload* going?" Well, that's not true – the most common questions I was asked were "When are you lot going to stop messing around with the C++ standard?" and "Don't you think C++ is too big and complex for the average programmer?" Several articles in this issue address both of those questions but back to *Overload*...

My stock answer was that it depends on what *you* write for *Overload*. Quite a few people wanted to see more material for the beginner or intermediate but a similar number wanted to see more advanced material. I don't think there'll be any shortage of the latter but I must appeal to all of you to consider the former. What I'd like to see in that vein are articles about your first C++ project: what went right, what went wrong and why, did C++ make it easier or harder, how different was it to using C (or whatever). If you feel that your company might not appreciate such candour, I will withhold names on request (but I will not accept anonymous submissions!).

Submissions

At some point, I'm going to run a survey to find out what percentage of you are using which platforms and how many of you have email. The submissions that I've received for this issue suggest that most of you use IBM-compatibles and most of you have email.

I do not use an IBM-compatible, although I have, with some effort, managed to read everything submitted so far. Some of the material has been very carefully formatted and looks great when it is printed but remember that *Overload* will, in general, be laid out by Alan Lenton so, to some extent, your careful formatting will be to no avail. Accordingly, my preferred formats for submissions are: plain ASCII text or RTF. Every WP package should be able to produce one of these. I'm also happy to receive Word 6.0 documents since that's my native format (on a Macintosh).

If you want to include fancy graphics with your article, contact me first to ensure that Alan and I can incorporate them.

Please use email, where possible, for submissions – I am allergic to paper :-) If you want to send compressed files, I have UNIX ***compress*** and ***gzip*** only (I run Tenon Intersystem's MachTen on my PowerBook – a BSD4.3 port). If email is not possible, I will accept soft copy by snail mail. I promise to acknowledge all email submissions – if I reject an article or want it reworked, I'll let you know fairly quickly, otherwise you can assume it will appear in the next issue of *Overload*.

Please note that it is *Overload*'s policy to print email addresses unless you explicitly request otherwise.

The Overload disk

In *Overload 6*, Francis asked what you thought we should do about the supplementary disk that appeared with previous issues of *Overload*. Several people have responded that they would prefer the code to be made available on Demon. Accordingly, there will be no disk with future issues of *Overload* and I will arrange for code,

documentation and other interesting material to be placed on Demon for anonymous ***ftp***. Full details will be in *Overload 8*. For those of you without ***ftp*** facilities, this would be a service that the Software Librarian could provide – if someone volunteers for the position!

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, our survey of C++ compilers continues and several contributors make critical comments about the cost and complexity of using C++. I have held the second instalment of my compiler-writing series over to *Overload 8* for various reasons that will, I hope, become clear in that issue!

C++ compilers – mainly for OS/2

by Francis Glassborow

Before I tackle the primary subject of OS/2 C/C++ compilers I'd like to take a little space to expand on my column in the last issue.

For some reason I completely forgot to mention the most outstanding feature of Salford Software's C and C++ compilers – their debug support and in particular, runtime debugging. Anyone who has written more than the most trivial of programs will have tripped over memory problems (though they may not realise it yet).

Memory problems

There are four major categories of memory problem:

1. Dangling pointers (and references in C++), i.e., using a pointer variable that is no longer attached to underlying memory. For example, returning a pointer or reference to a local (auto) object. Often such abuse actually appears to work because the associated memory has not been reused yet. This actually makes matters worse because the defect will only manifest rarely, and will get past many programmer contrived test suites.
2. Writing beyond the end of an object – or sometimes before the beginning but this is a far less frequent problem. C's mechanisms for handling array parameters (and dynamic arrays) make this problem particularly vicious and frequently impossible (or effectively so) to detect statically (at compile time).
3. Memory leaks – the commonest form of resource leakage. This is another problem that is difficult or impossible to detect statically and which needs special tools to detect dynamically. In a way, it is the exact reverse of 'dangling pointers' because it occurs when all pointers and references to dynamically assigned memory are lost before that memory is freed. The real sting in this problem is that it often only manifests as a serious problem when a program has been running for hours, days or possibly weeks. Virtual memory resources make it worse by delaying ultimate collapse.
4. Reading uninitialised memory. Any attempt to read from memory that your program has not previously written to will exhibit undefined behaviour. Unfortunately, undefined behaviour often manifests by doing exactly what you expected. That makes it rather difficult to detect.

I recently had an instance in my training room where a programmer was puzzled because his program always ran correctly the first time and failed the second. The first time the program ran, his assumption that a variable was zero agreed with the memory provided. On running the program again he got the same storage with the results he had written to it during the first execution.

Clever operating systems such as Windows NT make this kind of problem harder to detect because they clean up storage before re-allocating it to a new task.

I believe NT stamps 0xDEADBEEF all over freed memory? Clearly not a vegetarian operating system... – Ed.

There are tools available to tackle these problems and I hope readers will write in to describe

anything that helps them reduce the incidence of these problems.

Salford Software C & C++

The item that I completely forgot to mention in my notes last time was that these compilers provide special support for detecting all the above categories of memory problem. This is not the place to go into details except to say that it is probably the cheapest tool for this kind of debugging, certainly much cheaper than such commercial products as 'Purify'.

Symantec C++ 7.0

Note that this release requires substantial resources including 16 Mbytes of RAM (not the 8 that EXE magazine mentions). The full requirements as specified by Symantec are in the March issue of C Vu.

I was being a bit optimistic when I wrote my last column but my latest information is that version 7 will ship on March 25th (I guess those who had the sense to get to our AGM may already know that).

...if Symantec's team had turned up! – Ed.

The upgrade price is £89 and the special offer price (until the end of May) is £149 – at these prices it must be worth considering upgrading your machine to 16 Mbytes. The way things are going you are likely to need that much for sensible performance with the next generation of OS's and development tools anyway.

I mentioned last time that the parser had been disconnected from the rest of the compiler. I can now be more precise and tell you that it has been wired into the editor so that your code is being parsed while you write it. This is not intrusive in that it lets you write whatever you want to and ignores what it does not understand. May be we could persuade Symantec to provide an option which was a bit more intrusive (i.e., howls when you write code that will not parse).

The environment (IDE) is one of the best that I have used, though it will take those of you used to the cruder early 1990's PC IDE's some time to get to grips with it. Those coming from powerful workstation environments may wonder what is so special.

The MSWindows support is based on MFC so at least you will have a few familiar bugs and defects to work round. By the way, if your code is in any way critical you should only use MFC (and code generated using it) if you are familiar with the details of MFC. It is far too late to discover an MFC problem when running mission critical code. I, like many others, can live with the bugs in such Microsoft products as Word for Windows 6 because I do not multi-task critical software along side it.

Those of you who think you can just install a product and jump in to using it at once will find this a difficult product but that is because such attitudes are unrealistic. If you want something better you must expect to invest some effort in learning to use the new product.

Compilers for other OS's

Before tackling OS/2 compilers, a few words about all the others available. What I need from you, the readers, is short descriptions of the compilers you use in VMS, UNIX etc. I do not have the hardware to tackle most of these and even where I do, I lack the time to master yet more operating systems before looking at the relevant development tools.

There is an exception to this and this is GNU C and G++. Sean added a comment about UNIX users expecting these to be free. This is fundamentally true but – and in a PC world it is a big but – you will still have to get a copy as well as copies of all the other tools you will need such as debuggers, profilers etc. The cost of this in a UNIX context (remembering that Unix was designed with programmers in mind) is very low. In addition the tools will work just about straight out of the box (well for Unix gurus it will).

The cost in a PC environment is quite different. Here we expect the basic commercial tools to cost a few (very few) hundred pounds. For the novice, the tools must work directly without any fiddling. It was in this context that I was suggesting that the GNU development tools were not suitable and not that low cost, the actual delivery of the free software will cost close to the price of a low end PC C/C++ IDE such as Turbo C++.

I would still argue that the entire GNU development environment, debuggers and all, is free. However, Francis' point is well taken – GNU software does not always run "out of the box" and can therefore prove expensive to get running – Ed.

How about one of the Linux specialists writing a series on using G++ with Linux. Such a series could be at one of two levels. That for experienced UNIX users and professional programmers could focus on quality programming and tool support. On the other hand there is a place for a series for inexperienced UNIX users and part time programmers aiming at leading the reader from the start. The former would seem appropriate for publication here while the latter would, I think, better fit C Vu.

It would also be nice to see the Macintosh specialists report on the compilers available for their system. I find people often assume that everyone else will know as much as they do about what is available. It isn't true others know more, less and the same but different.

Anyone out there use Symantec, MPW or Code Warrior on the Mac? Write it up and send it to Francis to collate! – Ed.

C++ for OS/2

I wonder how many of you realised that there was some sort of order (though not a complete ordering) to the list I presented last time? Well I'm using similar criteria this time.

Free Software Foundation G++

Let me be entirely truthful about this product; I have never used it. I assume it must exist because I cannot believe that it does not. If any reader has used it, would they write in about their experience of it. I would be particularly interested in any support given to the OS/2 GUIs.

Metaware High C/C++

The problem I have with this one is that Metaware with extreme promptness shipped the wrong compiler across the Atlantic. They sent a Windows NT version, which is fine and when I get to do a round-up of Windows NT tools I'll have some relevant experience but in the meantime I can only make general comments about their compilers.

One point that is well worth keeping in mind is that their is a strong relationship between Metaware and IBM. Metaware wrote the SOM compiler for IBM and also provide a direct C++ to SOM compilation system.

What is SOM? Well that is a little complicated to answer in the current context but I'll give a brief (and I hope not too inaccurate) answer. One of the growth areas in current computing is DLLs and forms of object linking. The problem from the C++ point of view is that any change in a class declaration changes the object module so that relinking is often not enough. This is particularly problematical when your program utilises a DLL. If the DLL version does not have the same layout for classes that your code expects there will be a horrible crunch.

SOM tackles this problem by providing an extra layer of indirection in a language independent way. This means that for a relatively small overhead (less than 15%) in performance your program can use both current and future versions of other SOM conforming software.

The real fun starts as we move into distributed systems and support via DSOM.

Er, yes, but what does SOM actually stand for? – Ed.

Watcom C++ 10

The major advantage of this product is that you get the OS/2 version along with the MSDOS / Windows / Windows NT varieties.

As you would expect from a high quality compiler specialist, this is an excellent compiler. The IDE is pretty rudimentary, which is less significant for those who already have OS/2 development tools from which they can, to a large extent, build their own IDE.

I wish Watcom would go out and negotiate with companies such as Blue Sky and Kaseworks. Add products from these companies to Watcom compiler technology and you have something really special. The problem is that full products from these companies are expensive to buy for any but the specialist developer. Once you have tried special versions attached to a compiler you are likely to want the full product if your work merits it.

If you need to support more than one platform on an Intel x86 based machine this is a compiler you should consider very seriously.

Borland C++ 2.0 for OS/2

This is the only compiler product that I know of that supports both OS/2 GUIs and Microsoft ones.

Watcom, above, not withstanding? – Ed.

With this release Borland includes OWL for OS/2. This is not a perfect match for OWL for MSWindows but it a pretty good one. The product is well up to Borland's normal standard.

The down side is that it is a separate set of tools at a separate purchase price. What we really need is an x86 platform developers CD with both these tools and the MSWindows ones together.

In the meantime, if you need to develop for both Microsoft and IBM GUIs on an Intel x86 platform this has got to be worth serious consideration. The pity is that other priorities at both Borland and Novell (I think they are still responsible) have delayed the development of OWL for Appware.

IBM C Set ++ 2.01

This is IBM's package of development tools. It is the latest release version though by the time this is published we will not be that far from the next release.

As always with products from IBM this is a solid well constructed product. I don't mean that it is entirely bug free – I don't think that there are any products of this complexity for which you can say that. However if your code does not behave the way you expect the chances are pretty high that your expectations were wrong.

Of course, with a language still under development and refinement it may be that you know about the current state of the language while this compiler is still implementing the 1992 version but even the best of firms has this kind of problem.

The development environment is among the best that I have used and the bundled KASE:Set from Kaseworks puts all the other code generators for AFXs to shame.

If you program solely for IBM platforms (OS/2 etc.) then by all means look at the other products but this is the one that you will buy. I can hardly wait to get my hands on the next version.

Conclusion

Well that's my lot. If you want to know about other compilers you will have to hope that your fellow members will send me reports to collate and publish in future issues.

I hope you can understand why I get so irritated by those who ask me what is the best C++ compiler. There is no such thing and anyone who tries to give you an answer without first checking what you want to do is too ignorant to be worth listening to.

People who answer questions without asking any of their own are unlikely to provide useful answers.

Francis Glassborow

francis@robinton.demon.co.uk

No such thing as a free lunch

by Alan Griffiths

Introduction

C++ is a wonderfully expressive language but it places stringent demands upon the developer's competence. In doing this it imposes a cost on any development using C++ which has to be balanced against the benefits offered by the capabilities of the language. Expressive power and skill are often linked – a violin is harder to play than a Stylophone but can, in the hands of a virtuoso, produce music that is in a different class. However some of the difficulties associated with C++ are not caused by its capabilities, they are caused by the way in which the language has evolved. In particular: the need for compatibility with the past has brought such baggage as the C declaration syntax; while the "don't pay for things unless they're used" principle has led to such costly default options as static linkage of member functions.

I have used a wide range of programming languages over the last twenty years; C++ is unique both in the facilities it offers and in the continuing effort required to use it competently. I don't mind the effort needed to use the expressive power of the language but the effort required to circumvent soluble problems is a continual irritation. In short C++ programming is not only hard, but also harder than it needs to be.

I am not saying that programming in C++ is wrong; far from it – I frequently need its power

of expression, but this power often comes at an excessive cost. It takes considerable practice on the violin to play a tune (I can't), but anyone can play one on a Stylophone (at least I can). The other difference is that there are many more ways to play the tune – the results may be much better but the cost is higher. It is always necessary to consider the costs and C++ is pricing itself out of the market. If I have a program to be written and a choice of a trainee programmer and Visual Basic for a couple of weeks or an experienced C++ programmer for a couple of weeks (or an inexperienced one for a few months) which route am I going to take? The Visual Basic program may not be as elegant or efficient, but it is far cheaper.

Having just made some claims about the unnecessary cost of using C++ I should come up with some justifications! A continual problem for me is the unhelpful defaults of many features of the language, for instance:

- member functions don't default to virtual;
- default constructors, copy constructors, and assignment operators are generated automatically.

Other problems for the developer are caused by:

- the lack of a syntax for referring to classes by their relationships ("my base class"),
- with the addition of "exception handling" C++ is no longer a "better C", and
- constraints on the program that cannot be checked automatically (e.g., the "one definition rule").

Allow me to elucidate...

Non-virtual default for member functions

The static linkage of member functions (and destructors) is really an optimisation, and any optimisation choices really belong to the latter stages of the development cycle (that is not as a cost throughout the whole of program development). If member functions were declared "virtual" by default then, when it becomes apparent that a function needs to be overridden by a derived class, there would be no need to amend the original class and recompile it and everything that references the class declaration.

The default is "justified" on the basis that the overhead of a virtual function call is avoided except where explicitly requested. However, I cannot believe that the cost of dynamic binding is significant in the majority of cases. In speed terms suppose that dynamic binding adds 20% to the function call overhead and 10% of the programs execution time is spent in the function call overhead – this is almost certainly an overestimate and still only gives a 2% performance hit. Of more relevance are small classes that have large numbers of instances. These may not be able to stand the overhead of a vtable reference in the memory mapping of the class.

Before anyone writes in and tells me that I should just put **virtual** before almost all member function declarations let me point out that this is my argument. It is the need to know this is desirable and the time spent overriding the language default that are unnecessary costs.

In addition, (and this is common to a number of the other points) it is impossible to override the defaults in library code that is outside my control. To cite a particular example of a problem library: there are a number of classes in the MFC library that should (allegedly :-) have virtual destructors but don't. If the default were "correct" then this would be very unlikely to have happened. It is not just Microsoft that make this error – it is also a problem with the current draft of the proposed "Standard Library".

The "big three"

There are many classes for which the automatic generation of the "big three" (the default constructor, the copy constructor, and the copy assignment operator) is a positive menace. If, for example, a pointer to dynamic memory is not initialised (generated default constructor), or is "bit copied" (generated copy constructor or assignment operator) and then "deleted" in the destructor, then memory management is compromised and there are no guarantees of subsequent program behaviour.

The committee recently clarified that the generated copy constructor and copy assignment operator perform memberwise copy and memberwise assignment respectively. Such copying or assigning of an uninitialised value causes undefined behaviour so you may not even get to your destructor – Ed.

Any class that manages a resource needs to declare the “big three” to avoid problems. Of course to change the language to prevent automatic generation for classes which contain pointers (or member/base classes without the corresponding functions) leads to a problem about how to code copy constructors and assignment operators.

Naturally, tools like “lint” can be used to check for these functions (and some of the other problems mentioned). However, the need for such aids complicates the development process and (as mentioned above) does not help if it is library code in error.

Referring to related classes

Coding copy constructors and assignment operators “by hand” is difficult because there is no syntax for navigating the network of base classes. The lack of a syntax for “base class of this class” also leads to problems with maintaining inheritance trees in cases where derived classes supplement the behaviour of virtual functions by explicitly calling the corresponding function in the base class.

It would be nice to say, for instance, “the direct base class with this function”, but instead one must identify the specific base class whose member function is to be called and hope that anyone adding a class between them in the inheritance graph updates the reference. C++ would be simpler to use if this process were automated. (Of course, if one gets the design right first time...)

No longer a better C

For a large part of its development period it has been possible to treat C++ as “a better C”, which provides a pool of programming resources. Although ex-C programmers may not produce ideal C++, they could be productive and be gently introduced to C++ programming constructs during the course of a development. (One such programmer, after a few days spent coding some functions with “C++” names such as ***AClass::AClass*** and ***AClass::method*** was asking how one went about writing a class. He took some convincing that he had already written most of one.)

The advent of “exception handling” changed all that. This flow control mechanism affects every piece of code and needs to be understood by the programmer. As indicated above it is possible to produce correct code without a clear understanding of the “class” mechanism. However, a lack of understanding of “exception handling” is far too likely to lead to problem code like the following:

```
void f()
{
	char* buf1 = new char[100];
	char* buf2 = new char[100];

	if (buf1 && buf2)
	{
		// Something
	}

	delete [] buf2;
	delete [] buf1;
}
```

This is now badly broken – if an exception is thrown anywhere between initialising ***buf1*** and deleting it, then the memory that it references will “leak”. Of course, on many platforms losing a few bytes like this may not be an issue, but the same problem exists with more complex objects and other types of resource.

Some other languages that use exception handling also include “garbage collection” which trades these problems for another, more intractable set (when you find you have insufficient control over the “garbage collection” process you have no options). In C++ the code can be fixed (below) but the style seems less natural to those moving from C or early C++ implementations:

```
void g()
{
	char* buf1 = NULL;
	char* buf2 = NULL;
	try
	{
		buf1 = new char[100];
		buf2 = new char[100];

		if (buf1 && buf2)
		{
			// Something
		}

		delete [] buf2;
		delete [] buf1;
	}
	catch (...)
	{
		delete [] buf2;
		delete [] buf1;

		throw;
	}
}
```

Naturally, this is not the only solution, but unless you wish to obscure meaning by avoiding the direct use of pointers in this type of code then the alternatives are equally long winded.

The "One Definition Rule"

I'm not sure of the current phrasing of the "One Definition Rule" – the draft Standard makes it clear that "clarification" is taking place. It says something to the effect that there may only be one definition of any entity within a program, and if not the behaviour of the program is undefined. It also adds the helpful information that the development environment need not offer any diagnostic message.

This means that if both you and the developer of a library you are using decide to define the same "entity" then there need be no diagnostic and the program could do anything! Just imagine what trying to police such a requirement without diagnostic aids does to your development costs.

In conclusion

As I said at the beginning, "C++ is a wonderfully expressive language" – it is; it allows a wider range of programming idioms and algorithms than any other language that I've encountered. The downside of C++ is the need for a much higher level of competence in using it. If C++ had a different history, or there were less focus on "don't break existing code" these problems could be addressed.

At the time of writing the language standardisation process has reached a stage where the chance of fixing any of these problems is remote. The cost will now fall on the developer.

Alan Griffiths

alan@octopull.demon.co.uk

Subsidising lunch? – a reply

by Sean A. Corfield

First of all, let me say that I think Alan makes an excellent point about the demands that C++ places on developers. There is no doubt that the learning curve for a language as complex as C++ is much steeper than for, say, C. It may not be so clear-cut that the benefits are correspondingly higher too and so I shall not attempt to argue that point. I shall, however, put on my compiler-writer / X3J16 hat and respond to several of Alan's more specific points.

A non-virtual cost

Alan argues against the non-virtual default for member functions and estimates a 2% performance penalty for using virtual everywhere instead. Typically, 1 in 5 instructions in generated code are function calls. Even assuming calls are no more expensive that ordinary instructions (and they often are), a program will spend about 20% of its time calling functions. On a particular machine, a function call instruction takes 2 cycles – what overhead does a virtual call add? First, you have to load the address of the vtable from the object, which takes 3 cycles. Then you have to load the address of the function from that table – another 3 cycles. Plus the call. This *quadruples* the cost of the call. If half of all the function calls were virtual, this would add 30% to program execution time. Moving to another machine, the call takes 3 cycles compared to a load (average 6 cycles) and an indirect call (average 10 cycles) – a factor of more than 5 on the call, and an overall factor of 40% on the program. Of course, in these days of faster processors, even factors such as these should not matter too much...

My thanks to Derek Jones for providing typical execution times on two very different architectures – Ed.

As for the draft Standard Library making the mistake of using non-**virtual** destructors – I can't think of any library classes that are intended to be used as base classes, with the exception (sic) of the ***exception*** class hierarchy which does have **virtual** destructors.

Base class names

In one OO-language, you can refer to a base class with the keyword **inherited**. This was proposed for C++ by Dag Brück some years ago (see Stroustrup's Design and Evolution book for details). The proposal was not accepted for two reasons. Firstly, what happens if you have multiple base classes? Secondly, there was already a way to do this within the language:

```
class Derived : public Base
        // #1
{
public:
        typedef Base inherited;     // #2
        void f() { inherited::f(); }
};
```

Admittedly, this suffers from the multiple base class problem too, and if you change #1 without changing #2...

Exceptions break everything

I'd love to be able to argue with Alan on the negative impact of exception handling but, unfortunately, it's even worse than he indicated! Let's look again at the "fixed" version of his example:

```
void g()
{
	char* buf1 = NULL;
	char* buf2 = NULL;
	try
	{
		buf1 = new char[100];
		buf2 = new char[100];

		if (buf1 && buf2)
		{
			// Something
		}

		delete [] buf2;
		delete [] buf1;
	}
	catch (...)
	{
		delete [] buf2;
		delete [] buf1;

		throw;
	}
}
```

Is this fixed? Not quite! What happens if **new** fails? It throws an exception and does not return. In the example above, testing that ***buf1*** and ***buf2*** are not null pointers is redundant. In fact, it makes no difference in the above case but the fact that **new** throws ***bad_alloc*** instead of returning zero will "break" almost every program written before exception handling. One common trick in use today is to add the statement:

```
set_new_handler(0);
```

near the beginning of ***main***() which often sets the behaviour of global **operator new** back to the "old" behaviour. This was not portable and in Austin (March '95) the committee voted to remove this "hack" and provide a standard way to use **new** without having to deal with exceptions – see *The Casting Vote* in this issue for more details.

One solution to this problem is to embrace the "initialisation is resource acquisition" idiom where the "resource", in this case memory, is "acquired" by a constructor and released by the corresponding destructor. The draft Standard Library provides several ways to do this – for the example above, it would be more "natural" to use the ***vector*** template class:

```
void g()
{
	vector<char> buf1(100);
	vector<char> buf2(100);
	// Something
}
```

This does mean, of course, that you need to "know" even more about C++ and its library but the benefits are more maintainable programs since you no longer clutter up functions with error-prone housekeeping code.

Just One Definition?

Alan complains that no diagnostic is required for a violation of the "One Definition Rule" which is a reasonable complaint, but let us look back at C first. The ODR corresponds roughly to the link-model used in C: if you provide more than one definition of a function or object at link-time, it causes undefined behaviour. So we appear to have made no progress over C. Wait a minute though – what about C++'s "type-safe" linkage, you ask? Consider the following:

```
/* file1.c */
int a;
/* file2.c */
void a();
int main()
{
	a();
}
```

On some systems, a C compiler will successfully link this program because it uses only names for linkage, not types. Some systems might give a link-time message – I once saw the very mysterious "too far to jump" message from a linker presented with the above code. Now consider a C++ system: it typically encodes a function's calling sequence into the name. This means that the link-names of ***a*** the variable and ***a*** the function will be different. So C++ has actually helped us here!

My conclusion

At the end of the day, I basically agree with Alan – C++ is harder to use than C – and I think his comparison between a Stylophone and a violin is well-drawn. I don't blame the language (and I don't really think Alan does either) – I blame IT management for giving everyone a violin and saying "right, now play a tune!" What C++ highlights is the need for better training, better tools and more realistic expectations.

Sean A. Corfield

sean@corf.demon.co.uk

Operators – an overloaded menace

by George Wendle

I have received some private comments about George's last column so I feel compelled to explain my position: like Francis for CVu, I do not edit George's column (other than to correct typos) which means he may well be more controversial than you care for – he also might be completely wrong! That is for you, dear reader, to decide. I hope that George's columns will encourage several of you to respond – in the past, a particularly barbed attack on the C++ standards committee (CVu5.6) caused me to write a somewhat outraged response (CVu6.1) – Ed.

I like C++, it has the potential for being a great language but it is also exceptionally complicated almost, I think, to a degree where the designers themselves do not understand the implications of their decisions.

What I would like to see is a concerted effort to simplify the language itself and make it easier to use with predictable results – predictable, that is, to the ordinary working programmer not just to balding whiz kids.

The language designers seem prone to introducing things that make their lives easier, often by allowing compilers to implicitly support something which would otherwise have to be made explicit.

One area that is a minefield of unwanted complexity is that of overloading. What is so wrong about forcing programmers to disambiguate close decisions? Doing so might persuade them to look more carefully at their designs and reconsider the degree to which they overload things. By the way, it would be no bad thing if the designers reversed their habit of overloading new, subtly different, meanings onto keywords like **static**. Actually that keyword is a complete disaster akin to the term chosen for new style function declarations: "prototypes". Both words are already in active use in computer science for other purposes.

Enough of this pre-amble. Let me come to the point of this article – overloading, and specifically operator overloading. Before dealing with the latter let me take a quick look at function overloading.

Function overloading – a harmless convenience

I must admit that I think the idea of function overloading is quite elegant, even if it is generally unnecessary. Bjarne Stroustrup writes in his book "The Design and Evolution of C++" that the idea arose from the need to provide multiple versions of a class's constructor function. There are other solutions to this problem but I agree that function overloading is a 'nice' answer. Once you introduce it for that reason you might as well make it a general facility.

Once you have function overloading you need a method to resolve uses of an overloaded function name. The first part is to collect all the candidates for the decision.

The rule is currently simple (I say currently, because I do not understand namespaces well enough to be sure that it will remain simple in future.)

Start by examining the current scope, remembering that where the call is to a member function – always identifiable because an object or pointer to object will decorate the call – the initial scope is that class's scope.

Search that scope for all declarations of the required identifier, if any are found that is your complete candidate set.

Otherwise repeat the process for each scope containing that scope.

Keep going until you either obtain a candidate set or have failed while searching the global scope.

In the next stage trim the candidate set to those that have the right number of parameters (being careful to leave in appropriate versions of declarations that fit by using default parameters.)

Now look to see if one of the candidates has the types of its parameters exactly matching those of the arguments in the call. If so, use it (if two match at this stage, take the programmer out and shoot him/her – its probably an acne-ridden male teenage, bedroom whiz kid hacker, but to say so would make me guilty of so many -isms that the PC world would put out a contract on me.)

Don't worry George, the PC police do not roam the pages of Overload – Ed.

If not, you will have to go into best fit mode and start playing games with type conversions. This stage needs drastic simplification because the rules are just too fine grained for good sense. It may mean that ambiguity rarely arises, but it also means that sometimes the resolution is not the one that you expected, leaving some subtle defect in your work. I much prefer to have a compiler require me to be more precise than to have it double guess me. Now we have a range of new-style casts, disambiguation through casting an argument is much less dangerous.

The end result is that function overloading is fine. You only have to use it for constructors. If we shout loud enough the granularity of resolution might be coarsened or one of the providers of support tools might provide a tool that would warn of close calls.

Noted :-) – Ed.

Good programmers (usually those whose employers have supported with training and time to develop skills) will use function overloading with care. Bad programmers, well I doubt that anything will make them better (but see my column in CVu7.3).

Operator overloading

I bet you thought this was just a variety of function overloading. You could not be more mistaken. It is completely different, it is in the language for different reasons and it has its own overloading rules. These are so complicated that I am not sure that I fully understand them myself, so feel free to write in tearing the following to shreds. With Sean Corfield as editor I am sure he will act as referee and prevent any actual spilling of blood.

Before we start providing any overloading on operators, the language has a fully defined set of operators, each appropriately overloaded (or not provided if inappropriate) in the context of the built-in types. Whatever mechanism implementors use to support these operators, it is inaccessible.

On the other hand, programmers who wish to overload an operator must do so by providing a function to do the work. Despite the slightly eccentric form of such an operator function, it is a function and is subject to exactly the overloading rules that pertain to other functions.

This can lead to some weird behaviour. Consider the following:

```
void fn()
{
	int i;
	i= 1 + 2; // the RHS will, I
think,
		// be statically
evaluated
		// by the compiler.
	i = operator+ (1,2); // does
what?
}
```

Well that explicit call to the **operator**+ function won't be able to call the normal '+' for **int**s because no such function exists (well it may be an implicit function provided by the compiler implementor – but we cannot use that). Instead it will have to search global scope for any available user provided versions. These certainly will not be for two **int** arguments because the language rules explicitly forbid users providing their own versions for parameter lists that do not include any user defined types.

That rule is, in itself, an error because it prevents users from providing their own mixed mode arithmetic via operators. One of the eccentricities of C++ is the automatic type conversion rules it inherited from C and this rule prevents me from fixing that.

Actually, the language doesn't forbid this – but only when at least one operand is a user-defined type is the full search performed, otherwise only built-in operators are considered – Ed.

Next case. Consider:

```
void fn(){
	MyType m(...);		//initialised
with
			// appropriate
values
	int i;
	i=m+1;			//A
	i=operator + (m, 1);		//B
	i=m.operator + (1); //C
}
```

At line A the compiler first looks in the scope of ***MyType*** to see if I have provided an **operator** + function

If I have, it starts the normal process of overload resolution, but what is the candidate set? Only those in the current scope? Those in the current scope and the built-in ones? Those in the current

scope, built-ins and globals? All those from the current scope outwards through all enclosing scopes to global scope?

If you truly know the answer to this question, I take my hat off to you. I don't. Of one thing I am certain, the normal name hiding rules for nested scopes do not apply to operators. They cannot or else declaring an operator will hide and inhibit the use of all versions in enclosing scopes.

Now suppose that as well as an in-class definition of ***MyType*::operator+(*MyType*)** there is a file scope (or wider) definition of **operator+(*MyType*, int)**. Under what circumstances will this exact match be found? Only if no resolution (however bad) can be found in class? Never (i.e., the in-class version hides the other)? Always?

Suppose that ***MyType*** provides a conversion to ***YourType***. When will versions of **operator+** with ***YourType*** as the left operand be considered?

Now let me turn to line B above (explicit call to **operator+**). I assume that this can only consider versions provided in the scope where it is used or in some outer containing scope. However I have to confess that I am not entirely sure of this.

Whether I am right or wrong, it is certainly the case that the explicit use of an operator function will result in quite different overload rules from those that are used when I use the operator itself.

Obviously line C only searches within the scope of ***MyType*** and its enclosing scopes. Obviously? What about the case where ***MyType*** contains an **operator *YourType*()** function? Of course you already know the rules for this situation. You do, don't you? Oh, well, perhaps I over simplified the rules for overloading functions, or did I?

Questions, questions, everywhere a question

Have you noticed how many questions I have asked above? Some I know the answer to, some I don't, but my knowledge is irrelevant. The important thing is how much can we expect from the competent programmer like yourself. I bet if I gave my questions to two C++ experts I would get two sets of answers that differed in at least one instance.

Even those that can answer all the above consistently may find that they are not so sure when we throw **namespace** and templates into the mixture. When we get operators defined in template classes, or worse still get offered template operator functions, we really do need a very clear understanding of the overloading rules for both functions and for operators.

Conclusion

In the meantime I think we should all be very wary of overloading operators. I think we can just about live with the provision of in-class operators as long as they really do represent the natural expectations of naive users of that class.

On the other hand, I think that any global provision of operators is highly dangerous. Frankly, I would like to see producers of class libraries completely avoid the provision of out-of-class operators. If they must provide them, please do so by providing the functionality in-class and wrapping it up in an inline function (see Francis Glassborow's article in *Overload 6*). Such inline operator functions should be in a separate header file so that the user determines their availability not the library provider.

The rules for operator overloading need to be cleaned up and made comprehensible to mere mortals such as I, until they are the best advice is 'do not use them, they will introduce unexpected behaviour into your work and that of your clients'.

Finally, could our new editor (congratulations on your first issue) either write a detailed explanation of overloading or commission some other expert to do so. I guess it might even take several issues.

George Wendle

Thankyou George. A detailed explanation of overloading would be very likely to fill several issues of Overload! Perhaps I'll take up the challenge after I finish my cOOmpiler series or maybe I can persuade someone else to write a series on overloading? Just to add more spice to the issue, the Standards committee have been making changes to operator name lookup too – see my Casting Vote column in this issue – Ed.

Overloading on const and other stories

by Kevlin Henney

I read George Wendle's article "Overloading on **const** is wrong" in *Overload 6* with great interest. I have always been a keen advocate of **const** and the idea of **const**-correctness in code: it permits the visible expression of certain design level decisions in code for the benefit of both the compiler and the human. So where should we draw the line: why should some member functions be **const** and not others, what are the exceptions to the rule, and would you like biscuits with your **const**?

*I thought that was "would you like fries with your **const**?" – Ed.*

Sean also raised an issue in reply to an old letter of mine. Why should the assignment operator return a non-**const** reference to its left hand operand?

Overloading on const

George cited a few examples where overloading on **const** arguments appeared to be a bad idea. The only problem I have with these is that they did not appear to be real examples:

```
void fn(D&);
void fn(const D&);
```

Looking over my code, I only ever use **const** overloading in the context of a class and I have been unable to find any functions overloaded on **const** that do not differ in either return type or argument count. Clearly something interesting, and hopefully useful, is going on if I feel the accessibility of the current object should dictate the result type. George cites the classic example of **operator[]**. Providing a subscript operator for a ***vector***, ***string*** or ***map*** class is practically a fundamental requirement:

```
string motd = "hello";
motd[0] = 'j';
```

What such an operator must also ensure is the preservation of **const**-ness. Consider a ***string*** class with only one subscript operator:

```
char& operator[](size_t) const;
```

If it did not return a reference, the change to ***motd*** above would not be possible. However, not declaring it **const** would actually prevent routines passed references to **const** strings from reading through the string character at a time. There is a problem with this one size fits all approach:

```
const string greeting = "hey";
greeting[2] = 'p';
```

This is legal, but is clearly a violation of the expected semantics. The solution is to overload on **const**-ness to determine the level of access the user should have:

```
char operator[](size_t) const;
char& operator[](size_t);
```

Beyond subscription

If this were the only example of this technique I might be inclined to agree with George that it is an exception and should be catered for separately, but it is not. This example outlines a general principle related to member access. In search of concrete examples you need go no further than the STL[1]. Each container may be iterated over. An iterator is defined to have pointer-like semantics and may be initialised to the beginning of a container, its end or to the result of a search.

One problem that has previously caused problems with iterator classes is that they often fail to preserve the **const**-ness of what they are iterating over, i.e., through an iterator I can gain writable access to **const** objects. Alternatively, the iterator provides only lowest common denominator access — but it is frustrating being given read-only access to a writable object! The STL addresses this problem in a disarmingly simple manner by requiring both **const** and non-**const** iterators. For example, for access from the first element a container class would include the declarations

```
iterator       begin();
const_iterator begin() const;
```

Overloading should only be used to give similar concepts similar names, and this is clearly the case here. Suggesting that the **const** version should be renamed ***begin_const*** breaks with this, causing the programmer to do the name mangling instead of the compiler.

---

[1]The Standard Template Library is a collection (sic) of container classes and algorithms that has been accepted by the ISO committee as part of the C++ standard library. A copy of the documentation and a sample implementation, by the original authors of the STL, was on the disk with *Overload 5*.

Same thing, only different

All the discussion so far has centred on function pairs that differ in return value but are behaviourally identical. There are a few cases where the semantics and mechanism can also differ. An example of this is a *create-on-demand* awk-like array for which the non-**const** subscript operator creates the indexed element with a default value if it does not already exist. The **const** version would throw an exception:

```
const Type& operator[](const Key&) const
      throw(out_of_range);
Type& operator[](const Key&);
```

On the whole, behavioural differences between **const** and non-**const** versions of an overloaded pair should be either non-existent or minimal.

> *I agree and the STL gets around this by simply not defining a non-**const** version of the subscript operator for **map** (STL's associative array template class) – Ed.*

However, there is an example I feel would be useful that breaks with this requirement. One of the few areas that the C standard I/O library wins out over its C++ counterpart is pattern matching on input. As its name suggests, the ***scanf*** function implements a simple generic scanner, albeit a somewhat insecure and idiosyncratic one. Taking advantage of the difference between non-**const** references and **const** references or values it is not hard to imagine an equivalent facility for C++:

```
cin >> day >> '/' >> month >> '/' >>
year;
```

For such a scheme to work well, the type of literal strings would have to be **const char*** rather than **char***. Sean made a proposal to rid C++ of this irksome piece of C heritage; sadly it was not accepted by the powers that be.

> *And I haven't yet discovered why the Core WG did not adopt this proposal – Ed.*

The functionality described could be implemented using manipulators (see "Writing your own stream manipulators", *Overload 5*):

```
cin >> day >> match('/') >> month >>
             match('/') >> year;
```

These could take advantage of templates and template specialisation. However, I do not believe there are any proposals to standardise such a cluster of classes and it would be good to have a simple version already in place that echoed the versatility of ***scanf*** in softer, safer tones. Perhaps **const**-ness in C++ has not been taken far enough?

Back on the chain gang

Method chaining, also known as cascading, is a useful technique for grouping a sequence of related operations together in a single statement. The result of a function, that would otherwise be **void**, can be used for further operations on the object of interest. A primitive form of this is available with many of the C string functions. In C++ the most conspicuous example of chaining is in the I/O library:

```
cout << "The temperature at " << time
     << " on " << date
     << " is " << temperature
     << '.' << endl;
```

The result of each call to **operator<<** is a reference to the ***ostream*** that was used for output. Chaining is also present in the C language itself; it is not just restricted to the library:

```
a = b = c;
```

The result of each assignment is a modifiable *lvalue* of the left hand side and not a copy of that value.

> *Only in C++ I'm afraid! In C, the result of an assignment is not an lvalue – Ed.*

The proposed standard library, and much of my own code, follows this idiom. Non-**const** member functions that might otherwise return **void** often return ***this**.

```
coord.radius(new_r).radians(new_theta);
motd.assign(subject).append(" is ")
       .append(opinion);
dir_list.sort().reverse();
```

The last example is, for some reason, currently not possible with the STL. It appears to be an oversight that hopefully will be rectified by the library committee: first, it is clearly useful; second, it is important that all library components are written to a common style which, in this case, is that of chainability.

Assigns and wonders

All well and good, but what about the assignment operator? This is the issue that Sean raised in response to my criticism of one recommendation in the Ellemtel *Programming in C++: Rules and Recommendations* document (included on

the disk that came with *Overload 4*). The discussion above suggests that because the assignment operator is a non-**const** member function it should return a non-**const** reference to the assignee. Many other sources support this view:

- The definition of assignment for the built-in types;
- Compiler generated assignment operators return a non-**const** reference;
- Assignment operators in the fledgling C++ standard library return non-**const** references;
- Many of the good authors in the C++ community support this as a standard idiom (e.g., Stroustrup, Coplien, Meyers, etc.).

These are, to say the least, quite persuasive reasons. This is clearly standard form, yet the Ellemtel guide suggests that returning a **const** reference is better. To probe this decision we must better understand what coding rules and recommendations might help us to achieve:

1. readability, e.g., indentation, identifier names;
2. defined-ness, e.g., the result of ***a*[*i*++] = *i*++** is not well defined;
3. security, e.g., use of ***gets*** can seriously affect the health of your program;
4. insurance against accident, e.g., declaring without definition a private copy constructor and assignment operator prevents accidental copying of certain classes of objects;
5. conformance to expectation, i.e., preservation of the *Principle of least astonishment*;
6. interoperability, i.e., the ability to mix with other components written to a standard form.

In other words, rules and recommendations are a response to, and a preventative cure for, possible problems. What are the problems that the Ellemtel guide is trying to lay to rest? Unfortunately only one example is given:

```
(a = b) = c;
```

This is a pointless and pathological piece of code, but how does it measure up against the criteria for a problem seeking a solution:

1. This is quite readable — pointless, yes, but with parentheses forcing the precedence it is easy to see what is going on. Indeed, it might be argued that the chained assignment without parentheses offers more scope for confusion.
2. This is well defined: ***a*** is assigned the value of ***b***, and then ***a*** is overwritten by an assignment from ***c***. Again, pointless, but certainly well defined.
3. It is also secure — no problems with dangling pointers, corrupting memory, etc.
4. You have to force the precedence to get this code fragment, so such code is unlikely to be produced by accident. I don’t know about you, but my typos are normally quite simple: I have yet to accidentally enclose a well formed expression with balanced parentheses — and not notice!
5. In the light of what I mentioned earlier I would expect this example to compile cleanly.
6. If ***a***, ***b*** and ***c*** are iterators or containers, this code conforms to the signature requirements for assignment laid out by the STL for containers and assignable iterators.

The only problem I was able make out was that the authors of the guide were uncomfortable with C and C++! If they wish to break a *de facto* (bordering on *de jure*) standard, they will have to do better than one contrived and weak example. By this, I do not mean that many weak and contrived examples will strengthen their case ;-)

The Ellemtel guide even states, inadvertently, why you should ignore their recommendation:

> *Designing a class library is like designing a language! If you use operator overloading, use it in a uniform manner; do not use it if it can easily give rise to misunderstanding.*

I have already described the *uniform manner* above. In other words, a non-**const** reference returned from an assignment is not a problem but an expectation: the absence of a problem does not require a solution, but expectations should be met.

Kevlin Henney

kevlin@wslint.demon.co.uk

operator= and const – a reply

by Mats Henricson and Erik Nyquist

We are pleased to see Kevlin Henney so thoroughly scrutinising one of the recommendations in our public domain document. We are prepared to change this in our forthcoming book “Industrial Strength C++”.

The document was last updated in 1992, and at that time there were quite a few writers that advocated a **const** reference to this as return value. Actually, we got the idea from Scott Meyers after a speech at USENIX C++ 1991 in Washington. Also, Rob Murray’s widely acknowledged book, “C++ Strategies and Tactics” recommends this (page 32, 2.2.1 Return value of **operator=**):

> *Assignment operators should return a constant reference to the assigned-to object.*

One reason why a **const** reference might actually be of least astonishment is that this is the way it works in C. Try this in your favourite C compiler:

```
int main()
{
       int x = 1;
       int y = 2;
       int z = 3;

       (z = y) = x; /* From Sun C
compiler:
       illegal lhs of assignment
operator
       */

       return 0;
}
```

In C++, on the other hand, this code is legal since by default the result of an assignment expression is a non-**const** reference of the object assigned to. This is the motivation as to why a non-**const** reference is appropriate as return value for overloaded assignment operators.

Why have this incompatibility between C and C++? We really don’t know! Maybe Bjarne had a bad day in the early eighties when he decided to change this? ;-)

Mats Henricson

mats.henricson@eua.ericsson.se

Erik Nyquist

eny@alv.teli.se

I asked Bjarne Stroustrup about this gratuitous difference between C and C++ and got the following response – Ed.

Why make the change? Why not? The value of:

```
(a = b)
```

is ***a*** which is an lvalue. Also, we have found real examples of the general form:

```
T& f(T& a, const T&) { return a=b; }
```

Bjarne Stroustrup

bs@research.att.com

Whilst putting this issue together, I was reading Scott Meyers’ column in The C++ Report, January 1995, where he talks about writing max and min functions. He notes that maintaining ***const****-correctness is very difficult with templates and I can now see a parallel between that and the assignment operator. Like Mats and Erik above, I may well change my view on this – Ed.*

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!

In this issue, I report on the Austin WG21/X3J16 committee meeting from March 1995.

The Casting Vote

by Sean A. Corfield

The standards process has now reached a very interesting stage. In my last column I said we would know the result of the Committee Draft Registration (CDR) ballot and whether we would be progressing to the ballot that produces a Draft International Standard. The result of the CDR ballot was as follows: 8 countries voted to register the draft with no comments, 5 countries voted “yes with comments” and 2 countries voted “no”. This means that we will register the CD and proceed into the next ballot bringing an International Standard much closer. According to

the current schedule, the Draft International Standard will be produced at the end of 1995.

What of the comments, though? France and the Netherlands voted “no” because the standard is too large and complex and the library is too large and not yet stable. Both Australia and New Zealand made similar comments but voted in favour. So is the draft really too big? Judge for yourself – we are shortly to enter the ANSI public review: send a mail message to

`c++std-notify@research.att.com`

and when the public review starts you will be notified with details of how to make comments to X3J16 (the U.S. C++ committee). You can also make comments on the draft through your national standards body and for those of you in the UK, you can send your comments to

`c++comments@maths.warwick.ac.uk`

and a member of the UK C++ panel will collate them so that the panel can review them and feed them into the standards process. How will you get access to the Committee Draft? The `c++std-notify` list will tell you – the draft will be available by anonymous ***ftp*** in PostScript and probably PDF formats from both U.S. and UK sites. The review period is not very long so it is imperative you get your comments in as soon as possible.

In the meantime, committee business was conducted pretty much as usual in Austin.

Exception safety

Various concerns about exception handling were addressed in Austin. One concern was memory leaks when placement **new** throws an exception:

```
char buffer[SIZE];
X* p = new (buffer) X;
// if X::X() throws an exception,
// no cleanup is done
```

For the simple example above, the lack of cleanup is not a problem – the placement **new** used does not allocate memory. If placement **new** is, say, a pool allocator, then any memory allocated will not be deleted if the constructor throws an exception. The solution to this was proposed by Bill Gibbons and adopted in Austin: define a placement **delete** that is called automatically in such circumstances. For each **operator new** you define, you will now be able to define a matching **operator delete** that will be called by the implementation if an exception is thrown by the constructor:

```
void* operator new(size_t, bool);
void operator delete(void*, bool);
X* p = new (true) X;
// if an exception is thrown by X's
// constructor, operator delete will be
// called as:
operator delete(p, true);
```

Another exception-related issue is constructor initialisers – with the existing language definition, there is no way to catch an exception thrown by a member initialiser or base class initialiser. This is a problem if you delegate work to a library object and don’t want your users to see the exceptions thrown by that library:

```
class X
{
public:
	X() : libObj() { }
private:
	LibObj libObj;
};
```

If the ***LibObj*** constructor throws an exception, it will propagate to users of class ***X***. This lack of encapsulation led to a proposal to add a **try** block that encloses the initialisers:

```
X::X() throw (XException)
try
:	libObj()
{
}
catch (LibException)
{
	throw XException();
}
```

For symmetry, this syntax is also allowed on ordinary functions.

As I note elsewhere in this issue, the fact that **new** throws an exception on failure is a problem in itself. In Austin, the committee decided that the previous “implementation-defined” hack of allowing ***set_new_handler*** to restore the old behaviour – returning zero on failure – was precisely that: a hack. The solution adopted was originally proposed by John Skaller, I believe, and is extremely elegant: simply provide a placement **new** form that guarantees no exceptions will be thrown. The exact details still need some work, but essentially you will be able to write something like:

```
X* p = new (nothrow) X;
```

and guarantee that if **new** fails, it returns zero instead of throwing an exception. This almost allows you to compile your code with:

```
-Dnew='new (nothrow)'
```

and keep your old behaviour. Almost? Well, you'll have to ***#include*** the appropriate header to get the definition of ***nothrow*** (no, I don't know where it will end up) and it will break any existing placement **new**'s you may have. The committee did not add a symmetric placement **delete** but this may yet appear.

More template extensions

At the San Diego meeting in March 1994 the committee added member templates to the language and some library proposals adopted since have relied on this new feature. Quite a few implementors have expressed concern that member templates are very hard, or even impossible, to implement. In many cases, an alternative to member templates would be the ability to partially specialise a template, e.g., given a generic list class, it would be desirable to be able to specialise this for all pointer types. This facility has now been added for both functions and classes:

```
template<class T> void f(T);  // #1
template<class T> void f(T*); // #2
int i;
int* p;
f(i);  // calls #1 void f<int>(int)
f(p);  // calls #2 void f<int>(int*)
```

This is a combination of specialisation and overloading. If #1 and #2 had been declared in separate translation units, this probably worked on many implementations. For template classes the situation is somewhat more complicated:

```
template<class T> class X { ... };
template<class U> class X<U*> { ... };
```

The second declaration specialises the first for all pointer types. It doesn't declare a template, despite its appearance, but instead specifies a more specialised form of the first template declaration. The partial specialisation can appear to have more template arguments:

```
template<class T, class C> class X<T
C::*>
{ .... };
```

which specialises ***X*** for all pointers to members. Similarly, specialisations can appear to have fewer template arguments:

```
template<class A, class B> class Y { ...
};
template<class P> class Y<P, P*> { ...
};
```

The second declaration is a specialisation of the first – the template ***Y*** still has two arguments. I don't know about you, but I thought this was sufficiently confusing to vote against the proposal. Unfortunately, the majority of the committee did not share my concerns and adopted partial specialisation.

Returning briefly to member templates, the committee resolved a syntactic problem with explicit qualification of member function template calls. There are circumstances where syntax analysis cannot know whether an identifier is a template or not:

```
template<class T>
class Strange
{
...
       int odd()
       {
              return T::f<1>(2);
       }
...
};
```

Since we don't know what ***f*** is (other than it being a member of ***T***), we don't know whether this is an explicit qualification of a template member function of ***T*** or a double comparison:

```
class Static
{
public:
       static int f;
};
class Member
{
public:
       template<int n> void f(int);
};
Strange<Static> ss; // (Static::f < 1 )
> 2
Strange<Member> sm; // Member::f<1> ( 2
)
```

In a similar decision to the use of **typename** to identify member types in dependent names, the committee decided to allow the use of **template** to identify a member template:

```
template<class T>
class NotSoStrange
{
...
       int odd()
       {
              return T::template f<1>(2);
       }
...
};
```

Here, the identifier ***f*** is always treated as a template name and the ***<1>*** means explicit qualification (as in ***Strange<Member>*** above). Without the keyword **template**, the example would mean a double comparison of a static member (as in ***Strange<Static>*** above).

And now, for something completely...

...awful. Namespaces provide a way to parcel up components in a way that avoids name pollution but they have introduced many problems for name lookup. Consider the following program:

```
#include <iostream.h>
int main()
{
	cout << "Hello world!\n";
}
```

At the moment this works because **operator<<** is either a member function (of ***ostream***) or a global function. With namespaces, this would become:

```
#include <iostream>
int main()
{
	std::cout << "Hello world!\n";
}
```

This only works, under the current rules, if **operator<<** is a member function for the **char*** operand (it is). What if the operand is of a different type?

```
#include <iostream>
#include <complex>
int main()
{
	std::complex<double> unity(1, 0);
	std::cout << unity << '\n';
}
```

Perhaps surprisingly, this will not work because the necessary **operator<<** is in the namespace ***std*** and will not be found by name lookup because that scope is not searched under the current rules. The proposed solution was to additionally look in the namespace of the types of the operands, which would solve the above problem (by searching ***std***), but I noted that this would not be sufficient:

```
#include <iostream>
#include <complex>
namespace MyLib
{
	class MyComplex
	: public std::complex<double>
	{
	// ...
	};
}
int main()
{
	MyLib::MyComplex unity(1, 0);
	unity + unity;
}
```

Here, **operator+** is defined in ***std*** and the operand types are both defined in ***MyLib***. The "obvious" answer was to extend the name lookup to also search base class namespaces. This leads to the following possibilities for finding an operator:

1. a member function (found by existing rules),
2. a global operator (found by existing rules),
3. a built-in operator (found by existing rules),
4. an operator declared in the namespace of the types of the operands (new rule),
5. an operator declared in the namespace of any base class of the types of the operands (new rule).

This is the change I alluded to in my comment at the end of George Wendle's article. In the above example:

```
	unity + unity;
```

the search proceeds as follows:

1. look for a member function of ***MyLib::MyComplex*** (there isn't one)
2. look for a global **operator+** (may find some but assume we don't)
3. look for a built-in **operator+** (find several dealing with built-in types)
4. look in the namespaces of the operands (***MyLib*** has no **operator+** declarations)
5. look in the namespaces of the base classes of the operands (***std*** certainly has a suitable **operator+**)

These are all thrown in the pot for overload resolution where, we hope, ***std::complex*** **operator+(*std::complex*, *std::complex*)** wins!

As I said – awful. The UK did not support this but it does seem to solve the problem. Hopefully, someone can come up with examples that have undesirable behaviour under these rules and we can revisit the issue.

Out, out, implicit int!

At a previous meeting the committee voted to ban implicit **int** in a couple of places and deprecate it everywhere else. This meant that a future standard may consider removing the feature. However, the C standard is undergoing revision and it seems likely that WG14 (the ISO C committee) will deprecate, or possibly ban, implicit **int**. In this light, the C++ committee revisited their decision and decided to ban implicit **int** everywhere in the language.

Name injection

Although no vote was taken on this subject, it is a bubbling cauldron for many members of the committee. Here is an example that highlights the problem:

```
class A
{
       friend void f();    // injected
                           //
immediately
};
template<class T> class B
{
       friend void g();    // injected
              // only at instantiation
};
void h()
{
       f();          // fine - injected
by
                     // declaration of A
       g();          // error - no such
                     // name in scope
       B<int> b;     // causes injection
of
                     // friend g()
       g();          // fine - name
                     // injected by
                     // instantiation of
B
}
```

This may not seem too outrageous but consider this example:

```
template<class T> class C
{
       friend void injected();
};
template<class U> void t(U)
{
       C<U>   c;
}
void innocent()
{
       injected();   // error - no such
                     // name in scope
       t(1);         // call t<int>(1)
and
                     // instantiate
C<int>
                     // which injects
                     // injected()
       injected();   // fine - name has
                     // been injected
}
```

Somehow this seems more insidious than the previous example as no explicit template class has been used that could inject the name. What about the following?

```
void confused()
{
       t(1), injected();   // valid?
}
```

It becomes important exactly when an instantiation occurs. The Extensions WG were particularly uncomfortable with this example as it leads to the idea of instantiation sequence points (i.e., madness).

The German delegation are very concerned about this and, in my opinion, rightly so. The potential for confusion is high. In Austin, I suggested that instantiation be done in a synthesised scope so that injected names could not affect the original scope. In the example above, that would mean that the injection of ***injected*()** due to the use of ***C*<int>** would occur only local to ***t*<int>(*1*)** and would not affect the ***innocent*()** function. This seemed to gain support amongst the Extensions WG and we agreed to investigate this further. I hope that we can cap the problems of name injection at the next meeting.

Future meetings

The next ISO/ANSI meeting takes places in July '95 so *The Casting Vote* will next appear in the August issue – *Overload 9*. In the meantime, the public review will have begun so I hope to have many articles from you, the public, about the Committee Draft.

An aside

Because the general public feeling is that we (the committee) should not be adding to language, in Austin we took the decision to disband the Extensions WG – its members will now turn their attention to core language issues such as the "One Definition Rule"...and the many extensions that have been voted into the core language over the last few years.

Sean A. Corfield

sean@corf.demon.co.uk

C++ Techniques

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

Ian Horwill begins a series that examines C++ features from a beginner's point of view and in three related articles, The Harpist, Ulrich Eisenecker and Roger Lever look at object relationships and how multiple inheritance may or may not fit in.

Wait for me! – copying and assignment

by Ian Horwill

I am fortunate enough to (a) enjoy programming and (b) make my living by programming. Unfortunately, I am currently making it in C. I am therefore trying to get up to speed in C++ in some spare time. In this article I would like to present some of the interesting little features of C++ that I have encountered along the way. Anyone who is not a beginner should probably skip ahead to one of the more advanced articles in this issue!

A bug?

Imagine my surprise at finding a widespread bug in the Borland OWL 2.0 class library! In virtually every class, two functions had been declared with no corresponding definitions to be found anywhere.

The declarations were of the following format:

```
X(const X&);
X& operator=(const X&);
```

A little investigation revealed that a bug report to Borland would not be necessary. As you are no doubt aware, the first declaration is for a constructor for class ***X*** that takes a reference to an existing object of class ***X***. It can be used as follows:

```
void print_cheque(
       const Cheque& addressed_cheque,
       Value         amount
)
{
       Cheque
new_cheque(addressed_cheque);

       new_cheque.set_value(amount);
       new_cheque.set_date(today.date);
       new_cheque.print();
}
```

new_cheque is constructed as a copy of ***addressed_cheque*** – i.e., the constructor ***Cheque*(const *Cheque*&)** (called a 'copy constructor') is called to initialise the new object from the value of an existing object.

A couple of points to note here. First, the parameter to a copy constructor is passed by reference and not by value. Passing by value would result in an infinitely recursive succession of calls to the copy constructor to take a copy of its own parameter.

Secondly, had the declaration of ***new_cheque*** been as follows:

```
Cheque new_cheque = addressed_cheque;
```

it is still the copy constructor that would have been called, not **operator=**, because we are still constructing a new object rather than assigning to an existing one.

To complete the definition of a copy constructor, it is any constructor that can take a single argument of its class type, e.g.,

```
X(const X&, int = 42);
```

is also a copy constructor for class ***X*** because a default value is provided for the **int** parameter.

Now for **operator=()**. To recap, C++ allows us to define functions named after the built-in operators such as *, +, << and of course =. The 'name' of such functions is the symbol itself preceded by the keyword **operator**. For example, we could define the **operator+=** to concatenate one string to another:

```
String& String::operator+=(
       const String& s2
)
{
       String&      s1 = *this;
       // Code to append s2 to s1
       return s1;
}

void my_function()
{
       String s1("Remember, remember,");
       String s2(" the Fifth of
November");
       s1 += s2;
       // Could also write:
       // s1.operator+=(s2); but it
rather
       // 'defeats the object' :-)
}
```

Therefore, **operator=()** could be used as follows:

```
void print_cheque(
       const Cheque& addressed_cheque,
       Value         amount)
{
       Cheque new_cheque;  // initialise
              // with default constructor

       new_cheque = addressed_cheque;
       new_cheque.set_value(amount);
       new_cheque.set_date(today.date);
```

```
        new_cheque.print();
}
```

Here, **operator=()** is called to change the value of one existing object (***new_cheque***) based on the value of another (***addressed_cheque***).

Well this is all very interesting, but the problem is the compiler generates a default copy constructor and a default **operator=()** for every class that doesn't have explicit versions. These default versions initialise or assign to each member of the class, using that member's explicit or default copy constructor or assignment function. For example:

```
class A
{
public:
        A& operator=(const A&);
};

class B
{
        ...
private:
        A      a;
        char*  p;
};

void f()
{
        B b1, b2;
        ...
        b1 = b2;
}
```

In this example, ***b2.a*** is assigned to ***b1.a*** using the **operator=()** defined for class ***A*** and ***b2.p*** is assigned to ***b1.p*** using normal (memberwise) assignment.

The problem arises if you don't want users of your class (or member functions of the class itself) to be able to make extra copies. For example, if your class object monitors a physical resource, it probably doesn't make sense to have multiple copies of the resource monitor being passed around and getting out of step with each other. However, unless you declare a copy constructor and assignment operator, the default versions will allow extra copies of class objects to be made willy-nilly.

The solution? You do declare a copy constructor and an assignment operator. However, you don't have to define them! The mere fact of having declared them prevents the compiler from generating default versions. Of course, you can define them if you want to – you will still be in control of the copying.

If you put the declarations in the **private** section of the class declaration, the compiler will reject attempts to call them from outside the class itself. Inadvertent calls within member functions will cause the linker to complain that it can't find the required definitions.

Well that about wraps it up. Editor willing, next issue's article will be about perhaps the most confusing keyword in C++ – **virtual**. I'd be delighted to hear about any C++ issues that make you feel 'left behind'. Be warned that my answers will be based on "The Annotated C++ Reference Manual" (1991). I'll leave it to the likes of Sean to correct me on the latest developments.

Ian Horwill

100441.3700@compuserve.com

I'm willing, so get writing! :-) – Ed.

Related objects
by The Harpist

I sent Francis an article to forward to Sean Corfield about some uses of multiple inheritance. He read it and returned it with the suggestion that it was really the tag end of a much larger topic. So here is the first of what I intend to be a two part article on the way objects are related and can be used as components.

In the beginning there were the built-in types inherited from C. There were also a number of derived types, pointers, arrays (perhaps more correctly, vectors) and **struct**s. C and C++ added type qualifiers – one each, C++ added **const** and then C added **volatile**. In C, type qualifiers were just that and nothing more. They represented two simple concepts, read only access and unreliable memory (memory that could change at the most inconvenient moment by intervention from outside the program).

C++ added references

None of this would have mattered had not overloading been introduced into C++. That changed the rules out of all recognition. The type system was invoked to support overload resolution and suddenly types started to sprout in all directions. For example we now have not one but eight flavours of **int** (**int**, **const int**, **volatile int**, **const volatile int**, **int&**, **const int&**, **volatile int&**, **const volatile int&**). Are all the flavours actually different? Well yes, and no. It all seems to

depend on the context. For example when an **int&** goes out of scope, the underlying memory almost always remains.

Almost always – why not always? Well it might be a reference to a temporary. Now that leads to the interesting question for language lawyers “What is the difference between a local variable and a local reference bound to a temporary?” You don’t know? Well don’t ask me, because I haven’t the vaguest notion.

(When the language lawyers have finished with that question, perhaps they will turn their minds to what sort of type a **mutable int** is? I guess you cannot have a **mutable const int**, but can you have a **mutable volatile int**? These are really tough questions, and need not concern most of us but they highlight the problems that are caused by even apparently sensible minor extensions to the language.)

*Whereas **const** and **volatile** are cv-qualifiers and, hence, modify the type, **mutable** is ‘only’ a storage-class-specifier and does not affect the type. Members of a **const** object are not normally modifiable (e.g., inside a **const** member function) without casting away **const** – **mutable** was provided to obviate the need for the cast in certain well-defined circumstances – Ed.*

So far all we have in C++ is C things turned into types. Actually we can do quite a bit with this, particularly when we add in class concepts and conversions, both via constructors and via operators. The fun starts when we add in the next layer: derivation.

At its simplest, derivation just allows us to reuse code even when we do not have access to the original. If it stopped there we wouldn’t have much of a problem, but we also wouldn’t have the tools for object oriented programming.

This form of derivation often has a sense of refinement or improvement. Its like taking the basic concept of a screw and adding the idea of a cross-head to it. It often suffers from the same problems, something simple and utilitarian becomes more specialised and complicated to use. We can – at a stretch – use a knife on an old plain screw. Knives do not work on machine screws – worse, we need just the right sort of cross head screwdriver if we are not to damage our high-tech screw.

Hidden inside the concept of derivation for reuse is the concept that a derived type is a replacement for the original. To understand what is happening we have to step back and see that the concept of derivation gives us another way to build new types.

The old method is called aggregation or layering. We assemble a new type by wrapping up a number of earlier types into a single package. Aggregation is a little like using Lego®: you start with a number of building blocks, push them all together and finish up with something useful though more complicated. It is like building a computer from components, motherboard, power supply etc.

The new method allows us to start with an object and then add modifications to it. That gives us a decision to make. Should we cram a whole lot of bits together (aggregation) or should we modify an existing type (derivation).

Upgrading your computer

Even at this level we can have problems. Is replacing the video card in our computer derivation or aggregation or neither? Think very carefully because I do not think that even this simple real world action can be properly modelled with simple C++ technology.

When we designed our computer class we allowed for replacement video cards because we provided a ‘pointer to video card’ onto which we could attach a specific instance of a subtype of video card. But how do we provide for the enhanced functionality that our new card provides?

Perhaps we should have provided a function pointer for our video driver. Yes, that is obviously the answer. Have fun. Going fully object-oriented can take an awful lot of time.

Note that you cannot derive your SVGA computer from your old VGA one even though the former is a VGA machine because you are replacing a data item and over-riding functionality.

Inheritance versus aggregation

The mythology of object-orientation gives us a simple rule of thumb to decide which approach to use. We are supposed to ask if ‘X is a Y’ or does ‘X have a Y’?

I deliberately used the term mythology because this question is simplistic and misleading. It doesn’t work. It is a lousy criterion and conceals

some really serious problems with object-orientation. Problems so serious that I can find no answers within the C++ type system. Let me give you two examples:

A circle and an ellipse

I can remember Francis drumming into us in maths lessons that ‘a circle is an ellipse’. Mathematically a circle has all the properties of an ellipse. Mathematically a circle is a specialised ellipse.

Now let us look at an ellipse from a C++ type point of view. We sit and list all the functions that we want to apply to an ellipse. One of these functions will involve change of eccentricity either explicitly or through some other change such as magnification in only one dimension.

Actually change of eccentricity is a rather good function to consider – what happens if the eccentricity goes negative? Exactly! The ellipse stops being an ellipse. You see, our names for various conics deal with specific constraints that we can apply to one or more defining properties.

The OO concept of ‘is a’ requires that the derived object can substitute for the original in all cases. In the case of circles and ellipses this is not true. In fact, I know of no way of representing the relationship between a circle and an ellipse in terms of the C++ type system.

Some will claim that I have simply taken a pathological case and that most things fit the type system quite happily. I think that this will be seen to be just about as true as the Victorian attitude to what we now call fractal curves. The unnatural case is not the fractal one but those shapes that have an integral dimension.

Complex numbers and reals

Scott Meyers gives this as another example of ‘is a’ breaking down. He isn’t strictly correct because mathematically a real isn’t a complex number (with a zero imaginary part). However there is an isomorphism (one to one mapping of both data and operations) between reals and the subset of complex numbers with a zero imaginary part.

It is virtually impossible to represent this relationship in an object-oriented fashion. It makes no difference whether you try to derive complex from real, real from complex or provide a conversion operator; the relationship simply does not fit. The best we can do is to consider whether it is worth providing semi-intelligent division (and perhaps multiplication) to cope with cases where complex operands degenerate to reals.

It is worth noting that most implementations of complex numbers you find in books and magazines completely ignore efficiency in this area. Division of a complex by a real, multiplication of a real by a complex and of a complex by a real should be provided directly and not by converting a real to a complex.

I have written about reals here because I am looking at this from a mathematical view but it is worth noting that there are no reals in computing, only rationals.

Polymorphism

Where a number of sub-types share functionality but differ in implementation of that functionality it makes sense to design an abstract base class that declares the functionality with (pure) virtual functions which will be defined in the derived classes. But if this is what you are doing you should think very carefully before adding functionality in a derived class. If you do so, it will only be available directly through that sub-type. This seems to be an error to my way of thinking.

A cluster of polymorphic types should be interchangeable, whatever one can do the others should be able to do as well, though by a different mechanism.

Perhaps that last paragraph overstates the issue, but I wrote it because so many texts seriously understate it.

Take the example of your ***Shape*** hierarchy. The purpose in providing such a hierarchy is precisely because you will not know at compile time what specific shapes will be used. You can only use generic shapes in your program so providing any special feature for a specific shape will be a complete waste – you will not be able to use it.

Inheritance for modification

Though based on substantially the same language mechanisms this use of inheritance is completely different. We are not trying to model a cluster of functionally related objects with different implementations. What we are trying to do is to reuse an earlier implementation by changing or enhancing it.

In this situation I can accept suppression of functionality in the derived class, addition of functionality and even quite radical modification. Some will argue that private bases should be used in such cases. I do not agree. I see nothing wrong with taking table and deriving a folding table from it. You can even use your folding table as a table. However if you want the property of folding you will need to use it as it really is. We are not using polymorphism, we must know that we have a folding table before we can use it as such.

I think that the main motive for RTTI (run time type information) is to try to cater for this double view of inheritance so you can have polymorphism and modification at the same time. Next time you will see that I think such duality is best implemented via multiple inheritance.

Template classes

These add an entire new dimension to the possibilities. They deal with the cases of things that are usually functionally identical, down to implementation detail, but based on unrelated types. Inheritance deals with multiple refinements and specialisations from a single base class. Template classes deal with similarities for distinct, unrelated types.

For example, for type safety a container class needs to be coded for the type of object that it is containing. We need separate linked lists of **int**s, **float**s, ***Shape***s etc. We need these because we will often need type specific declarations for variables, parameters and return types even though the functionality is identical.

Polymorphism deals with "same data sets, different implementation details" while template classes deal with "same implementation details, different data sets".

I oversimplify because sometimes a template class will need a specialisation to provide an implementation tuned to a specific data set. But it is the principle that concerns me here.

Summary, different types

- Built-in types, sometimes called scalars.
- Qualified and derived built-in types (pointers, **const**, reference etc.)

These two groups are related both within each group and between groups by built-in conversions. Any attempt to summarise the rules is about as complicated as simply listing the conversions.

Simple user defined types: **enum**s, **union**s, **struct**s and **class**es. The relationships between these are governed by built-in rules (e.g., those for **enum**s) and by user provided specifications (single parameter constructors and conversion operators).

Derived user defined types. For cv-qualification, etc, user defined types follow the same rules that apply to built-ins. Those derived from bases have both a language-provided relationship between base and derived as well as a conceptual relationship. The conceptual relationship can include polymorphism.

Template types (classes) raise another question. What part, if any, do they play in the type system? Before you dismiss this as a trivial question answered with 'none', stop and consider the impact of partial specialisation. For example:

```
template<class T1, class T2> sometype
{...};
template<class T> sometype<int, T>
{...};
template<class T> sometype<T, int>
{...};
sometype<int, int> s;
```

What happens to this declaration of ***s***?

It should be ambiguous but maybe we'd better wait to see the exact wording in the working paper, since such partial specialisations were only added in Austin – Ed.

Even before we consider multiple inheritance (next issue) we have a rich range of choice. Mixing single inheritance with template classes is really fun.

The problem is that we need to have a very clear idea about the strengths and weaknesses for each method for developing new user defined types. The classic 'is a' and 'has a' relationships are completely inadequate. As we have seen they do not relate to much of our formal experience in mathematics. The excuse that attempting to derive a square from a rectangle shows failure to analyse the problem domain correctly is a cop out. Show me how to do it properly!

What is the relationship between single and double precision maths? (not just floats and doubles, but complex floats and complex doubles, quaternion varieties, polynomial ones etc.) This would seem to be the domain of template classes even

though there will probably be only two (perhaps three with **long double**) types of each. How do I provide conversions between types based on the same template? To be honest, I do not know. For many the answers are of no importance but for those working in computationally intense areas it matters a lot.

Theoretically, by using member template conversion operators...if anyone can ever get them to work properly – Ed.

Conclusion

I started out to write about multiple inheritance (mixins and addins). Francis persuaded me to think again on the grounds that there was much more to the story. On reflection, I have to agree that he was right though the problem is that much of the rest is like the old maps annotated with 'Terra Incognita'.

Before we even begin to think about MI, we need a much better understanding of how to use the C++ type system to develop objects that map the relationships found in the real world.

The Harpist

Related addendum

by Francis Glassborow

I have given a lot of thought to the problem exemplified by the relationship between circles and ellipses. One of the most unfortunate features is that polymorphism is so often explained in terms of ***Shape*** and ***draw***(). To get that inheritance graph right requires a deeper understanding of problem domains and OO than is possessed by most.

What is needed is a mechanism for providing polymorphic objects rather than polymorphic types. In other words we need an object that is sometimes a circle exhibiting circle functionality and is sometimes an ellipse with elliptical behaviour. The same object, but two behaviours. I think I can do it but before I write it up for the next issue, I'd be interested to hear your ideas on the subject.

Exercise

Write up a C++ implementation of the relationship between circles and ellipses. Send it in and I'll collate the results and then provide my own answer.

That will be easy because it takes a mind that thinks round corners to tackle the problem and most (if not all) of you will leave it to someone else.

Francis Glassborow

francis@robinton.demon.co.uk

I hope that quite a few of you will prove Francis wrong :-) – Ed.

Multiple inheritance in C++ – part I

by Ulrich W. Eisenecker

This is the first in a series of articles. This part is about the basics of multiple inheritance such as syntax and multiple base classes and their initialisation. As an introduction, I will summarise details of inheritance and virtual functions.

A review of inheritance and late binding

Inheritance is mainly a technique for reusing a description of an existing class to describe a new class. If ***Derived*** inherits from ***Base*** it means, that in some respect ***Derived*** is like ***Base***. Normally one would add further data members or methods to ***Derived***. It is even possible to substitute an inherited method with a new implementation. This may be a complete substitution or an extension, in the sense that there is new code which calls the old implementation. From this point of view it is adequate to speak of a class hierarchy. To illustrate this relation it may be helpful to think of ***Derived*** having a ***Base*** subobject (Fig. 1). This relationship is not to be confused with a has-part relationship.

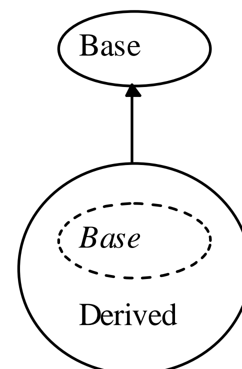


Fig. 1: Inheritance between classes

Another important aspect is that, by default, method calls are resolved at compile time (stati-

cally). Consider a method ***m*** in ***Base***, which is over-ridden in ***Derived***. If a pointer to ***Derived*** is assigned to a pointer to ***Base***, invoking ***m*** for that pointer will execute ***Base*::*m*()**. Actually, in most circumstances the execution of ***Derived*::*m*()** is wanted. To achieve this, so called late binding is needed, which takes place at run-time. To specify late binding for a method, its declaration in a class is qualified by the keyword **virtual**. This needs to be done only once (in ***Base***) to be effective for all descendants of ***Base***, but it is not an error to repeat it when declaring a method over-riding ***m***. In the example below, screen output is marked by a preceding ">".

```
class Base
{
public:
  virtual void hello()
  { cout << "Base::hello()\n"; }
};

class Derived : public Base
{
public: // Next use of "virtual" is not
        // necessary!
  virtual void hello()
  { cout << "Derived::hello()\n"; }
};
...
Base* p = new Base;
p->hello();
p = new Derived;
p->hello();
...
>Base::hello()
>Derived::hello()
```

In C++, inheritance can be controlled by access specifiers, namely **public**, **protected** and **private**. With **public** derivation an instance of ***Derived*** can always be used when an instance of ***Base*** is expected. From this point of view one may speak of a type hierarchy. If inheritance is **protected** or **private**, the described assignment and execution of inherited methods no longer works.

```
class Base
{
public:
  virtual void hello() {}
};

class public_Derived : public Base
{
public:
  virtual void hello() {}
};

class protected_Derived : protected Base
{
public:
  virtual void hello() {}
};

class private_Derived : private Base
{
public:
  virtual void hello() {}
};
...
Base* p;
p = new public_Derived;    // ok
p = new protected_Derived; // error
p = new private_Derived;   // error
```

This simply means that in C++, a class hierarchy does not necessarily coincide with a type hierarchy. And, in contrast to many other object-oriented programming languages, C++ provides language constructs to explicitly express differences between those hierarchies and therefore to control them.

The need for multiple inheritance

Multiple inheritance is a simple extension of single inheritance in so far as a class can inherit directly from more than one class. Multiple inheritance is often said to be unnecessary. This is not true for at least two reasons:

1. There are cases when modelling using multiple inheritance preserves more of the problem-specific semantics.
2. Due to the inheritance-based polymorphism in C++, multiple inheritance is essential for accessing combined objects by pointers.

Let us look at an example, which is taken from [EIS93], where a phone and a TV form a new device. We start with the following classes:

```
class Phone
{
public:
  virtual void dial(char* number)
  {
    cout << "Dialling " << number
         << "...\n";
  }
};

class TV
{
public:
  virtual void switchOn()
  { cout << "TV switched on.\n"; }
};
```

A first approach to building a two-in-one device could be to make either a ***TV*** or a ***Phone*** part of a new device called ***PhoneTV***. In either instance you must forward specific requests to the embedded device:

```
class PhoneTV : public TV
{
  Phone aPhone;
public:
  virtual void dial(char* number)
  { aPhone.dial(number); }
};
```

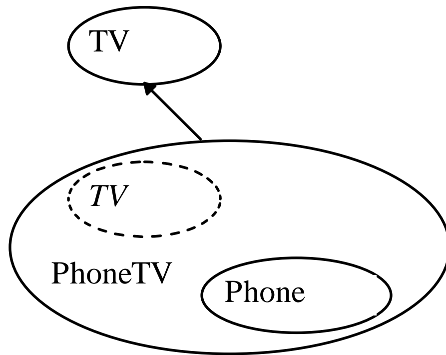


Fig. 2: ***PhoneTV*** with single inheritance

An instance of a ***PhoneTV*** can be switched on and can be used for dialling a number:

```
PhoneTV aPhoneTV;
aPhoneTV.switchOn();
aPhoneTV.dial("073127174");
```

But what happens if a pointer to a ***TV*** is initialised with a dynamically created object of type ***PhoneTV***?

```
TV* aPhoneTV = new PhoneTV;
aPhoneTV->switchOn();
aPhoneTV->dial("073127174");
```

At least BC 4.0 issues the error “'dial' is not a member of 'TV'”. That is because there is no method ***dial*** defined for ***TV***, and the information about the availability of ***dial*** is lost when assigning a pointer to ***PhoneTV*** to a pointer to ***TV***. If, instead, we try:

```
Phone* aPhoneTV = new PhoneTV;
```

the compiler complains that it “Cannot convert 'PhoneTV *' to 'Phone *'”. The reason is that ***PhoneTV*** is not a descendant of ***Phone***.

Without explicit type conversion, pointers to a more specialised class may only be assigned in C++ to a pointer to a public ancestor of this class (i.e., all inheritance provided by **public** derivation).

This means that polymorphism in C++ works only along the inheritance graph. This can be different in other object-oriented languages. For instance, polymorphism in Smalltalk is signature-based. A Smalltalk-object receiving a message checks whether the signature (message name plus parameters) of the message is known to the object’s class or to any of its ancestors. If so, the first method found is executed. Using this technique, called forwarding, (fig. 2) is a common procedure for combining the behaviour of two classes in Smalltalk. Signature-based polymorphism means that there is no need for multiple inheritance in Smalltalk, even though combining classes in this way can be conceptually dirty.

The way to solve the problem with phones and TVs in C++ with only single inheritance is to introduce a common superclass for ***Phone*** and ***TV***, which has abstract methods ***dial*** and ***switchOn***. But this is not a good design, since the devices which will be combined in future are unknown. That implies the need to change the definition of this superclass whenever another method is needed. This implies many problems: the source code must be available, recompilation is necessary, the semantics of derived classes may be affected, name conflicts may occur if a derived class already has a method with the same name, and so on. Classes, and especially abstract classes, should always be designed to be stable and only be altered as a last resort. The problem of overloaded root classes is well known in languages without multiple inheritance but providing polymorphism through inheritance. See the early versions of C++ (e.g., in The Annotated Reference Manual).

Syntax of multiple inheritance

So all that is necessary is multiple inheritance, and the syntax is quite simple. The classes from which the derived class inherits are listed, separated by commas:

```
class PhoneTV : public Phone, public TV
{};
```

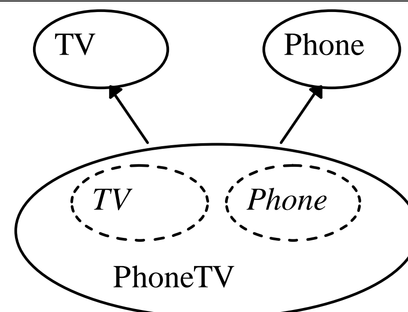


Fig. 3: ***PhoneTV*** with multiple inheritance

Now all works as expected:

```
PhoneTV* aPhoneTV = new PhoneTV;
Phone* aPhone;
TV* aTV;
aPhoneTV->switchOn();
aPhoneTV->dial("0731-27174");
aPhone = aPhoneTV;
aPhone->dial("0731-27174");
aTV = aPhoneTV;
aTV->switchOn();
```

Of course it is possible to mix **public**, **protected** and **private** derivation deliberately.

Initialisation of base classes

As always in C++, there is something going on behind the scenes! Let us add default constructors and destructors to ***TV*** and ***Phone***:

```
class Phone
{
public:
  Phone()
  { cout << "Phone\n"; }
  virtual ~Phone()
  { cout << "~Phone\n"; }
  virtual void dial(char* number)
  {
    cout << "Dialling " << number
         << "...\n";
  }
};
class TV
{
public:
  TV()
  { cout << "TV\n"; }
  virtual ~TV()
  { cout << "~TV\n"; }
  virtual void switchOn()
  { cout << "TV switched on.\n"; }
};
```

Now it can be shown that the order in which the base classes are declared determines the order in which constructors and destructors of the base classes are called. In the next examples screen output is again marked by a preceding ">":

```
class PhoneTV : public Phone, public TV
{ };
...
PhoneTV();
...
>Phone
>TV
>~TV
>~Phone

class PhoneTV : public TV, public Phone
{ };
...
PhoneTV();
...
>TV
>Phone
>~Phone
>~TV
```

This ordering can not be overridden by explicitly calling the constructors of base classes in a different order:

```
class PhoneTV: public Phone, public TV
{
public:
  PhoneTV() : TV(), Phone()
  {}
};
...
PhoneTV();
...
>Phone
>TV
>~TV
>~Phone
```

Disambiguation of name conflicts

What if both ***Phone*** and ***TV*** have a method ***mute*** introduced? For ***Phone***, ***mute*** means that transmission of speech is interrupted, until ***mute*** is pressed again. When ***mute*** is sent to an instance of ***TV***, the speaker volume is set to zero. Pressing ***mute*** again, restores volume to its original value. For the purpose of demonstration, the methods just print out their names. All works fine until the moment ***mute*** is called. Then it is necessary to resolve the conflict. This is done by qualifying ***mute*** with the name of the desired class followed by two colons:

```
class Phone
{
public:
  virtual void dial(char* number)
  {
    cout << "Dialling " << number
         << "...\n";
  }
  virtual void mute()
  { cout << "Phone::mute\n"; }
};
class TV
{
public:
  virtual void switchOn()
  { cout << "TV switched on.\n"; }
  virtual void mute()
  { cout << "TV::mute\n"; }
};

class PhoneTV : public Phone, public TV
{};
...
PhoneTV().Phone::mute();
PhoneTV().TV::mute();
...
```

A nice challenge is modelling a twin-phone. What about simply deriving it twice from a phone?

```
class TwinPhone : public Phone, public
Phone
{};
```

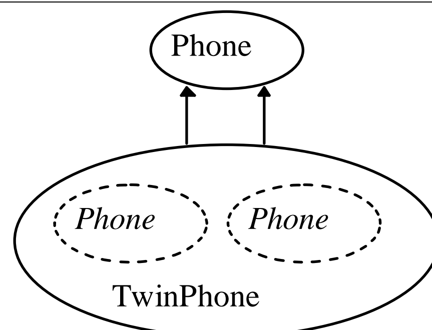


Fig. 4: An impossible ***TwinPhone***

As this new class now incorporates two phones, it also has two methods ***dial***. The mechanism introduced above to resolve ambiguities will not work here, because there is no way to distinguish one phone from the other. This is the reason that C++ forbids direct derivation from the same class more than once. However, a class may indirectly inherit a base class any number of times. Conflicting names can then always be disambiguated by providing suitable class scope qualifiers using the **::** notation.

Next issue

In the next article, I will introduce virtual base classes using an example from mathematics – combinations, and a program to generate them.

References

[EIS93] Eisenecker, Objektorientierung und Wiederverwendbarkiet. In: unix/mail 6/93, pp420-429.

Ulrich W. Eisenecker

eisenecker@dbag.ulm.DaimlerBenz.com

On not mixing it...

by Roger Lever

The articles in *Overload 6* by Francis Glassborow (Friends – who needs them?) and Graham Kendall (Putting Jack in the Box) were very interesting, but I wasn't entirely comfortable with the concepts being put forward. So I decided that I would put pen to paper.

Before I put forward a rationale for an alternative approach allow me to establish my credentials – I have none! I work as an Analyst Programmer using mainly Visual Basic, MS Access and Plexus (a 4GL specialising in imaging). My personal interest is in C++ and my experience to date is at the 'toys' level, but I take my toys very seriously!

The section entitled "Mixins and printable" (pp10-11) takes an approach with which I am not entirely comfortable. I can see the rationale and it offers a certain elegance but public inheritance should be used to mirror the problem domain and express one of the two (now) classic relationships of:

1. *is-a* e.g., a car *is-a* type of, or kind of vehicle
2. *has-a* e.g., a car *has-a* engine (also known as composition)

However, the article uses mixin classes (***Printable*** and ***Storable***) and creates an inheritance hierarchy for ***Record*** that does not express this *is-a* or *has-a* relationship. The article points out that the alternative approach of using *has-a* fails because:

- You can't instantiate an ABC (i.e., ***Printable*** or ***Storable***)
- Late (or dynamic) binding requires an inheritance hierarchy

I shall come back to this thread later, for now I want to move onto a later article within *Overload 6*.

The section entitled "An answer from the Harpist" (pp22-25) stresses the difference between Object Based Programming (OBP) and Object Oriented Programming (OOP). However, the solution to the problem "Putting Jack in the Box" seems overly complex, in particular the use of contents and container as part of the inheritance hierarchy. The inheritance hierarchy again does not map onto the *is-a* relationship but is used as a mechanism to enable a polymorphic solution. I'm in favour of an alternative design:

1. The container view of the problem should be expressed with templates
2. The inheritance hierarchies should only use *is-a* / *has-a* – like ***Person***

If a solution can be expressed simply then it should be, so opportunities to simplify multiple inheritance should be examined. An example of simplifying a multiple inheritance hierarchy is in section 12.2.2 of Bjarne Stroustrup's C++ Programming Language 2e (pp404-407). More generally, Tom Cargill's C++ Programming Style also offers excellent general advice with a chapter dedicated to unnecessary inheritance.

Both of the articles use inheritance incorrectly when using the strict *is-a* or *has-a* interpretation. The mixin approach appears to offer a simple solution to providing printer and disk services and the alternative of using composition fails on the two items quoted above. At least that was the author's contention – I'm not so convinced, but then again I do fall into the category of inexperienced! Everything has a cost, so what are some of the costs with the mixin style?

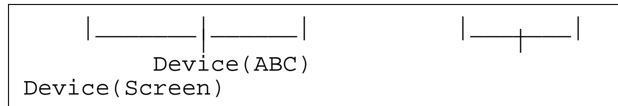
- The complexity of the software rises (OK! Very subjective :-)

- *is-a* and *has-a* inheritance are subverted to use / add a mixin style
- Multiple inheritance is invoked very quickly and also subverted
- Virtual base classes become almost a necessity
- The potential impact of ambiguity (collision of names) rises
- Recompilation costs are increased by the inheritance lattice

My objective is to show an alternate to the mixin design, which uses 'proper' inheritance. In the process I also hope to provide an answer to the two quoted objections to using composition.

The key to design is to find the right abstractions for the problem. The two abstractions here are record and device where device could be the screen, printer or hard disk. The important point is that the services required, "printable" and "storable", have been abstracted into a ***Device***. ***Device*** can therefore be an ABC, or the base class of an inheritance hierarchy if we want the benefits of dynamic binding. This approach does not subvert *is-a* as a ***Printer*** (or ***Disk***) *is-a* ***Device***.

```
Lattice 1                   Lattice 2
  Screen  Printer Disk        Printer
Disk
```



If ***Device*** is an ABC (Lattice 1 above) then it cannot be instantiated (and the compiler gives an error). However, this would be the preferred approach as it defines the interface for all objects derived from it. However, I started with the second version! (Lattice 2 above) The reason is that I started with just a ***Device***, printing to the screen, and ***Record***. I ran across a number of problems before arriving at this solution. Code implementing this lattice is shown at the end of this article. There is plenty of scope to improve this code, such as using an ABC to define a minimal but complete interface for ***Device***, adding exception handling etc.

The code given below uses the C++ version of multiple polymorphism and also uses a buffer to reduce the coupling between ***Record*** and ***Device***. If readers are interested in how exactly I arrived at this point I could be persuaded to bore you some more!

Roger Lever

rnl16616@ggr.co.uk

It would probably be quite educational to see the earlier, discarded, designs that led you to this one – Ed.

```
// Compiled using Borland 4, Output to a DOS Standard EXE file
// No special code used or Borland specific libraries. Organised into
// two files record.h and main.cpp. All classes were defined inline in a
// single module - this is only suitable as an example.
// Complete listing of the working code which can be cut and pasted into
// a project for experimentation. Starts from here...
// record.h-----------------------------------------------------------
#include <strstrea.h>     // provide the buffer service for output
#include <fstream.h>      // provide the ofstream extensions to device

// Device default output is to the screen member functions are
// virtual as inheritance will be used to extend this class to
// different types of devices, such as disk, printer, optical...
class Device {
public:
       Device(void) { cout << "Device born\n"; }
       virtual ~Device(void) { cout << "Device dies\n"; }
       virtual void output(ostrstream& os) const {
              cout << os.str();
       }
};

// Very basic record class inspired by Overload 6. It is declared
// after the Device class since output() takes a Device parameter
// Functions are declared virtual since a derived class will want
// to exploit the polymorphic behaviour especially buildOutput()
class Record {
public:
       Record(void) { cout << "Record born\n"; }
       virtual ~Record(void) {
              cout << "Record dies\n"; strm.rdbuf()->freeze(0);
       }
```

```
	virtual void output(Device* dev) { dev->output(strm); }
	virtual void buildOutput(void) {
		strm << "Build Record output\n" << '\0';
	}
protected:
// Protected to enable derived classes to access strm but not
// provide public access to it
	ostrstream strm;
};

// Very basic extension of the Record class to demonstrate
// dynamic binding within a derived class which uses the
// inherited interface item output()
class ExtendRecord : public Record {
public:
	ExtendRecord(void) { cout << "ExtendRecord born\n"; }
	virtual ~ExtendRecord(void) {
		cout << "ExtendRecord dies\n";
	}
// Override what the derived class wishes to send to output
// but there is no need to override the behaviour of output
	virtual void buildOutput(void) {
		strm << "Build ExtendRecord output\n" << '\0';
	}
};

// Extend device to support generic harddisk services. This class
// should ideally support more options especially filename and
// file access mode
class Disk : public Device {
public:
	Disk(void) {
		cout << "Disk device created with default hardcoded name\n";
	}
	virtual ~Disk(void) { cout << "Disk dies\n"; }
	virtual void output(ostrstream& os) const {
		cout << "Disk writes to rubbish.txt\n";
		ofstream out("rubbish.txt", ios::app);
		out << "Disk output to a file:-" << os.str();
	}
};

// Extend Device to support generic printer services. This class
// would need to encapsulate the horrible details of dealing with
// hardware. For example the ‘print-stream’ may be fine but the
// desired result may not be achieved because the printer is
// disconnected, out of paper...
class Printer : public Device {
public:
	Printer(void) { cout << "Printer born\n"; }
	virtual ~Printer(void) { cout << "Printer dies\n"; }
	virtual void output(ostrstream& os) const {
		cout << "Printer output echo to screen\n";
		ofstream cprn(4, os.str(), os.pcount());
		cprn << os.str();
	}
};

// main.cpp-----------------------------------------------------------
#include "record.h"
int main() {
// Uncomment one of the following three devices to show dynamic
// binding in action. Use device for a generic device which prints to the
// screen. This device is extended to include disk and printer services
//	Device* pdev = new Device;
	Device* pdev = new Disk;
//	Device* pdev = new Printer;

// Create an arbitrary record and output to the required device
	Record a;
	a.buildOutput();
	a.output(pdev);
// Create an extended version of record and output to the required device
	ExtendRecord b;
	b.buildOutput();
	b.output(pdev);
// Clean up the new’d item, destructors will cleanup the record objects
```

```
        delete pdev;
        return 0;
}
// End -------------------------------------------------------------------
```

editor << letters;

Hi Sean!

Well although I am a member of the C++ SIG, I admit that I am not an accomplished C++ programmer, and at the moment program in C. The reason for this letter is, I realise that although as a Special Interest Group the majority of members are probably fairly competent C++ programmers, the reason that I joined the SIG was to learn more C++. So, would it be possible to have some kind of “beginners’ corner” or some kind of series running to introduce C++ in an efficient manner. I’m not saying “start from scratch”, but maybe show the advantages of C++ over C and where it can be used to great effect. Maybe some kind of project where a final product is produced and people would contribute ideas. If it started off from a fairly basic level, this could introduce basic C++ and also allow people to offer ideas and thoughts. It would also be possible to ‘teach’ program design and the use of methodologies and the project may highlight limitations of certain methods, some people could probably offer new, or modified, ideas as regards program design, or show the way forward when everyone is baffled. This may well make the SIG more accessible, but my perception of what its function is may differ from other people’s ideas.

I realise I’ve only put forward an idea, but why not put it to the SIG and see what people say? And that a lot of extra work would need to be done in order to get something like this started. Unfortunately I’m nearing the end of my final year and so can only say I’ll try something after I’ve finished. It probably would be hard to keep the more experienced programmers from ‘throwing their weight’ around, but they could lend a hand when things go wrong or basically didn’t go forward. It would also probably need an accomplished and dedicated programmer to control the whole idea.

Phil Shotton

SP134@greenwich.ac.uk

I completely agree with Phil – it’s all very well getting advanced C++ articles from many of the accomplished programmers in the membership but novices need a hand too. CVu caters for those C programmers interested in C++ but Overload should cover everyone moving beyond that. From the various responses I’ve had to Overload 6, I think I can safely say that there will be a broader mixture of articles in future issues – Ian Horwill’s article in this issue is a good example.

Dear Sean,

Welcome to your new post as Editor. Best of luck!

FWIW & IMHO (this was sent by e-mail, after all!), some comments on *Overload 6*:

First of all, the level of expertise from your contributors is impressive. I feel there is a lot to learn about C++ programming and this is a good place to do it. I also love being able to read about what’s going on at the standards meetings.

The ubiquitous Francis has come up trumps with his article on friends and how to get rid of them. This is just the sort of removing of wool from eyes that we could do with more of. However, I find it interesting that Francis can talk with the same vehemence about leaving the return value out of ***main***, and adding **return;** to the end of **void** functions; surely this is of far less practical importance!

Moving on, I find myself disagreeing with George Wendle on the evils of allowing overloading on **const**. If we were worried about language features letting us do stupid things, we’d be programming in Ada, not C / C++! It’s valid to point out the pitfalls and, forewarned, press on. Let good programming continue to come from understanding, not restrictions (which approach never works anyway).

Well done to Kevlin for submitting his proposal to the ISO committee. I hope it gets through. **wchar_t** looks too much like a user type for my liking. With regard to **operator=** returning a

const or non-**const** reference, currently we have the choice (i.e., ***X*& operator=(const *X*&)** vs. **const *X*& operator=(const *X*&)**) so what's the problem? Please don't promote language restrictions to force people to write 'good' code.

I loved the combined article from Graham Kendall and the Harpist on "Putting Jack in the Box". This style of article is extremely informative. I hope you will continue to feature such articles, even if they have to be contrived (to overcome a shortage of people humble enough to submit 'trivial' problems that in fact are held to be common by many).

Overall a thought-provoking issue! Well done and good luck for the future.

Ian Horwill

100441.3700@compuserve.com

Thanx for the encouragement, Ian! Not sure about your comment regarding Ada – seems to me that even Ada gets the 'subset' treatment to prevent the unwary making mistakes :-) Please note that Ian also contributed a beginner's eye view of copy and assignment – Wait for me! elsewhere in this issue – I would strongly encourage other beginner / intermediate C++ programmers to write articles about their experiences.

Sean,

I much enjoyed *Overload 6* and was interested to read the "Putting Jack in the Box" question from Graham Kendall and the excellent answer to the problem from the Harpist. I thought the Harpist hit the nail on the head by differentiating between object-based and object-oriented programming, and also saying that "one problem with OOP is that you do need to get the design right to start with". I couldn't agree more. But, at the risk of muddying the waters, I wonder if there are further aspects to the "Putting Jack in the Box" problem.

Finding objects to model one's first object interactions is not as easy as might appear. Firstly, you have to design them. That presumably means discovering or inventing abstractions that are relevant to the problem. That in itself brings on a problem – being aware of what you're modelling. In a recent JOOP article [1], Steve Cook and John Daniels sublimely state the obvious when they say that "Software isn't the real world". They go on to explain that when capturing candidate abstractions which are to be the basis of your classes and objects, you're not modelling the real world but your system. In other words, your model represents a deterministic system, not the probabilistic real world.

Selecting the objects to model can be made difficult by the sort of object you choose. The Model-View-Controller paradigm has been around a long time and was recently described by Jim Rumbaugh [2]. This approach suggests that objects of a system are either Model objects (the objects directly traceable to the problem domain), View objects (e.g., the GUI objects) or Controller objects (objects that contain the "rules" of the system). I suggest it's much easier to concentrate on domain objects in your early modelling – and preferably in a problem domain you're comfortable with. If it's banking, try ***Customer*** and ***Account***; if it's traffic management systems, try ***Car***, ***Truck*** and ***Bus***. "Putting Jack in the Box" might prove a little tricky since we're modelling the association of ***Jack***, a domain object, with ***Box***, an interface object. The issue has to be resolved at some stage of course, but maybe later.

So, for people getting to grips with these issues for the first time, perhaps the problem of "Jack in his Box" is soothed by understanding that it's a system you're modelling rather than the real world, and picking domain objects from a domain with which you're comfortable. Hopefully then the object relationships are more tangible, can be modelled and coded more quickly and easily, and convey to the person a sense of satisfaction at progress achieved rather than frustration at thorny issues unresolved.

[1] "Software isn't the real world", Cook and Daniels, Journal of Object-Oriented Programming, vol. 7, no. 2, May 1994, pp22-28.

[2] "Modelling Models and Viewing Views", Jim Rumbaugh, Journal of Object-Oriented Programming, vol. 7, no. 2, May 1994, pp14-19.

Christopher Simons

I agree that identifying the correct objects can be one of the hardest parts of designing a system. I'm reminded of an OOA/D seminar I attended where the presenter gave the example of an oil refinery and showed how the "obvious"

objects (tanks, valves etc) did not give the most flexible design. He then turned the design around so that the connections became the objects – the most important attribute was the topology of the refinery – and this made the model easier to adapt and extend. Very thought provoking!

Dear Sean,

One of most common functions in almost any class is the function that returns the value of a private member variable:

```
class fred
{
public:
       int    getAttribute()
       { return attribute; }
private:
       int    attribute;
}
```

Is there a better way of doing this? I can only assume that there isn't, as all the C++ code I've ever seen is always littered with ***getThis***(), ***getThat***() and ***getTheOther***(). It would be so much more elegant if there were some way of defining a member variable as being private for writing, but public for reading, or even – hold on to your hats ANSI committee – how about allowing the overloading of variable and procedure names:

```
class fred
{
public:
       int    attribute()
       { return attribute; }
private:
       int    attribute;
}
```

Dave Midgley

100117.2522@compuserve.com

I suppose this is why member data often gets an artificial name:

```
class fred
{
public:
       int attribute()
       { return attribute_; }
private:
       int attribute_;
}
```

I don't much care for this (nor any other prefix or suffix convention) but it's probably too late in the standards process to do much about it. I rather like functions to have names of the form "verb" or "verb object" so ***getAttribute()*** *seems fine to me. What do other readers think about this?*

Sean,

Just a letter to thank you and Mike Toms for *Overload 6*. With any luck this letter is appearing in *Overload 7*. My last letter took the slow boat to the letters page, missing an issue and dropping from my memory – I wondered why I agreed with so much of what it said :-)

Thanks to a typo the price of my opinion was cheap: only 1 cent. To make up for this, and also to fall in line with the unfortunate pound for dollar pricing adopted by most companies pedalling their computer wares on both sides of the pond, the opinions here are hopefully worth the full two pennies worth.

I admit that I was a little surprised when I read Francis' EXE article last year on reducing the space and time overhead for returning large objects. I could not see why the method he employed was better than a copy-on-write reference counting technique. As it turns out, when push came to shove neither could Francis, as he revealed in last issue's "Blindspots". Given the bristling armoury / stable / toolbox (depending on your attitude to development) of techniques a competent C++ programmer should possess, blindspots are inevitable.

Handle classes are useful in their place. They are well described in Coplien's C++-must-have, "Advanced C++ Programming Styles and Idioms", and I made use of them in my "Strings Attached" series in CVu. However, some words of warning in case you should get carried away with this technique. It is an optimisation, and hence a measured response to a performance or resource usage problem. Like any other optimisation, it should not affect the correctness of the program's run time.

Multi-threading makes the expression of certain ideas simple, whilst making mincemeat of some previously correct code. For instance, blocking on I/O whilst carrying out a background task is trivial, whilst any use of static data is an open invitation to corrupt data. I have never been a great fan of non-**const** static data: it invites back many of the problems associated with global

data, including the possibility for interrupted write access and thus incoherent state.

When creating a threaded object the initialisation of the thread's members occurs before the thread is actually spawned, hence there are no problems with mutual exclusion. On the other hand, reference counting allows two separate objects to transparently reference the same state. Unless a copy can be forced, e.g., with an ***ensure_unique***() member, these two objects could accidentally end up sharing state between two different threads. If one thread pre-empts the other part way through an operation on the reference counted part to perform its own update, the behaviour of your program will become undefined. This is a classic race condition and will be hard to track down.

Yet if you do not use reference counting this problem will never occur. This is a situation where such an implementation is anything but transparent – the class implementation violates the abstract type. You might suggest making the body part of the object thread-safe, ensuring that each operation on it is a mutex-guarded critical region. Leaving aside the problem of how many mutexes your system has available versus the number of strings you anticipate using, the efficiency loss will be quite dramatic: every access on a fine grained object like a string is locked and unlocked by a pair of system calls. Such heavy use of resources and reduction in performance is not an 'optimisation' in anybody's book!

Another blindspot I found interesting was the use of anonymous enums for constants in C. I have used for class compile time constants in C++, but it was only when teaching someone else some C after doing so much C++ work that I realised, like the Harpist and Francis, it was generally applicable. It was a kind of "aha" moment when I was comparing the two languages – again showing that learning C++ retrospectively improves your understanding of C and how best to use it. Hopefully C9X will sort out some of the shortcomings of **const** in C.

Referring to Graham's letter in the last issue over the use or otherwise of NULL in C++, I saw an interesting post in *comp.std.c++* from Scott Meyers (of "Effective C++" fame) on how to write a user defined null pointer. It went something like

```
class null
{
public:
        template<class type>
              operator type*() const
        { return 0; }
};
const null nil;
```

So any use of ***nil*** in the context of a pointer will return a correctly cast null pointer for that type. This basic class can be elaborated to make ***nil*** behave more like a built-in null pointer. Template members are still not widely supported and so I cannot test this code out. However, I can't say I'm in any hurry to replace 0 with ***nil*** as I am personally not allergic to well defined raw values.

Kevlin Henney

kevlin@wslint.demon.co.uk

The C++ committee have considered some standard form of null pointer like this but there are subtle problems. In the example given, every use of **nil** *relies on a user-defined conversion. Consider the following code:*

```
class A
{
public:
      A(const char*);
};
void f(const A&);
f(0);  // actually f(A(0))
f(nil);       // fails - only one
UDC
      // allowed
```

The only solution to a portable null pointer would appear to be adding a new keyword that behaved 'magically', but could everyone agree on how to spell it?

Your comments about multi-threading code make me wonder whether I could persuade you (or perhaps some other reader) to contribute an article on the pitfalls of writing MT-safe code? I'm sure it would provide food for thought and it is likely to become a very important topic as increasingly more parallel machines appear.

Dear Sean,

Inspired by John Smart's article "A Text Formatting Class" in *Overload 6*, I thought that it was about time that I entered the fray here with some

comments about C++ streams versus C's ***printf*** style output. Let me begin by saying that I have recently made the painful transition from C to C++, and have become a big fan of C++ and object orientation. I am currently engaged in a large C++ project using OOP techniques. Hence I can see that C++ output streams, using the overloaded << operator, are extremely elegant, and I appreciate the type safety that they offer. However, I find C++ streams rather limiting in real world situations (or my version of the real world anyway :-) and I have come to the conclusion that there are many situations in which ***printf*** style formatting offers distinct advantages. Let me explain further.

What I miss with C++ streams is the ability to express the formatting information for a message in a single call to a user defined function with a ***printf*** style signature. The truth is that I very rarely want to send simple formatted output to ***stdout***, which is what the examples in the books tend to show. I find this especially so in the brave new world of visual environments ;-) In the past, in the course of several large C projects, I have made extensive use of functions that take ***printf*** style argument lists, for things like error messages or paginated output. Inside the function, the argument list is decoded using ***<stdarg.h>*** (or the pre-ANSI ***<varargs.h>***), and formatting of the message is done using ***vprintf*** (or ***vsprintf*** or whatever). It can then basically do whatever it wants to with the formatted data. Some of the advantages of this approach that spring to mind are:

1. The actual destination of the message can be encapsulated inside the function, and can be changed without modifying the calling program. Thus for example an error message function can be defined, without the calling program needing to know the actual destination of the messages, which may well change during the evolution of a project.
2. Messages can be routed to more than one destination, e.g., the operator's console and a log file. Again, this can be encapsulated inside the function, and the calling program remains the same.
3. The destination doesn't have to be an actual device, e.g., messages could be sent to a window, or deposited in a memory buffer. The point is that the calling program doesn't need to know any different.
4. The function can manipulate the formatted data on its way to the destination, e.g., it could count newlines and insert a page heading at the appropriate places, or a time stamp could be added to messages.
5. The function can have side effects, e.g., an error function might set an error flag, or perform some other action, as well as outputting an error message.
6. Pointers to functions with a ***printf*** style signature can be passed around to specify where messages should be sent, including functions which do some of the things in items 1 to 5, i.e., not necessarily straight I/O. This is particularly useful in library functions, to avoid embedding application specifics in the library code.
7. Additional parameters can be supplied to the function along with the message formatting information.
8. The function can provide a return value, e.g., the message could be a prompt for a dialogue, with the response being decoded by the function and returned as an enumerated value or a boolean.

Of course, the down side of the above approach is the lack of type safety due to the "..." in the ***printf*** signature. I guess many would consider that an overriding factor, and indeed I am veering towards that view myself. But right now my feeling is that the convenience outweighs the lack of type safety.

Some, but not all, of what I want to achieve could be done with C++ streams if I could set up an ***ostream*** object to which I could direct my messages, but rather than being attached to an output device the formatted data would be delivered to a user defined function, preferably on a line by line basis. I think this could probably be achieved by deriving from the ***streambuf*** class, but this does not seem to be well documented, not for the general user anyway, and would certainly not be easy, plus delving into the internals of the class would make me nervous about portability. I would be interested to hear from anyone who has some ideas on how to do this.

On a slightly different subject, something that I don't like about C++ streams is that the notation tends to become rather verbose when using anything other than the default formatting parameters. Also the behaviour of different formatting

parameters does not seem to be consistent. For example, say I want to output an **unsigned char** as a 2-digit, zero-filled hexadecimal value. In C this is done easily and succinctly using:

```
printf("%02x\n",uc);
```

and everything is fine. In C++ I innocently write:

```
cout << setfill('0') << setw(2) << hex
     << (unsigned int)(uc) << '\n';
```

which appears to work fine except that I suddenly find that *all* subsequent integral values are being output in hex. Yes, I know that the program should set the format back to ***dec*** afterwards (or ideally save the format flags before and restore them afterwards), but my point is that it's not consistent in that ***setfill*** and ***setw*** only remain in effect for the one inserter, whereas the effect of ***hex*** is permanent. And I certainly wouldn't want to have to output too many values in this format using the long-winded C++ notation!

To sum up, I guess the conclusion I have come to is that C++ streams are fine for simple formatted output, but for anything even a little bit complex good old ***printf*** style output seems to come into its own, despite its recognised shortcomings in type safety. On the other hand, having made the transition to C++ and object orientation in most other respects, I feel like maybe I should be using C++ style input / output in new projects, despite everything. What is the status of ***stdio*** and ***printf*** style formatting in the C++ standard anyway? Is it deprecated, or is it even supported at all?

Bob Firth

Troika Associates Limited

firth@troika.demon.co.uk

*The whole of the ISO C library, including the **printf** family, has been incorporated into the draft C++ standard. The committee recently decided to remove **stdiostream**, which was previously intended as a bridge from stdio to streams – **fstream** now does the same job, only better. If it's any consolation Bob, I find streams almost completely impenetrable and would dearly love someone to write a clear and simple article on how to derive new classes from parts of the streams library – any takers?*

And finally, Nicholas Rutland asks of *Overload 6*:

Does 'transitional' always mean 'missing pp6 & 35'?

I'd be interested in the missing pages. Email is fine.

Nicholas Rutland

rutlandn@oldpaul.agw.bt.co.uk

Oh dear! I hope that Nicholas was the only reader whose copy suffered such gremlins...

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!

Phil Shotton asks:

If I was thinking of writing bespoke application software (probably customer databases, maybe also windows programming) would the package Borland C++4.5 and Database Engine 2.0 be good enough? (as an aside, as I'm a registered user I can buy the two for £180 or thereabouts) But I suppose money doesn't really enter into the question, as the initial outlay on a good environment would be benefit by allowing quicker product development.

Phil Shotton

SP134@greenwich.ac.uk

Unfortunately, as Francis notes elsewhere in this issue, this is an almost impossible question to answer! I asked Mike Toms, who knows much more about Borland's products than I do, and his response was "well, you can't answer that question without knowing a lot more about the intended applications – maybe Visual Basic would be suitable?"

Roger Lever asks:

Using BC 4.0 and the STL, as supplied on the previous *Overload* disk, when I tried to add a list item to my code the compiler generated errors in the STL regarding:

- Duplicate definition of ‘max’ and ‘min’
- Incorrect structure operation of pointer in the destructor code

If the exact message is required I can provide that. The point is that STL wouldn’t compile a list template for me. Presumably I need to set options within BC4? Surely I do not need to edit the STL itself?

More generally, are there any examples or documentation of how to use the STL?

Roger Lever

rnl16616@ggr.co.uk

I think I can guess what the problems are as I had similar problems when I first started porting STL to Symantec C++ on the Mac.

STL defines max and min functions as templates. The Symantec compiler also defines max and min so they clash with STL’s definitions. Borland very likely does the same. The ‘solution’ is to comment out the definition of both functions in algobase.h in STL.

The destructor code error is due to code that looks like this:

```
pointer->~T();
```

in defallloc.h (where T is a template parameter). A lot of compilers get this wrong but you can also ‘solve’ this problem by commenting the line out.

Development of STL is still progressing – the version shipped with Overload was current at the time. The most up-to-date version can be obtained by anonymous ***ftp*** *from*

butler.hpl.hp.com

Look in the directory ***stl*** *which contains source and examples (you may need to use the direct IP address instead which is 192.6.19.31, I believe). If you have a Web browser, you can also try:*

http://www.cs.rpi.edu/~musser/stl.html

I will run an article on STL in a future issue. Note that STL will not compile on many compilers as it pushes their support for templates to the limit. Differences between BC4.0, 4.0.1 and 4.0.2 mean your mileage may vary.

Another question I have for *Overload 7*! One problem I ran into was that I would like to have used the syntax of:

```
Device& device = ...;      // screen or
disk
device << "Output:" << obj.output();
```

Device would be a base class which could be invoked polymorphically such that ***output***() would not know where it was actually outputting to. So I tried to derive ***Device*** from a stream, rather unsuccessfully. How do I achieve this?

Given my design approach I would simply employ overloaded **operator<<** as the derivation from stream to ***Device*** would not be ‘proper’ inheritance. However, I am interested in finding an answer to the above problem...

See both Roger’s article (On not mixing it...) and my response to Bob Firth’s letter in this issue.

++puzzle;

Since I didn’t get many questions to answer in this issue, I thought I’d set *you* a little puzzle! The question is “What is the longest sequence of distinct keywords and reserved words possible in a valid C++ program?” To get you started, here is a small example:

```
const volatile unsigned long int x;     // 5 keywords
```

Answers to the editor by May 8th. I may even offer a prize...

Books and Journals

I am still in the process of taking over Mike Toms' editorial contacts with various publishers so it may be some time before I have any books available for review. In the meantime, I would like to see thorough reviews of books that are already on your shelves – books that you come back to, again and again, that you would recommend.

Coming soon!

An exclusive preview of the forthcoming Henricson / Nyquist book "Industrial Strength C++". Following the success of their public domain "Rules and Recommendations: Programming in C++" made available by Ellemtel, Mats Henricson and Erik Nyquist are writing a book for Prentice-Hall that will expand and revise the public domain material. In *Overload 8*, Mats Henricson will tell the story behind the book and explain why it is taking so long...

The C++ Report

This almost monthly journal (it comes out nine times a year) should be compulsory reading for all professional C++ programmers. Regular columns by Scott Meyers, Barton and Nackman, Andrew Koenig, Tom Cargill and others, highlight both the pitfalls and the power of C++. The magazine covers analysis, design and implementation issues with additional features on project management, tool support, ODBMS and a very useful "best of" comp.lang.C++ (otherwise one of the highest noise to signal ratios going). Although it is not cheap – $104 per year for UK subscribers – the information it contains could save you a fortune! For more subscription information, send an email request to:

P00976@psilink.com

News & Product Releases

This section contains information about new products and is mainly contributed by the vendors themselves. If you have an announcement that you feel would be of interest to the readership, please submit it to the Editor for inclusion here.

Programming Research to distribute TestView

This information was taken from QA:News, Programming Research's bi-annual newsletter – Ed.

PR:QA announced their UK distributorship of TestView at the Software Development Show on 22 November 1994 in Birmingham, England. TestView is an automated Graphical User Interface (GUI) Testing Tool. The tool, developed by Radview, is fully client / server aware and operates in a completely Object Oriented manner. Radview, based in Israel, is part of the RAD group who specialise in developing networking tools.

The rapid growth of client / server applications places new demands on software testing and hence distributed testing introduces innovative methods to meet these demands. With the aid of an automated testing tool, testing can be carried out frequently and thoroughly without additional overhead, the software produced is more reliable and of higher quality and the time to market is greatly reduced by eliminating the testing bottleneck.

What does TestView do?

TestView is essentially a record / playback GUI testing tool. Interactions with the application under test are recorded in a maintainable script form to be played back when the application undergoes testing. Explicit tests can be built into the test procedure, ensuring GUI components are present, text fields are correct and even that bitmaps are correct. The script language – Test Management Language (TML) – that TestView uses is remarkably C-like enabling user programming with minimal fuss.

Object-Oriented record, playback and verification describes the user's commands instead of mouse actions. For example, when the OK button is clicked, TestView records the interaction with that object and not the click at screen position X=123, Y=246. Hence if the OK button is moved the test remains valid, as TestView is not adversely affected by the objects position, font or colour etc. That is, unless of course you want to test these attributes...

TestView's client server aspect allows specific client server tests to be remotely executed on multiple client workstations to simulate real life application use. Distributed tests communicate with each other using both synchronous and asynchronous messaging. These remote tests can be controlled and monitored from a single workstation.

Complete test suites can be developed once and reused across multiple software releases and development platforms, saving time and eliminating repetitive labour intensive tasks.

TestView specifically ensures that General Protection faults (GPFs) are successfully handled in a user defined way. When a GPF occurs TestView shuts down gracefully and captures the GPF instead of just crashing. Testing continues even if undesirable or unexpected events occur for example, when running an unattended test, a mail arrives, TestView handles this by clicking on the 'Read Later' button and continues with the test. In fact TestView is a very 'open' system allowing user specific tests to be written and then automatically incorporated into the tool.

Supported environments for TestView are MS Windows 3.1, Windows NT (under development) and an X-Windows (all major UNIX platforms) version which should be available by summer 1995.

Where to go from here?

PR:QA are holding seminars up and down the country throughout the year in order for people to gain a firm understanding of the tool's many benefits. Telephone Nicky Crooks on 01932 888 080 for more details.

Further to the seminar we offer a day's training for interested parties to gain experience of the tool's extensive functionality and ease of use.

Nicky Crooks

nicky_crooks@prqa.co.uk

NoBUG

The Norwegian Borland User Group recently announced their formation on several newsgroups. Their aim is to promote and support the Norwegian community of Borland product users. In addition to Borland C++, the group covers Pascal, Delphi, OWL etc. For more information, send an email request to:

nobug@falcon.no

Credits

Founding Editor

Mike Toms

miketoms@calladin.demon.co.uk

Managing Editor

Sean A. Corfield

sean@corf.demon.co.uk

Production Editor

Alan Lenton

yeti@feddev.demon.co.uk

Advertising

John Washington

john@wash.demon.co.uk

Subscriptions

Dr Pippa Hennessy

pippa@octopull.demon.co.uk

Distribution

Mark Radford

Copyrights and Trademarks

Some articles and other contributions use terms which are either registered trademarks or claimed as such. The use of such terms is intended neither to support nor disparage any trademark claim. On request, we will withdraw all references to a specific trademark and its owner.

The copyright of all material published in *Overload* (except book and product reviews whose copyright is the exclusive property of ACCU) remains with the original author. Except for licences granted to (a) Corporate Members to copy solely for internal distribution (b) members to copy source code for use on their own computers, no material can be copied from Overload without the prior written consent of the copyright holder.

Advertising Rates

Full A4 page £100, 1/2 A4 page £50, 1/4 A4 page £25. Advertising copy to be submitted to the editor with payment (made out to ACCU) by the copy deadline for the issue in which the advert is to appear.

Next Issue

In the June issue, *Software Development in C++* will continue “So you want to be a cOOmpiler writer?” and provide an introduction to the Shlaer-Mellor OOD methodology by David Davies. *The Draft International C++ Standard* will look at Kevlin Henney’s proposal to provide construction-time discrimination of **const**. *C++ Techniques* will continue the discussion of multiple inheritance and Kenneth Jackson will show how to perform fine-grained MFC control validation. The *Vendor Focus* will be on Edison Design Group. The rest is up to you!

Copy deadline

All articles intended for inclusion in *Overload 8* (June) must be submitted to the editor by May 8th.