ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 12

February 1996

Editorial:
Sean A. Corfield
13 Derwent Close
Cove
Farnborough
Hants
GU14 0JT
overload@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

This issue is late. I'd like to be able to put my hand on my heart and say it's my fault. I'd like to be able to say that all the contributions were in by the copy deadline and the only reason I didn't get the magazine to the printers was that I went snowboarding in Alpe D'Huez.

I'd like to...but it simply wouldn't be true. This issue is late because when the copy deadline passed I had received only two contributions. Of course, it *was* xmas and you were all taking a well-earned rest. I was actually working all over xmas – I took just one day off: Friday the 29th. Am I going to rant about the lack of contributions? No, I'm not. Francis, however, says it all in a guest editorial below.

In March, I am attending the next ISO/ANSI C++ meeting. The copy deadline for *Overload 13* is just prior to that meeting. However, I am taking the opportunity of being between contracts to tour California which means that the April issue will most likely be the May issue. Perhaps, in an unprecedented fit of enthusiasm, you can all make sure that the delay is entirely my fault this time? :-)

By the way, I can recommend Alpe D'Huez!

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

We have a problem

guest editorial by Francis Glassborow

Look back over the last year's issues of *Overload*. What do you notice? A very small number of faithful contributors are responsible for well over 80% of the content. Much of this content is highly erudite, well informed and well written by people who either do or could write for commercial publications (and get paid for their efforts). A member's (as opposed to a commercial) magazine (and there is no way that the profile of any ACCU publication fits the term 'newsletter' that some choose to use) needs the core of expertise but it should not be almost exclusively that, otherwise we become a non-commercial immitation of the excellent publications such as *C++ Report* that already exist. Quite apart from anything else, it is unfair on those regular contributors who donate many hours of their time as well as giving you articles that they could sell elsewhere.

A member's magazine is something like a 'fanzine' in that it can and will publish material of a wider range of quality and helps new writers to develop their skills. It is not, or should not be, just a showcase for experts to exhibit their arcane knowledge. That, in my opinion, leads to lazy thinking where the typical reader assumes that it must be right because an 'expert' says so. I frequently get things wrong, sometimes deliberately (there is one blatant error in the current issue of *C Vu* that has only been commented on by two readers so far) and sometimes get things wrong through ignorance or relying on a compiler to refine my understanding (as I recently did in *EXE* magazine). These errors do not worry me because out of them everyone, myself included, can learn.

A member's magazine is a place for controversy, correction of errors and explanation for 'religious' beliefs (Contrast writers who insist that theirs is the one true way to lay out source code with the ones who explain why they choose the layout rules they use. Which is more useful?)

Every reader of *Overload* has something to say or a question to ask that will help someone else by shining a light on some aspect of C++ programming. Paying your £15 C++ SIG subscription should not be the end of your contribution, it is only an enabling fee to provide the mechanism for you to add real value by sharing something with the rest of the readership.

A missed opportunity

Let me get down to specifics. Over 400 of you had a chance to participate in the 'design a date class' competition (It actually had a prize worth more than your annual C++ SIG sub) but only one actually sent in an entry. The criteria for the competition were deliberately set so that anyone above pure novice could compete.

Maybe the subject matter was not that inspiring. Once, many years ago, I attended an evening class on programming in FORTRAN – the only formal course I have ever attended through choice – where the course presenter was a fanatical campanologist whose practical exercises involved writing a program in FORTRAN to print out the changes for a clarion of bells. He did not seem to understand why his students were less than enthusiastic. Those of us who did not need a certificate of completion did not bother to finish the program, much to his frustration.

Maybe I did not make the problem clear enough so that some thought that what was wanted was way beyond their abilities. I suspect that quite a few simply assumed that there would be many much abler readers who would provide them with answers to read and learn

FULL PAGE ADVERT GOES HERE

from in the future. You may be right in believing that you are not very skilled, but you are in excellent company and may be much better than you think. You will never know until you try.

Maybe it was the idea of producing some kind of design document that scared you. One ACCU member has spent six fruitless months trying to find a programmer who understands enough about class design to meet the needs of a job specification. There are far too many C++ syntax ‘experts’ (who know less than they think they do) and far too few class designers. Many, given a design that includes a class definition, can flesh out an implementation. What they cannot do is produce a satisfactory design. What I wanted from you was a class definition with a discussion of why those choices had been made. Elsewhere you will find a contribution from me drifting over some design thoughts (the things that cross your mind as you start to work on a design at this level). No doubt a number of experts will leap in and tell me why that is not the way to do it. Did you ever do Physics at school? If so, I bet you were as heartily sick of the formal write up of experiments as I was. That isn’t the way we arrive at experiments. Maths is even worse, the deductive part only comes after much play (experiment).

Maybe those of you who earn a living from programming were reluctant to put your name over something that you knew was going to be less than perfect lest it damaged your professional reputation. I sympathise and this is one (legitimate) reason for using a pen-name. Within our own community we know that the process of learning includes making mistakes but we do not want that to leak out into the wider world and have it hung round our necks for ever and a day.

So what was your excuse for not taking part? Don’t tell me that it was so easy that it wasn’t worth doing. I have heard that before and it is only when I have persuaded the person to try that they have discovered the hidden problems. If WG21/X3J16 took such a casual attitude to design you would already have your C++ Standard but it would be totally broken.

More generalities

When enquirers ring me up to ask about ACCU they often ask what we have to offer them. I usually, apparently in jest, include in my reply the question ‘What have you to offer us?’ Those who know me well know that this is no joke. Of course we need your subscriptions (we need 500 C++ SIG members to finance *Overload* at the current rate of £15 per year) but that is not where it stops. We are all busy people and finding time to write in-depth, considered articles may be beyond us but a quick bug-report, question about why a piece of code does/does not work, a comment on material in an earlier issue etc. should be within the capability of all. Any time you have had to work to get a piece of code to perform (or even compile) it is worth checking that the final code works the way you think it does and does not have any hidden traps. Even those claiming to be experts get it badly wrong. Two examples for you to consider.

Almost any training course for C++ novices will include an example something like:

```
class Base {
  // something simple
  virtual ~Base ()
  { cout << ”Base destroyed” << endl; }
};

class D : public Base {
  // something simple
  ~D ()
  { cout << ”D destroyed” << endl; }
};

int main(){
  Base* bp;
  bp = new D[10];
// do something
  delete[] bp;
  return 0;
}
```

To demonstrate the need to use **delete[]** rather than **delete**. Perfectly true, it does demonstrate that and it leaves a much more serious defect. Getting the wrong **delete** will, probably, only leak memory; missing the other defect leaves the students with a belief that they can safely do something that will one day disastrously break their code. I’ll leave you to identify the problem.

For almost eighteen months I had the following in my model code for an introductory C++ course (well I have simplified it and put the code in-class to focus on the problem):

```
class Record {
  char* name;
// other private members
public:
  void setname(char* s) {
    delete[] name;
    name = new char[strlen(s)+1];
    strcpy(name, s);
    return;
  }
// rest of definition
};
```

How many of you can spot the fundamental defect in this code? Once again, I am leaving it to you. In case you are wondering, the constructors guarantee that name has been initialised to the NULL pointer before use.

And finally

If you are worried that you may be wrong, why not format your contribution as a question? That is a much better approach than the one of keeping silent. I promise you that the experts will not laugh and most will thank you for writing what they had only thought.

Francis Glassborow
francis@robinton.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++.

My compiler-writing column returns, Francis Glassborow brings us up-to-date on recent PC compiler releases and Alan Griffiths takes a close look at Microsoft's much-fêted new release.

So you want to be a cOOmpiler writer? – part IV

by Sean A. Corfield

Introduction

In the last article I skimmed very briefly over the preprocessor and said that in this issue I would start to look at the type system. For once, I'm actually going to do what I said I would!

The type system

What does the draft say about types? It very conveniently partitions them into different categories that we will model directly. These partitions include:

- integral types
- arithmetic types
- scalar types

An obvious class hierarchy should already be forming in your mind! What about the concept of "type" itself? What questions can we ask of a type?

- size (for **sizeof**)
- name (either for debugging or for **typeid**)
- equality
- promoted type
- ...

A first pass gives us something like:

```
class AbsType
{
public:
  AbsType() { }
  virtual ~AbsType() { }

  virtual size_t         size() const =
0;
  virtual const string&  name() const =
0;
  virtual bool      operator==(const
AbsType&)
                           const =
0;
  virtual AbsType*       promoted()
                           { return
this; }
//...
};
```

The *size* and *name* pure virtuals should be self-explanatory: every concrete derived class must implement these, even if it is just to say "Error: you cannot take the size of a function." for example.

operator== needs more thought because a typical derived class version will look like:

```
bool
CharType::operator==(
       const AbsType& rhs
) const
{
  if (CharType* rhsp =
         dynamic_cast<CharType*>(&rhs))
  {
    // test they are the same char type
  }
  else
  {
    // rhs is not char
    return false;
  }
}
```

We must use RTTI to ensure that the dynamic type of both arguments is the same. The lhs type is known (because the virtual **operator==** despatches through that type) but we must check that the rhs is at least as derived as the lhs (generally the test is that the rhs *is the same type*).

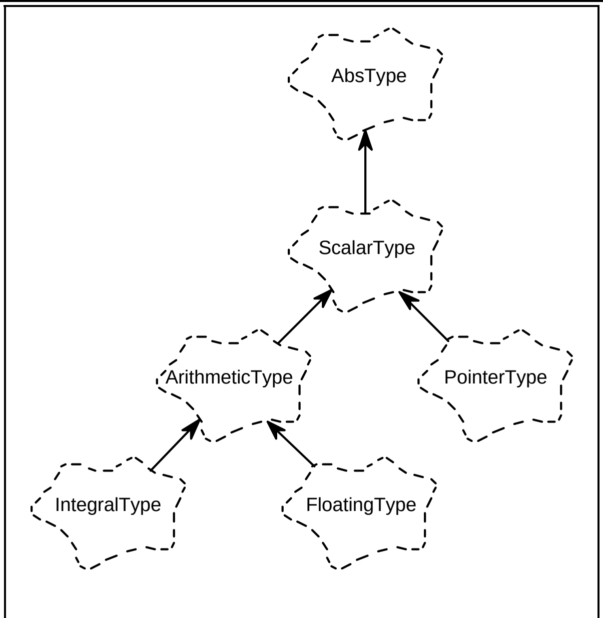

Figure 1: Scalar types

See Uli Breymann's article on this pattern elsewhere in this issue.

What about *promoted*? Why isn't it pure virtual? Because very few types actually promote to anything, it makes sense to provide a default action that "does nothing".

Building blocks

The scalar types form a fairly straightforward hierarchy (figure 1) but some of the other types pose more interesting problems. **class**, **struct** and **union** clearly share some attributes – they all have members, constructors and so on – but they also have differences, especially from the point of view of source code analysis (my original brief for this column). There is another construct in C++ which also has members: **namespace**. Abstracting appropriate classes from this problem is hard. I went through several iterations, discussing the pros and cons of early ideas with Scott Meyers (thanks Scott!) before settling on a four-level hierarchy below *AbsType* (see also figure 2):

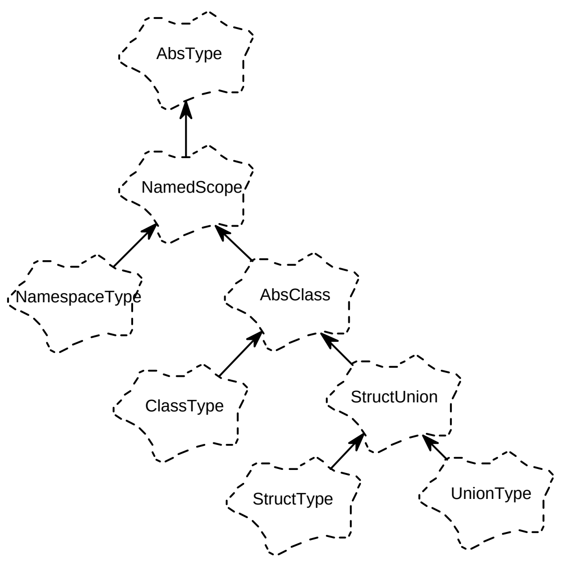

Figure 2: Class types

```
class NamedScope : public AbsType { };
class NamespaceType : public NamedScope
{};
class AbsClass : public NamedScope { };
class ClassType : public AbsClass { };
class StructUnion : public AbsClass { };
class StructType : public StructUnion {
};
class UnionType : public StructUnion {
};
```

Some words of explanation. First of all, **namespace** is not strictly speaking a type. However, handling of declarations is greatly simplified if every declared name can have a type associated with it. Furthermore, when dealing with qualified names, e.g., *X*::*m*, it is unmportant whether the qualifying name is a **class** or a **namespace**.

Why have a separate layer between *AbsClass* and *StructType* (and *UnionType*)? I was designing a source code analyser to check coding standards, amongst other things. Common in coding standards are rules that say things like "treat **struct** and **union** like C, keep C++ features for **class**". In terms of analysis, this means that finding member functions or access specifiers inside a **struct** or **union** should elicit a warning. The code to check the rules in the standards is embodied within methods in the type hierarchy in such a way that checks common to every derived class appear in base classes and differing checks are performed in overriding functions:

```
void StructType::checkRules()
{
        StructUnion::checkRules();
        // other checks
}
```

This pattern is repeated throughout the type class hierarchy, and in fact throughout the entire application.

Mixing in templates

In the original design, template information was held with the declaration and the type system representation stayed "pure". This caused several problems – not the least of which was the fact that *A*<**int**> and *A*<**void***> were both treated as plain old *A*. If this seems a strange decision, and with hindsight it certainly was, some words about the origins of the project are in order. In order to provide an accelerated path to market, the beta release of the product relied on the preprocessor provided on the target platform and templates were not supported. Lack of template support became an issue after a couple of early releases and then had to be grafted on fairly quickly. As compiler support for templates has improved, and especially with the advent of STL, the template support in the analyser needed revising.

Most aspects of an instantiated template class are identical to a non-template class. The template-specific attributes of template **class**es, template **struct**s and template **union**s have something in common so it seems natural to abstract these into a class, *TemplateType*. Clearly a template class must have both *AbsClass* and *TemplateType* as bases. Because of the demands of source code analysis (rather than compilation), it is reasonable to enquire of a type whether or not it is an instantiated template. This leads to the observation that *TemplateType* should be derived from *AbsType* and so we have a mixin diamond – see figure 3. A secondary observation is that this approach makes it easy to support template **namespace**s and **enum**s should either of those become common vendor extensions.

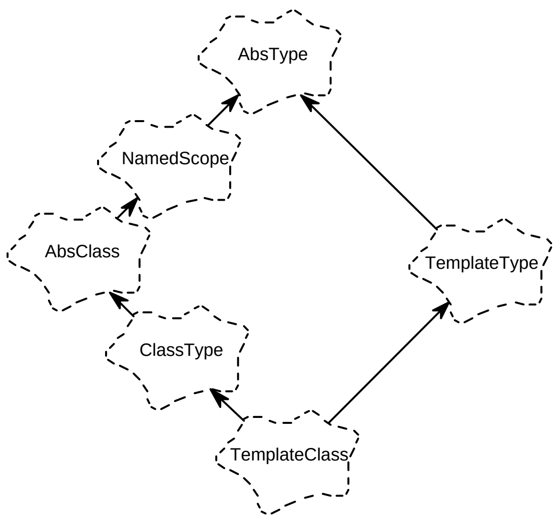

Figure 3: Template types

Next time

I'll leave you to ponder the impact of changing the original hierarchy in this way and next time I'll discuss some of those implications and the difficulties I encountered.

Sean A. Corfield
Object Consultancy Services
ocs@corf.demon.co.uk

Compiler updates

by Francis Glassborow

From the state of my postbag (electronic and snail) it seems that none of you use anything other than compilers for PC based operating systems. I know this isn't the case and I find it sad that those using other hardware assume that reports on their development systems are unnecessary. A couple of years ago I had a telephone call from a software house in Wales. They wanted to know if there was a compiler for C on an Apple Mac. As it happened I could give them an answer but it was then, and would be now, far from the kind of comprehensive run down on the choices available that I can give if asked about PC C/C++ development systems. You may think that nobody would be interested in your choice of development tools – this results in people making ill-informed decisions because they do not know they have a choice.

Compilers for small machines

As far as I know, it is no longer possible to buy the kind of development software that will run on an old machine such as an Amstrad 1640.

Worse still, your choice is pretty limited if you only have a 386 with 4Mbytes of RAM and a couple of hundred megabytes of disk space. Of course the professional will be using something with much more clout but what about the young/old enthusiast or those trying to develop their skills at home. Many of these have to make do with older equipment and now find that C/C++ programming tools want more hardware than they have?

Thoughtlessly I recycled all the disks of earlier versions of C/C++ compilers so I am in no position to help when someone rings up with a problem. Now don't all rush to send me your old, dust-gathering boxes of Borland/Microsoft/Symantec/Whatever C/C++ development tools, but please do not destroy them yet. I will shortly be trying to organise a register of old software so that next time someone comes looking for C on an Atari 800 or whatever I will be able to point them at a source.

Visual C++ 4.0

Finally I managed to put aside a long weekend to dig into this product and give it something of a workout. I booted up Windows NT 3.51 (a much better product than its predecessors) and set about installing it. A day later and after several uninstall/reinstall cycles I was no further forward. Each time I clicked on the Microsoft Developer Studio Icon I got the same, deeply frustrating, application error message: 'instruction at ... referenced memory at ... The memory could not be written.' I finally rang Microsoft's PR people (who more than earn their keep) and they tried to get me some technical support so that my weekend would not be wasted. That was 11am on a Friday, Microsoft did not get back to me. No doubt the problem is something silly but the end result is that they have missed the time slot and you will have to wait till next time. Well, not quite, because one of you emailed me the following:

```
int main() {
        int i=0;
        i->i();
}
```

with the statement that VC++ 4.0, with warnings set to level 4, compiled it without even a murmur. Many of us have the (bad) habit of using a compiler to validate the syntax of our programs. With the ever increasing visual complexity that things such as STL introduce into our source code we certainly need some tool to help us. Not only must a compiler correctly compile our correctly written code, it must not compile nonsense. I can speculate why this problem is happening but until I can get my copy to install and work I cannot explore any further.

The real problem with examples such as this is that they undermine our confidence in our tools. When code does not behave correctly there remains the nagging doubt that the fault is not ours.

Borland C++ and other things

The current version of this is 4.53 (and we have Turbo C++ for Windows 4.5, the DOS version is still 3.1, and is likely to remain so). Borland have also released Code Guard that is supposed to provide some run time checking on memory usage etc. They promised to send me a version but it has yet to arrive. Perhaps they have decided to wait until some of the initial reported problems have been solved.

Version 5.0 is still on the runway, warming up for its launch. As C++ now makes some heavy demands on the skills of implementors I would not be surprised to find that it (like VC++) suffers from some obscure behaviour. What many of those who moan about the time it is taking to get a Standard agreed for C++ fail to realise is that unless the 'corner cases' are sorted out it is impossible to write a lexer and parser that always does what the human programmer expects. Compilers cannot use human insight based on a mixture of context and experience. Going back to the example above, any experienced C++ programmer knows (without having to do a formal analysis) that the code is wrong. The only way that the compiler can know the same is by applying a set of formal, deterministic rules to the code. As these rules beome more and more complex to deal with such problems as template type parameters having to cater for both builtins and user-defined types the potential for wrong answers increases dramatically.

As an aside, I think that the logic of templates is increasingly pushing us towards accepting what many have wanted: make builtin types classes (indeed, I have been heard to suggest even more radical changes such as making all classes template classes, just that some have an empty template parameter list).

As well as working on the continuing development of Delphi (version 2 is due out shortly) and their C++ tools (note that future versions of Delphi are compiled by Delphi, and future versions of Borland C++ will be compiled by themselves), Borland have also made a strong commitment to provide Java development tools. It seems that Borland are returning to their roots – providing high quality, relatively low cost, development tools.

Symantec C++ 7.2(1)

All who are registered owners of 7.0 should now have received their free upgrade to 7.2. Its heavy demand for hardware resources is no more severe than VC++ 2.0 and upwards. If you want to work in mixed 16-bit and 32-bit development this product is a strong competitor for the Borland products. The IDDE takes some getting used to, but it grows on you and provides an excellent working environment.

Now what about that parenthetical (1)? Soon after 7.2 had been released the continuing work on the product resulted in a bundle of further bug-fixes and refinements. These are all bundled into a single 4 megabyte archive of patch tools. It is available by ftp from ftp.symantec.com. This is probably best fetched early on a Sunday morning, after the Americans have gone to bed and before too many Europeans have got up (the site is particularly busy at the time of writing because of Symantec's release of a free virus-tool for Windows NT).

Now once you have upgraded to 7.21 another facility becomes available, Java applet and program development in the same environment. At the time of writing this is only an alpha version heavily based on Sun's beta version of Java. This comes (by ftp from the same site) in a substantial 5 megabyte archive to upgrade 7.21 to 'Expresso' (complete with start-up picture of a steaming cup). Unfortunately, when I followed the instructions everything unpacked happily but with long filenames converted to default 16-bit FAT ones (8.3 style). It may be something that I do not understand about using Windows NT, but nothing I could do would remedy the problem. Fortunately, I already had Sun's beta version with proper (required by Java) long names. Copying that directory tree in to replace the Java tree in SC almost fixed the problem, a bit of a clean up (being careful with project files, which were only in the Symantec version) and I had it all up and running. Symantec say they will shortly fix the name problem (they seem to believe that it will work with FAT style names, all I can say is that I tried and it doesn't on my system).

The upshot is that I have not only a nice C++ development environment, but one that will support my programming in Java in the same environment. It is only an alpha release and so there are some limitations but it gives some of us a head start.

Wrapping up

Notice that in every case I have had to mention inadequacies and too often ones that suggest some complacency within the producing company. If only every company could behave as if it were running second and needed to work hard for first place all might benefit. 'It's good enough' is not good enough. Even 'It's the best' is not enough. Only 'It works, and does what is specified' will satisfy me and nothing less should satisfy you. None of us have time to waste sorting out problems from a sloppy finish.

Now let me hear from you.

Francis Glassborow
francis@robinton.demon.co.uk

Notes on Microsoft Visual C++ V4.0

by Alan Griffiths

At the time of writing most C++ implementations are a fair way from the language described in the ISO draft. I can claim some familiarity with three compilers: Symantec SC72, Borland BC45, and Microsoft VC4. Of these, only one (VC4) supports **namespace** and none of them do templates quite right. (SC72 and BC45 exhibit different sets of problems with templates but are both a lot closer than Microsoft).

I am currently working on a development project which uses the Microsoft compiler and have been recording the problems I have encountered. The following notes illustrate these problems and show the work-arounds I have developed.

Exception handling

The draft language standard defines a hierarchy of exceptions that include *exception* and *bad_alloc* (which is thrown when **new** fails). Although these are documented in the VC4 on-line documentation they are not supported by the run-time library. Even if you were to fix this by modifying the runtime library (if you have a copy of VC2, the files required for their support are provided as an example and appear to work with VC4), the MFC library redefines the behaviour of **new** to throw a pointer to an MFC exception class *CMemoryException*.

This an issue if you wish to develop portable code – on some platforms you need to deal with *bad_exception* and on some with *CMemoryException******. (Some platforms throw *xalloc*, but that can be dealt with by a **typedef**.) This is not a place for a critique of the MFC library design, but it should be obvious that libraries should not modify the behaviour of global entities (such as the **new** operator).

Implicit type conversions

```
class Thing;
class ThingHandle {
public:
  ThingHandle(Thing* pt = 0) : rep(pt)
{}
  ThingHandle&
    operator=(const ThingHandle& h)
        { rep = h.rep; return *this; }

private:
  Thing* rep;
};

void f() {
  ThingHandle h;
  // ...
  h = 0;  // VC4 "cannot convert from
          // 'const int' to
'ThingHandle'
}
```

According to the standard **0** may be converted to a *ThingHandle* by the constructor and the assignment operator used. According to the on-line help, this inability to treat a **0** as a pointer is a change made for conformance to the "draft ANSI standard" – this apparently spurious claim is made for a number of the problems discussed in these notes.

Templates and nested classes

It took me a long time to work out the name binding rules for template instantiation applied by the Microsoft compiler. In C++ they are confusing enough: a name is either bound in the scope of its use in the template declaration (if it does not depend on the template parameter) or in the scope of the template instantiation. In VC4 the latter is replaced by the global scope at the point of the template instantiation. This affects the following code:

```
#include <vector.h>
// fix for MSVC++:
#if defined(_MSC_VER) && (_MSC_VER <=
1000)
  template<class T> class Class_Value {
  public:
    Class_Value() {}
    Class_Value(const T& t) : v(t) {}

  private:
    T v;
  };
#endif

template<class T> class Class {
public:
// correct code for other compilers:
#if !(defined(_MSC_VER) &&        \
                    (_MSC_VER <= 1000))
  class Value {
  public:
    Value() {}
    Value(const T& t) : v(t) {}

  private:
    T v;
  };
#else
  typedef ::Class_Value<T> Value;
#endif
  Value f(const T& t) const { return t;
}
private:
  vector<Value> array;
  // without the fix, VC4 says:
vector.cpp(90) : error C2065: 'Value' :
undeclared identifer
};
```

Templates and namespaces

There are a number of areas in the draft standard that are far from clear; those dealing with templates and their interaction with namespaces are amongst them. I cannot claim therefore that the following code conforms (although I consider that it's OK and any possible problem lies in the wording of the draft – from his comments I think Sean agrees).

```
#include <vector.h>
namespace MyNameSpace {
  template<class T> class Element {
  public: T t;
  };

  template<class T> class Container {
  public:
    vector<MyNameSpace::Element<T> >
array;
  };
}

// fix for MSVC++:
#if defined(_MSC_VER) && (_MSC_VER <=
1000)
  using MyNameSpace::Element<int>;
#endif
typedef MyNameSpace::Element<int>

MyIntElement;
typedef MyNameSpace::Container<int>

MyIntContainer;
int main() {
  MyIntContainer collection;
  MyIntElement   e;
  e.t = 1;
  collection.array.push_back(e);
  return 0;
}
```

Without the *using-declaration*, VC4 says:

```
name.cpp(53) : error C2065:
'Element<int>' : undeclared identifier
```

Throw or return

The Microsoft compiler does not believe that **throw** terminates a function, thus following every **throw** there needs to be a **return** that provides some spurious value. This requirement is not always easy to accomodate, for example:

```
MDataSourceManager&
MDataSourceManager::theInstance()
{
  throw MX::NotImplemented(__FILE__,
                           __LINE__);
}

testfreq.cpp(725) : error C2561:
'theInstance' : function must return a
value
```

Covariant return types

A long time ago the C++ rules for overloading functions were changed so that, given suitable "conformance" requirements the return type could differ (an example of using this feature is shown below). This change has not found is way into VC4.

The idea is that code which has a *Derived* pointer may use the return value from *makeClone* directly as a *Derived* pointer (without requiring a downcast).

```
class Base {
public:
  virtual Base* makeClone();
};


class Derived : public Base {
public:
  // this should be legal:
  virtual Derived* makeClone();
};

testfreq.cpp(394) : error C2555:
'MTestColumnDetails::makeClone' :
overriding virtual function differs from
'MColumnDetails::makeClone' only by
return type or calling convention
```

Alan Griffiths
Senior Systems Consultant
CCN Group Limited
agriffiths@ma.ccngroup.com

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 the absence of an international meeting since the last issue, I focus closer to home on the work of IST/5/-/21.

A UK perspective

by Sean A. Corfield

I've mentioned in several preceding columns the schedule for standardisation of C++. We are currently in the process of resolving National Body comments from the first Committee Draft Ballot. In March, the joint committee meets in Santa Cruz to complete resolution of that first ballot and produce the document that will go forward as the second Committee Draft for balloting during the middle of 1996. This meeting will be hosted by Borland.

The UK C++ panel, IST/5/-/21, continues to meet every couple of months to discuss issues within the draft with which we are unhappy. So far, the panel have produced a database of several hundred issues from a review of clauses 1-12 of the first Committee Draft. It's a mammoth job. In order to complete the review of clauses 13 (Overloading) to 27 (Input/output library), the panel have allocated one or two clauses each to reviewers who have volunteered to go through them with a fine-toothed comb.

Sticklers

The UK have a reputation for being sticklers for detail where standards are concerned and, although some members of the joint committee find our approach unnecessarily pedantic, there are many people who are pleased that someone is willing to dot the 'i's and cross the 't's.

Progress has generally been very good on the UK issues. Many issues are editorial – fine wordsmithing – and therefore non-controversial. Most of the technical issues have been taken up by one or other of the technical working groups within the joint committee. A couple of terminological issues are proving more difficult to resolve:

- linkage – C introduced external and internal linkage purely because it didn't have a proper mechanism to partition the global name space. C++ has **namespace**, precisely for this purpose, which renders much of the linkage terminology (inherited from C) as excess baggage. The UK are investigating how a rewrite of the relevant clauses of the draft to reflect this improvement over C would look;
- lvalue/rvalue – these terms had relatively clear meanings in C (and other languages) but C++ introduces object-rvalues which have many of the properties of lvalues. The UK has long felt that the draft could be made clearer by introducing a third term (e.g., "ovalue") to describe this hybrid. Again, the UK are investigating exactly what impact on the draft such a change of terminology would have.

In both cases, the UK position has support within the joint committee but the concern is that it may be too late to make such changes. The UK panel has been asked to do the analysis and write up the changes as formal proposals to ease the workload of the joint committee.

Public reviews

For some countries, notably the USA, the CD ballot signifies a public review. ANSI received many public comments on the draft during 1995. A second CD ballot will mean a second public review for ANSI with, hopefully, lots more constructive comments from their public. The rules in the UK are somewhat different: a public review is conducted only once the document reaches the Draft International Standard stage. However, the UK panel are keen to collect comments on the draft at any stage and welcome active new members. For more details about joining the UK panel, contact the convenor, Richard DeMorgan **mailto:demorgan@parallax.co.uk**, or if you want to discuss technical issues contact myself, Francis or Steve Rumsby **mailto:steve@maths.warwick.ac.uk** – Steve is the maintainer of the UK C++ information web site **http://www.maths.warwick.ac.uk/c++**

Sean A. Corfield
Technical Director
Object Consultancy Services
ocs@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.

Ulrich Breymann shows how RTTI provides an elegant solution to a common problem, Kevlin Henney looks at how to generalise a simple transformation and continues his series on template techniques, and Roger Lever rounds off his development of debugging classes.

An implementation pattern using RTTI

by Uli Breymann

Run-time type information (usually abbreviated to RTTI) has been available in C++ for some time. There are applications where RTTI allows a much more elegant design and has additional advantages from an object-oriented viewpoint. This is shown here by a simple example.

What is Run-Time Type Information (RTTI)?

The extension to add the capability of run time type information to C++ was proposed by Stroustrup in 1991 and adopted by the Standards Committee March 1993 [1]. By means of the operators **typeid** and **dynamic_cast** is it possible to determine the polymorphic type of an object at run time. Polymorph objects are represented in C++ by pointers and by references. Example:

```
class Base { /* ... */ };
class Derived : public Base { /* ... */
};

Base *p1, *p2;
p1 = new Base;
if (some_special_runtime_condition)
  p2 = new Derived;
else
  p2 = new Base;
```

The type of the two pointers is *Base**, however, we are interested in the type of the objects (**p1*, **p2*) they point to. The * operation (dereferencing) yields a reference to the object to which the pointer is pointing. The static type of ***p1** and ***p2** is *Base*, as well as the polymorphic type of ***p1**. However, the polymorphic type of ***p2** depends on some run-time condition and may possibly be *Derived*. It cannot be determined at compile time.

Before coming to the main subject of the article, I will explain in short the **dynamic_cast** operator and **typeid()**.

The dynamic_cast operator

The RTTI mechanism allows a safe type cast from a base class to a specialised (derived) class (downcast). Polymorphic behavior is assumed, i.e., inheritance and dynamic or late binding. This holds for pointers as well as for references.

```
class Base { /* ... */ };
class Derived : public Base { /* ... */
};

Base    *p = new Derived;
Derived *pA;
pA = p;                          //
error!
pA = dynamic_cast<Derived*>(p); //  ok!
```

The type cast is safe because it is checked at run-time whether the pointer *p* is connected to an object of type *Derived*. In that case the address of the object is returned, otherwise 0 is returned.

Unlike pointers, references cannot have undefined values. If the argument of **dynamic_cast** is not of the same or derived type, **dynamic_cast** will throw an exception of type *bad_cast*.

```
Base BaseObj;
Derived DerivedObj;
Derived& X =                    // ok!

dynamic_cast<Derived&>(DerivedObj);
Derived& Y =                    //
exception!
        dynamic_cast<Derived&>(BaseObj);
```

The typeid operator

The **typeid** operator returns an object of type *type_info*. That is the reason why *typeinfo.h* has to be included. The argument of **typeid()** can be an object or a class. However, the static type of the argument is not of importance, but its polymorphic type at run-time:

```
void f(const Base& X)
{
  if (typeid(Derived) == typeid(X))
    cout << "X is of polymorphic type"
         << "'Derived'";
  else ...
}
```

An implementation pattern

Here we show how the operators **typeid** and **dynamic_cast**, which had been invented for RTTI, work in a typical example. In this example a pattern is used which can easily be modified for different purposes. The pattern is suitable for binary member functions having polymorphic parameters, whose execution makes sense only if caller and parameters are of the same type. Binary means that one parameter is necessary in addition to the calling object. Often the type of polymorphic objects can be determined at run-time only.

The pattern can be used in virtual member functions (methods) of a class. It is well applicable to implement the CHAIN OF RESPONSIBILITY pattern [2]. We will use the property of the **dynamic_cast** operator of throwing an exception in case of “wrong” types (listing 1).

```
Void
Derived::binaryMethod(const Base& param)
{
  try
  {
    const Derived& X =
       dynamic_cast<const
Derived&>(param);
    // Here goes code working with  X
and
    // the object which called this
    // function. This code is specific
for
    // Derived objects.
    // ...
  }

  catch(bad_cast)
  {
    // ... do nothing or error message
  }
}
```

Pattern of a method (listing 1)

Typical candidates for methods of this kind are the copy assignment **operator=()** (see [4]) and the relational operators, e.g., **operator==()**, but you can think of other functions. We choose the boolean equality operator for simplicity. Suppose there is a class *Base* from which two classes *A* and *B* are derived. Classes *A* and *B* differ by the number and the values of their private data which can be made visible by calling the method *show***()** (listing 2).

```
#define bool int
#define true 1
#define false 0
#include <iostream.h>

class Base
{
public:
  // pure virtual functions
  virtual bool operator==(const Base&)
                               const =
0;
  virtual void show() const = 0;
};

class A : public Base
{
public:
  A(int i)
  : AValue(i) {};
  virtual bool operator==(const Base&)

const;
  virtual void show() const
  {
    cout << "A: " << AValue << endl;
  }
private: int AValue;
};

class B : public Base
{
public:
  B(int i1, int i2)
  : B1Value(i1), B2Value(i2) {};
  virtual bool operator==(const Base&)

const;
  virtual void show() const
  {
    cout << "B: " << B1Value << '\t'
         << B2Value << endl;
  }
private: int B1Value, B2Value;
};
```

Declaration of classes *Base*, *A* and *B* (listing 2)

The implementation of the equality operator assumes that two objects are not equal anyway if they differ in type. Therefore error handling is not required in the catch clause (listing 3).

```
#Include <typeinfo.h>
// translate ANSI/ISO C++ : bad_cast
//  to Borland C++ 4.5    : Bad_cast
#define bad_cast Bad_cast

bool
A::operator==(const Base& object) const
{
  try
  {
    const A& compareWith =
            dynamic_cast<const
A&>(object);
    // comparison makes sense for class
A
    // objects only
    return AValue == compareWith.AValue;
  }

  catch(bad_cast)
  {
    return false;
  }
}

bool
B::operator==(const Base& object) const
{
  try
  {
    const B& compareWith =
            dynamic_cast<const
B&>(object);
    // comparison makes sense for class
B
    // objects only
    return B1Value ==
compareWith.B1Value
        && B2Value ==
compareWith.B2Value;
  }

  catch(bad_cast)
  {
    return false;
  }
}
```

Implementation of **operator==()** (listing 3)

The common base class makes it possible to manage heterogenous, dynamic *A* and *B* objects by means of a container if the container elements are pointers of type *Base** which point to the objects. This is quite a common way to do this, e.g., a group of graphical objects in a CAD system (computer aided design). Listing 4 shows the possibilities to get run-time type information. First a container taking *Base** objects is declared. Then the container is partly filled and its contents shown. Instead of taking the *vector* template you can use a normal C array.

```
#include ... // (class declaration of  A
             // and B)
#include <vector.t>
    // vector class template (see e.g.
[3])

void showContainer(const
Vector<Base*>&);
            // see below

void deleteElement(
               Vector<Base*>&, const
Base&);
            // see below

int main()
{
     Vector<Base*> Container(10);
     Container.init(0);

     // fill Container with 10 different
     // elements
     int i;
     for(i=0;  i< 5; i++ )
       Container[i] = new A(i);
     for(; i< 10; i++ )
       Container[i] = new B(1,i);
     showContainer(Container);

     cout << "show B objects only" <<
endl;
     for(i = 0; i < Container.size();
i++)
         if(Container[i]
            && typeid(*Container[i]) ==

typeid(B))
             Container[i]->show();
     cin.get();
     A anA(3);
     cout << "look for A(3) "
```

```
          << "and remove from container"
          << endl;
     deleteElement(Container, anA);
     B aB(1, 8);
     cout << "look for B(1,8) "
          << "and remove from container"
          << endl;
     deleteElement(Container, aB);
     showContainer(Container);
}

void showContainer(const Vector<Base*>&
V)
{
     for (int i = 0; i < V.size(); i++)
         if(V[i]) V[i]->show();
}

void deleteElement(
    Vector<Base*>& V, const Base& what)
{
     for(int i = 0; i < V.size(); i++)
       if(V[i] && *V[i] == what)
                  // polymorphic call of
==
       {
           delete V[i];
           V[i] = 0;
       }
}
```

main() shows the possibilities of run-time type information (listing 4)

Operator **typeid** comes into play if we want access only to objects of a certain type. In our example only objects of class *B* are shown on standard output.

The following lines show how objects of distinct types can be removed from the container by the method *deleteElement*() which expects a parameter of type **const** *Base***&** for the object to be compared. Therefore we can pass all kinds of objects to the method provided that they are of type *Base* or derived from *Base*.

The line

```
  *V[i] == what
```

is to be understood as follows: *V*[*i*] is a pointer to an object of type *A* or *B* (in our example). Consequently ******V*[*i*] is a reference to the object, by which the equality operator for exactly this object is called, thanks to the virtual mechanism:

```
  (*V[i]).operator==(what);
```

Advantages and disadvantages of RTTI

Clearly a disadvantage is the possibility that an error may not be detected until the execution of a program. Type checking at compile time is generally preferred. Sometimes, however, type checking at run-time allows much more elegant solutions which are nevertheless safe, as shown here. In our example with objects of heterogenous types the compiler guarantees that all container elements are derived from *Base*, and the check at run time within the equality operator yields correctly either **true** or **false**.

The advantages become clear if we think of realizing the example without **typeid()** and **dynamic_cast**. A special type management would be necessary which is outlined here.

First the class *Base* and all classes derived from it need a virtual function *whichType*() which returns the object's type, which can be coded as an enumeration (**enum** *objectType* **{** *AType***,** *BType* **}**). Hence the possible types have to be known when the base class is written.

In the second place it would not be possible to use virtual equality operators, because virtual methods have identical parameter lists, and a downcast is not feasible. Therefore the equality operators must have a parameter of class type, for example for class *A***: operator==(const** *A***&)**.

Yet to check equality with **operator==**, one can think of a global operator (listing 5):

```
// in base.h
enum objectType {AType, BType};

// global function as an alternative for
// typeid()
objectType whichType(const Base& X)
{
    return X.whichType();
           // polymorphic call of
whichType
}

bool operator==(
    const Base& lhs, const Base& rhs)
{
  if(lhs.whichType() != rhs.whichType())
    return false; // definitively not
equal
  switch(lhs.whichType())
  {
  case AType:
    return ((A&) lhs == (A&) rhs);

  case BType:
    return ((B&) lhs == (B&) rhs);

  default:
    cerr << "unknown type!" << endl;
  }
  return false;
}
```

Part of necessary modifications if there is no RTTI (listing 5)

The global function *whichType*() calls the correct method having the same name by means of the virtual mechanism. According to the determined type the correct equality operator is called for the object. The type casts to *A***&** and *B***&** are ugly, but safe because of interrogating the type before.

Of course, *main***()** has to be modified, too. The disadvantages compared to using RTTI are summarized here:

- The type of an object has to be known in advance. Above all, extension by additional classes derived from *Base* is possible only if the enumeration type *objectType* is also extended. For that purpose a file belonging to the base class has to be modified!
- The global **operator==()** has to be supplemented each time another class is added.
- The application program (here *main***()**) has to be modified possibly at many places.

An extension by additional classes derived from *Base* therefore entails changes at many places and thus involves the risk of inconsistencies. Using run-time type information makes all this superfluous.

Using RTTI places only one requirement upon subsequently derived classes: the adherence to the interface of the **virtual operator==(const** *Base***&)**.

The way of applying RTTI shown above can be used for all methods which have a parameter of their own class (dynamic type) and which are to be used in a polymorphic manner.

Dr Ulrich Breymann
breymann@alf.zfn.uni-bremen.de

References

[1] Bjarne Stroustrup: The Design and Evolution of C++, Addison-Wesley 1994

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Addison-Wesley 1994

[3] Ulrich Breymann, Nigel Hughes: Composite Templates and Inheritance, *C++ Report 7*(7), September 1995, pp. 32 - 40, 76

[4] Ulrich Breymann: A deeper look at copy assignment, *Overload 11*/1995

[5] Stan Lippman: Pointers versus References. *C++ Report 6*(6), July 1994, pp. 42 - 45

Rot in L

by Kevlin Henney

The *rot13* encryption algorithm is a simple but effective method for obfuscating text against casual reading – it is not even remotely secure, so you won't win money from Netscape for discovering this! Its principle application is for encoding text placed in a public place that might otherwise be found offensive by others, for example in a post to a newsgroup. The reader must make a conscious effort to decode it.

The cutest feature of *rot13* is that applying it twice to a piece of text is the identity operation, i.e., the encode and decode algorithm are one and the same. The English 26 letter alphabet is used and you simply rotate each letter in the text through it by 13 places. All other characters are left as is – in spite of Asterix et al's best efforts, it is hard to offend with only punctuation and digits at your disposal.

Assuming a character set that supports an ordered, continuous alphabet encoding, here is a little map function that does the job for us:

```
char rot13(char value)
{
  return unsigned(value - 'A') < 26
       ? 'A' + (value - 'A' + 13) % 26 :
         unsigned(value - 'a') < 26
         ? 'a' + (value - 'a' + 13) % 26
:
           value;
}
```

In a future */tmp/late/** column I will be covering value constraint techniques that can be applied here to check that this particular implementation will not accidentally be compiled on a platform using something like EBCDIC. Before some of you put finger to keyboard: no, using the preprocessor is not the correct solution.

For a given character set you could write a more efficient implementation using array look up, i.e., a predefined array of encoded character codes looked up on the unencoded character. Typing this table out is tedious for ASCII, but I would humbly suggest that any character set larger than this is better handled using the code above. Oh, and if you do use table look up don't forget to cast to **unsigned char** to index the array.

So what can we encode? Obviously a **char** to a **char**. So how about a *string* to a *string*:

```
string rot13(const string &source)
{
  string result(source);
  for(size_t at = 0;
      at < result.size();
      ++at)
    result[at] = rot13(result[at]);
  return result;
}
```

Should we be returning by value? Or should we be changing the *string* in place:

```
void rot13(string &result)
{
  for(size_t at = 0;
      at< result.size();
      ++at)
    result[at] = rot13(result[at]);
}
```

Overloading on both of these is an exceptionally bad idea, creating the kind of confusion George Wendle was talking about in *Overloading on const is wrong, Overload 6*.

What about raw C strings? Lists of **char**? Streams? The list is open ended, suggesting that a new *rot13* function overloaded for each new type is not the way to go. The solution is to build on the general algorithm and container framework of the STL – the “L” in the title of this article (and indeed its motivation).

To transform a *string* in place:

```
transform(
    for_encode.begin(),
for_encode.end(),
    for_encode.begin(), ptr_fun(rot13));
```

Into another *string* of sufficient size:

```
transform(
    unencoded.begin(), unencoded.end(),
    encoded.begin(), ptr_fun(rot13));
```

Over an existing array:

```
transform(
    char_array,
    char_array + sizeof char_array,
    char_array, ptr_fun(rot13));
```

You get the idea. Now some words of explanation. The *transform* template function takes two iterators that define the sequential range of the input. They refer to the first and one past the last elements. These are input iterators which may be used for single pass algorithms. They must support at least **operator*** for reading and **operator++** operations [see *Seduction: The Last?* in *Overload 9*]. The third argument to *transform* is an output iterator to where the output characters are written, and which is incremented after each value is assigned.

The final argument is the transforming function – or, more accurately, functional object. The *ptr_fun* function relies on template type deduction to take a conventional function pointer and return an object of *pointer_to_unary_function*<**char, char**>. The *pointer_to_unary_function* class is an adaptor class enabling function pointers to be used with STL algorithms that use functional objects. As it appears in the standard:

```
template<class Arg, class Result>
class pointer_to_unary_function :
    public unary_function<Arg, Result>
{
public:
    explicit pointer_to_unary_function(
                              Result
(*)(Arg));
    Result operator()(const Arg &)
const;
};
```

The base class is merely a non-polymorphic place holder:

```
template<class Arg, class Result>
struct unary_function
{
    typedef Arg argument_type;
    typedef Result result_type;
};
```

See */tmp/late/** in this issue for some comments on template parameter style – in this respect the working paper for the standard could do better.

Finally, a program of use. Iterators are available on IOStream objects, so the following gives you a program that allows you to *rot13* the standard input to the standard output.

```
int main()
{
    transform(
        istream_iterator<char>(cin),
        istream_iterator<char>(),
        ostream_iterator<char>(cout),
        ptr_fun(rot13));
    return 0;
}
```

An example of use would be on UNIX to email a conundrum to someone with the solution appended in *rot13*:

```
( cat problem;
  echo 'solution:';
  rot13 < solution ) |
    mail -s'conundrum' someone
```

Kevlin Henney
kevlin@two-sdg.demon.co.uk

Simple classes for debugging in C++ – Part 3

by Roger Lever

Part 2 covered quite a bit of ground such as using macro magic, the canonical class form, virtual destructors, new and delete operators and collections. *RNLI* is in serious danger of becoming useful!

Still to be completed in Part 3:

- Provide some heap walking capability to “see” what’s in memory
- Provide some random check capability within ‘main’
- Output debugging information to a file

Heapwalking

This conjures images of tightrope walking and in some senses is probably more dangerous! Within *RNLI* the hard work has already been done – collecting the heap items into a list. Consequently, all that needs to be done is to add the following declarations:

```
class RNLI {
public: // as before ...
  void showHeap(void) const;
  void showStack(void) const;
  void showMemory(void) const;
};
```

To walk through the items in the Heap list:

```
void RNLI::showHeap(void) const {
  RNLI* nextItem;
  for(RNLI* ptr = rnliHeap;
      ptr; ptr = nextItem) {
    cout << “Heap object at “ << ptr
         << endl;
    nextItem = ptr->next;
  }
}
```

Clearly *showStack***()** would be very similar and *showMemory***()** would simply call both of these functions. Visions of extending this to include named objects, sizes or maybe garbage collection or memory compaction are premature – remember that some simplifying assumptions were made to ease building *RNLI*. For those who missed Part 2, one important assumption was that the global **new** operator was not being overloaded for another class.

To see the output would require *main***()** to include:

```
  ptrD->showMemory();
```

Giving the following output (your addresses may be different):

```
Heap object at 0x1732
Stack object at 0xffee
```

Of course, seeing pointer addresses is perhaps not the easiest way of summarising the contents of the Heap. It would be simple to add a static count to help identify any mismatch in numbers of objects.

More macro magic

Part 2 outlined the technique for using macro magic to either add or remove *RNLI* from the final code based on the value of a flag variable. This can be extended to pepper the runtime code with random checks. Using this approach satisfies a need to actually provide status *CHECK*s of code in *main***()**. This can be achieved by:

```
#ifdef CHECK_ON
  #define USE_CHECK : public RNLI
  #define CHECK_MEMORY(aPtr)
       \
                      (aPtr->showMemory())
#else
  #define USE_CHECK
  #define CHECK_MEMORY(aPtr) ((void)0)
#endif
```

Within *main***()** the *ptrD->showMemory***()** call can be replaced with:

```
  CHECK_MEMORY(ptrD);
```

This mechanism now enables *RNLI*’s dominion to extend outside of its own class declaration and definition and provide random checks of the state of memory. However, it could be improved:

```
 #define CHECK(aPtr) (aPtr->isValid() ?
\
     (void)0 : (aPtr->dumpMemory())
```

This macro expands to check the pointer using the *isValid***()** capability. If *isValid***()** returns OK the program continues, if it fails the program outputs memory and exits with the error line and module name. The *dumpMemory***()** would *showMemory***()** and then do whatever cleanup was necessary before exiting the program. Kinder souls could provide diagnostic messages and allow the program to struggle on.

Naturally these random checks could be used within *Base* and *Derived*, however, some sort of control needs to be exercised. It would be somewhat farcical if the source contained more debug code than code for instance!

Output to a file

Onto the last milestone! *RNLI* currently outputs to the screen, not the best place to output debug information unless the program has just crashed. The usual place to put it is in a file and an obvious implementation is:

```
class RNLI { // file: rnli.h
  // as before
private:
   static Out2Disk* output;
};

class Out2Disk { // file: 2diskcpp.h
public:
  Out2Disk(char* filename = "dump.txt")
  { out.open(filename, ios::app); }
  // append, may be useful for
comparison
  // to a previous session
  ~Out2Disk(void)
  { out.close();}
  void dump(const char* from,
            void* str, size_t size)
  { out << from << " is at "
        << str << " size " << size
        << " bytes" << endl; }
  // Other overloaded dump functions
private:
  ofstream out;
};
```

The class name was chosen to show the write only operation that *RNLI* performs and to give no indication of being able to read from files. The designed intent is that *RNLI* only write out debugging information – it cannot read it back in or parse it for analysis. The pointer is used to show that the class declaration does not *contain* the *Out2Disk* object but *uses* it. This also provides a level of freedom regarding the implementation of *Out2Disk*.

This uses a technique that Stroustrup describes as *initialisation is resource acquisition*. Obtaining the resource as an object which enables the exploitation of the C++ destructor to remove the object at the appropriate time, releasing that resource safely. This is very simple and powerful and is the foundation of many other techniques.

Paradise Lost

The original C++ version of *Out2Disk* was not closing the file properly: If the program encountered an error the file was very likely to disappear. Changing the static initialisation to use the *cerr* stream instead of a file gave a clue as to what was happening. The output would simply stop in the middle of printing a sentence. It appeared that during an 'Abnormal program termination' the stream was not guaranteed to complete its operation before it too was destructed. It was as a result of this that I started to use a C version instead:

```
class Out2Disk {
public:
  Out2Disk(char* filename = "dump.txt")
  { out = fopen(filename, "a");
    assert(out); }
  ~Out2Disk(void)
  { fclose(out); }
  void dump(const char* astring)
  { fprintf(out, "%s\n", astring); }
private:
  FILE* out;
  // Prevent automatic assignment
operator
  // and copy constructor
  Out2Disk& operator=(const Out2Disk&
o);
  Out2Disk(const Out2Disk& o);
};
```

This did complete correctly during an abnormal crash. For practical purposes the *Out2Disk* class is the same since the interface remains constant. The implementation is very similar except it uses the *stdio* equivalents of *iostream*.

Paradise Regained

Mixing C and C++ like this was not satisfactory and prompted more investigation...

Checking the details on the lifetime of static objects in C++ FAQ I came across a mechanism which indirectly dealt with the problem. The crux of the matter is to treat the file as a local resource which is destroyed before the class or file scope static objects. This can be done by replacing the declaration with a function which returns a reference to a local scope static object that is contained inside the functions. The declaration therefore changes:

```
  static Out2Disk* output;
                               // old
declaration
  static Out2Disk& output();
                               // new
declaration
```

Comparing the two implementation styles:

```
  output->dump("Output string")
                 // static Out2Disk*
output
  output().dump("Output string")
                 // static Out2Disk&
output()
```

Using this technique resolved the problem of the stream disappearing prematurely and enabled the use of the iostream version of *Out2Disk*! Now when the program crashed the stream would complete its operation to the disk file and then clean up the rest of the objects before exiting completely. This was considerably more satisfying than mixing C and C++.

Hmm, anyone care to hazard a guess why Roger found this to be the case? It sounds very compiler specific to me as I would not expect a stdio file to flush correctly when a program crashed, nor would I expect local static objects to be correctly destroyed in order. BTW, the functions-for-static-objects technique is highly recommended since it *avoids all sorts of nasty order-of-initialisation problems – Ed.*

What was not done in RNLI

Now that *RNLI* is virtually complete in terms of its original objectives it would be useful to make a few comments about what was not included and why not.

No other constructors are provided for *RNLI*, the obvious other one being a named object i.e., *RNLI***(char*** *namedObject***)** or perhaps *RNLI***(***string namedObject***)**. Currently the class *Base* is derived from *RNLI* and since *Base* does not know about *RNLI* the default constructor of *RNLI* is used. To pass a name to *RNLI* the *Base* class would need to know about *RNLI* and use an initialisation list to pass that name. To hide the use of name from *Base* would require that the macro box of tricks is used again as a substitution mechanism like *USE_CHECK*. However, what about *Derived*? How is a name passed from *Derived* to *RNLI* via *Base*? The level of complexity of these macros would multiply out of control.

There are no protected members since I do not expect to derive a class from *RNLI* which might need a protected interface. Even if another class were to be derived the need for protected access is questionable.

Mutators are not defined, functions capable of changing the object state. The designed intention is to report state not change it.

Additonal members for *RNLI*. At the moment *RNLI* has achieved a balance of simplicity versus functionality I would describe it as minimal but complete! Well it is a subjective measure:-)

Summary

This mini series of articles started out by noting some of the most common C++ problems; pointer and memory errors such as:

a) Memory leaks (such as a **new** without a corresponding **delete**)

b) Deleting the same pointer again (probably corrupting the heap)

c) Wild pointers (the pointed to object no longer exists)

The debug class *RNLI* was built up from scratch and in the process a number of interesting topics were touched on such as approach to design, static variables, the canonical class, operators **new** and **delete** and virtual destructors.

The end result is that some of the original objectives have been met:

- Provide some macro magic to automatically include/remove code
- Differentiate memory allocated via the heap
- Provide some heap walking capability to "see" what's in memory
- Provide some random check capability within '*main*'
- Output debugging information to a file

However, creating a debugging class was not the overall objective. That was a convenient vehicle to understand some of the processes that are going on in a C++ program. If you find some use(s) for *RNLI* or decide to extend it – that's fine, but remember that there are plenty of full featured tools already which target this area in a great deal more depth.

Roger Lever
rnl16616@ggr.co.uk

In Overload 13, Roger will return to look at the application which motivated the design of RNLI – Ed.

Heapwalking problems
by Sean A. Corfield

In *Overload 11*, I claimed there was a bug in Roger's *RNLI* class and asked if anyone could spot it. No-one wrote in so I shall explain the bug myself.

Spot the bug

Consider the linked list that *RNLI* builds during construction and manipulates during destruction – what exactly happens given the following sequence of **new**s and **delete**s?

```
Base* p1 = new Derived;
Base* p2 = new Derived;
delete p1;
Base* p3 = new Derived;
```

Debugging line by line

Let's trace through each line, watching *rnliHeap* and the *next* pointer in each object:

To start with, *rnliHeap* is zero.

```
Base* p1 = new Derived;
```

Now *rnliHeap* and *p1* both point to the new object and *p1->me* == *p1*, *p1->next* == **0**.

```
Base* p2 = new Derived;
```

We are building the linked list, *rnliHeap* == *p2*, *p2->me* == *p2*, *p2->next* == *p1*.

```
delete p1;
```

In the destructor, we change *rnliHeap* to *p1->next* (zero) and then set *p1->me* to zero. This leaves *p2* pointing at the non-existent object previously pointed at by *p1*! *p2* is orphaned from the heap list at this point.

```
Base* p3 = new Derived;
```

The constructor now chains this new object onto the (incomplete) heap list and... I think you get the idea!

Fixing the problem

The key here is that objects may not be destroyed in the reverse order of creation if they are on the heap. An object being destroyed may be anywhere in the heap list so the choices are:

1. walk down the heap each time to locate and unthread the object being destroyed,
2. use a doubly-linked list to ease removal of objects from the middle of the list.

The former would probably be extremely slow but the code is simpler. I leave it as an exercise to modify *RNLI* to use a doubly-linked list.

Sean A. Corfield
Object Consultancy Services
ocs@corf.demon.co.uk

/tmp/late/* Constraining template parameter types

by Kevlin Henney

All that genuinely constrains what type may be given as a parameter for a template function or class is the way it is used in the program text. There is no assumption on the part of the compiler that there is more about the given type that may be checked in advance of its use in executable code.

This is at once both a strength and a weakness of C++'s template mechanism: a strength in that otherwise unrelated types with a similar set of operations may be used, e.g., **int**, **double** and *complex*<**float**> all support binary **operator+**; a weakness because use of a function name with a type for which that function is not defined is typically not detected until link time, often with an obscure error message.

Sometimes it is obvious what operations on a type are expected. For instance, the *complex*<> template class expects some kind of numeric that supports standard arithmetic operations. We expect *complex*<**long double**> and *complex<rational>* to be legitimate, but not *complex<string>* or *complex<window>*.

I think the following stands a good chance of working – Ed.

```
complex<string> msg1("hi",
"good");
complex<string> msg2("ya",
"bye");
msg1 += msg2;
cout << msg1 << endl;
```

Providing reasonable names for template parameters can go some way to clarifying what is intended. For a numeric type, such as required by *complex*<> and *valarray*<>, *numeric* or *numeric_type* are more helpful names than either *type* or *T*, which incorrectly suggest that a more general type is acceptable.

It is possible to be yet more precise using some form of specification, as the STL has done, outlining minimum requirements for a type, eg., any type used with a container class must have an assignment operator and a copy constructor, amonst other things. Such documentation is external to the code, but is nonetheless useful.

On the whole most type substitution errors, and certainly all those relating to function signatures, will be picked up by link time. The techniques outlined above merely help in prevention. The errors that slip through tend to be semantic constraints that may not manifest themselves until run time. We may have very good reasons for restricting the expected type, e.g., where memory management, persistence, low level mapping, or mixed language programming are issues.

Elaborate

Until recently the **class** keyword has been the only way to introduce a type name in a template argument list. Out of the original C virtue of keyword conservation, **class** was pressed into service to indicate any type in this context and not just a user defined type.

Having a minimal set of keywords is an ideal that ended up on the cutting room floor some time back. There is now a better candidate for the job, **typename**, and it is one that developers should use in preference to **class** when a class type is not necessarily required.

```
template<typename numeric_type> ...
```

clearly reads more accurately than

```
template<class numeric_type> ...
```

Originally introduced for different reasons, in this context the **typename** keyword is somewhat self explanatory – the use of **class** to represent type names other than classes always requires explanation.

It would help matters if the draft standard itself used ***typename*** *in template specifications where appropriate – Ed.*

One hopes that history will consign this use of **class** to the dustbin; eventually when **class** is used it will mean just that. There is, however, a technique that may currently be used to constrain a template type argument in this fashion:

```
template<class value_type>
class container_of_class
{
public:
    typedef class value_type value_type;
    ...
};
```

I have used **class** in the template argument list here because I actually mean it [NB: whenever I use the word **class**, I am also referring to **struct**]. By elaborating it later with the **class** keyword I have constrained it to not be a built-in type, a **union** or an **enum**. The same trick works with elaborating as **union** or **enum**, although in these cases **typename** should be used in the argument list as neither **union** nor **enum** received the same privilege as **class** in this context.

A **typedef** of a user defined type name as itself is harmless and often pointless except for compatibility with C code, i.e.,

```
typedef struct type {...} type;
```

However, in the *container_of_class*<> example above it has the effect of exporting the template parameter as a public member of the class. It would otherwise be visible only within the class scope.

Of course, the key to the technique is the elaboration, wherever it occurs, and the export technique is simply another useful technique which does not in itself require elaboration. You may decide to use different names, to make this **typedef** private, or to just use the elaboration at one or more points where the type is used within the class definition. This is needed particularly where a template function rather than a class is being constrained.

Arithmetic types

We may have something like the opposite requirement: the type parameter must be a built-in arithmetic type (**int**, **char**, **double**, **enum**, **bool**, **unsigned long**, etc.). To enforce this constraint we need some feature of the language that accepts all of these types and no others.

Looking closely we see that all of these can be assigned, using either implicit or explicit casts, the value **0**. We can use a dummy variable that in some way depends on this value:

```
template<typename arithmetic>
class some_numeric
{
  ...
private:
  // template parameter type constraint
  enum { constraint = int(arithmetic(0))
};
};
```

The cast to an **int** is to cater for the floating point types. This class will only compile when the expression initialising the **enum** constant is legal. As the constant is a dummy value, and hence not intended for use, I have made it private with a hopefully meaningful name as documentation – such techniques rely on generating errors at build time, so this is quite important.

Integral types

If we are interested only in integral types (arithmetic types excluding the floating point types), we could simply remove the cast to **int** used in the previous example. Alternatively we can use another context where only integral types are permitted: bit fields. Here it is convenient to encapsulate the constraint in a class:

```
template<typename integral>
class integral_only
{
private:
    integral : 0;
};
```

An anonymous alignment field is used to enforce the constraint in an otherwise functionality-free class. We would use it as follows:

```
template<typename integral>
```

```
class some_class
{
    ...
private:
    // template parameter type
constraint
    static const integral_only<integral>

constraint;
};
```

To be strictly correct we also need a definition for *constraint*, not shown here.

The *constraint* variable is never accessed, so it is possible that on some existing systems it will simply be ignored and the constraint will go unchecked. If this is the case either reference it somewhere else, doing nothing with it, or move the **static** from class to block scope in a function guaranteed to be called, e.g., a constructor or destructor.

Some systems even allow you to get away with a completely stateless constraint. Merely mentioning the constraint type in a **typedef** is enough to cause attempted template instantiation, and thus validation of the constraint:

```
template<typename integral>
class some_class
{
    ...
private:
    // non-portable constraint
    typedef integral_only<integral>

constraint;
};
```

However, it is unwise to rely on this as many other systems do not bother instantiating the template unless it is used to define an object.

Integers only

What if we genuinely only want integer types, i.e., the integrals excluding **enum**s? One solution would be to take the solution just given, give the bit field a size, name it, and initialise it to **0** in a dummy constructor. Because of the stronger typing in C++ this is not a legal assignment for **enum**s and it will only compile for genuine built-in integer types.

Alternatively we can take advantage of the, relatively recent, addition to the language of **static const** initialisers within a class definition. These are only valid for integral types and must be initialised by compile time constants. We also avoid executing any code as a consequence of introducing a constraint – true, that a constructor initialising a bit field is not a great cause for concern, but we would rather not introduce any executable code as a consequence of our type checking efforts.

```
template<typename int_type>
class ints_only
{
    ...
private:
    // template parameter type
constraint
    static const int_type constraint =
0;
};
```

Again the use of a **0** excludes **enum**s. Thus the following types are legal:

```
ints_only<int>
ints_only<unsigned char>
ints_only<bool>
```

And the following are not:

```
ints_only<double>
ints_only<void*>
ints_only<string>
```

Derived classes

A common requirement would be to constrain a class parameter to be derived from a particular class. In other words, constrained genericity similar to that found in languages like Eiffel. Some may be tempted to use *assert* and RTTI, but this is an abuse of both these features. The errors are of static type and, as such, are statically detectable with a little lateral thinking. The technique here is again to use dummy **static**s and create a general purpose constraint class:

```
template<class base, class derived>
class subclass
{
private:
    static const base* const
substitutable;
    static const size_t not_void =

sizeof(base);
};
```

The key to the substitutability constraint is in its definition:

```
template<class base, class derived>
const base* const
  subclass<base, derived>::substitutable
=
    (const derived*) 0;
```

The explicit cast ensures that *derived* is either *base* or a class derived from it, otherwise it will fail to build. The other constant is simply there to ensure that *base* is not **void**, since a pointer to any type is certainly substitutable for a pointer to **void** but it is illegal to take **sizeof(void)**.

Assuming sensible naming, the following are legal:

```
subclass<window, graph_display>
subclass<socket_address,
internet_address>
subclass<ostream, ofstream>
```

And the following are not:

```
subclass<void, any>
subclass<long, int>
subclass<window, text>
```

Summary

The key to all the techniques described here is in forcing a failed build: static error detection is far superior to dynamic error detection. Where this involves operational compatibility it is clearly quite simple, and is the basis of C++'s template system. For our other constraints we can use, quite literally, a more declarative approach and in such a way that also acts as a form of documentation.

Enforcing compile time constraints using templates is a theme I will be returning to in future articles.

Kevlin Henney
kevlin@two-sdg.demon.co.uk

editor << letters;

It seems you took my admonitions in *Overload 11* to heart! It's interesting to note that many of the letters I have received are concerned with Microsoft's latest release.

Sean,

Can you or any of your readers recommend a book which properly covers the creation of Doc/View applications in Borland C++, preferably utilising the AppExpert and ClassExpert. (I use 4.0, but will probably move to >=4.5 fairly soon). None of the books I have found (including Borland's own) seem to recognise the existance of AppExpert and ClassExpert, and none of them give more than a paragraph on the Doc/View system, along the lines of "derive a class from *TView*, one from *TDocument* and put them together with *TDocManager* and then everything is easy". In fact ClassExpert gives you the option of deriving from various other classes such as *TListView* and *TFileDocument*, but then leaves you with no idea of what functions these classes supply and what members should be overridden to provide your specific functionality. I generally find ClassExpert a very useful tool, with its list of the virtual functions for the class under development, but it seems to fall down with mutliply-derived classes such as *TListView*. It is very frustrating to have tools to make things easy, but the books only tell you how to do it the hard way.

Any suggestions would be appreciated.

Dave Midgley
100117.2522@compuserve.com

I can't help so over to the readers – any takers?

Hello Sean

On Java:

I for one would be interested in seeing *Overload* contain a 'Java Corner'. A rationale follows: I would agree with The Harpist's comment that Java (based on C/C++) can co-exist quite happily with other languages like C or C++ without replacing them. Since it is based on C++ I would view it as appropriate for *Overload* where something like Delphi (based on Pascal) is not.

On languages:

It seems to me that developers generally would like that one tool (ABC or XYZ programming language), that can be used for everything – hence the talk of using ABC or XYZ. However, it appears more typical, certainly in a corporate environment, that the developer will use ABC and (maybe) XYZ dependent on requirement. The bottom line must surely be a choice based on fitness for purpose of that tool, in combination with the strategic and tactical issues that that choice involves.

On developer expertise:

When I first started to look at programming languages I was convinced that it was essential to be an 'expert' in one of them and largely to the exclusion of all others. This was surely better than being a jack-of-all and master-of-none? However, that view has changed to: have a good understanding of the key languages (2 or 3), their strengths and weaknesses and match this, where possible, against requirements. Choosing the right tool for the job.

On programming situations:

Programming is a complex business and contrary to some popular opinion it is not getting easier. One only has to look at the connectivity, configuration, integration and usability issues of modern applications to recognise that there are vast areas of uncharted territory. As programming technology advances so does requirement, in a never ending spiral!

As technology and ideas advance, programming languages evolve to support the new paradigm – C++ is an obvious recent example. As a developer it is important to leverage that – where appropriate. Conversely it is important not to be dragged off course, tangentially chasing every new advance. Extremes, as always, are rarely the best course.

Client/Server as a paradigm is here and has been adopted by many businesses, Client/Network heralded by the Internet and languages such as Java are visible on the horizon. It is important that developers have a good understanding of its merit and based on that knowledge decide on extending their understanding to an in-depth knowledge.

There is an intentional bias towards the developer here, or the person(s) involved in writing applications since it is important that

a) Project decisions are influenced by both the business and developers

b) Decisions are timely and based on a sound understanding of requirement, technical capability, business impact

c) Technical skills in the current market are maintained

Change is the only constant – it needs to be managed, by everyone.

Roger Lever
rnl16616@ggr.co.uk

As editor I would welcome contributions on Java – I have started looking at the language but, sadly, HotJava is not yet available for my favoured platform (although it is apparently in alpha testing internally).

As an adjunct to Roger's, very reasonable, comment that we as developers should be aware of more than one language/tool, it should probably be noted that the development environment is often chosen by corporate policy rather than by the more sensible "fitness for purpose" line of thinking. One company at which I worked used a mixture of C, Prolog and 4GLs to achieve their requirements, carefully picking the best tool for each part of the project. Corporate policy dictating, say, Microsoft Visual C++ 2.0 for all tasks does no-one any favours (and I'm not, for once, picking on Microsoft – it's just the first example to come to mind).

The following exchange of letters was between Chris Simons and Andrew King (of Microsoft) and copied to Overload – it follows on from correspondence in Overload 10 and Overload 11.

Hi Sean and Andrew,

I read and enjoyed *Overload 11* – lots of goodies as ever :-)

I note with interest Andrew's letter to the editor saying that upgrading to VC++ 4 will 'be a winner' with STL. It's uncanny then that this week my compiler was upgraded to VC++ 4 and so I dived straight into STL.

Except that it's not part of the install procedure...

One has to:

1) manually copy the files

2) #define *NOMINMAX* to prevent clash with windows macros,

3) create a namespace to wrap STL headers,

4) perform 23 edits in four files.

Good job the **readme** was clear!

Frankly, I'd hardly call this install 'on to a winner' :-/

I've still got problems intergrating with pre-existing project code (which, of course, cannot be altered. *sigh*). Something to do with **#include**ing header files within a namespace definition and elsewhere – perhaps preprocessor multiple include guards? More research required there I guess.

Chris Simons
cl-simon@csm.uwe.ac.uk

Hi,

The version of STL Microsoft shipped was a public domain version. We were, therefore, unable to alter the sources. Yes, sadly there is some setup work to do in integrating STL, but it's 1) a known quantity; and 2) a lot more benefit than if we had said to ourselves, "this might be time consuming to do, we'll just not include STL." Remember, the mods to STL happen only once. You can reuse the code over and over again.

On the subject of SETUP, if you look at the CD, there is a file called **autorun.exe**. This should automatically run under Win95 when you put the disc in the drive. Under Windows NT you can just run it (Windows NT doesn't have "spin and grin" yet for CDs). It's a master setup and it offers to install (i.e., copy) STL to your hard disk.

Hope you like VisualC++ 4.0.

Andrew King
andrewki@microsoft.com

Well, STL (and other components of the standard library) should all be in the std namespace but I agree that proprietary third-party libraries such as MFC should be in a vendor-specific namespace – that is, after all, what ***namespace*** *was designed for.*

I will resist commenting on Andrew's claim that Microsoft could not alter the public domain STL source code, except to note that other compiler vendors seem able to ship a version that works out-of-the-box with their compilers. I would be interested to hear the real reason that Microsoft felt it acceptable to ship such a hack without going through proper QA and integration procedures – they had clearly spent some time ensuring that a modified version of STL would work with STL (up to a point, as Chris's second letter shows).

Hi Sean,

Might further feedback on VC++4.0 and STL be of interest?

Installing VC++4 and STL for use with existing project code has proved a challenge. The order of **#include** directives across multiple source files has proved critical if STL is put in a namespace as Microsoft recommend, thus

```
namespace stl
{
  #include <vector.h>
  // etc. etc.
}
```

When any STL file includes, for example, **<iostream>**, streams then become part of the STL namespace, but also cannot then be included once more in the global namespace due to preprocessor multiple inclusion guards. Careful analysis of the order of header file inclusion has been required to overcome this.

The issue is much less problematic when classes declared in namespaces reside within preprocessor multiple redefinition guards.

Come to think of it, shouldn't MFC be in its own namespace and STL at global scope?

Pip-pip,

Chris Simons
cl-simon@csm.uwe.ac.uk

Sean,

In *Overload 11* you asked about whinges re MS VC4.

I didn't really, but I'm always glad to hear of people's experiences with commercial products!

The following is the help given on one of the warnings from this compiler:

```
C++ Exception Specification ignored

A function was declared using exception
specification. At this time the
implementation details of exception
specification have not been
standardized, and are accepted but not
implemented in Microsoft Visual C++.
Code compiled with ignored exception
specifications may need to be recompiled
and linked to be reused in future
versions supporting exception
specifications. You can avoid this
warning by using the warning pragma:

#pragma warning( disable : 4290 )
```

Of course, MSVC4 does support the draft standard exceptions – except where it has not been "standardized". ;-)

I tried the above advice which does remove the error message, but the following fragment still doesn't compile:

```
class string
{
public:
  // Exceptions
  class outofrange
  : public exception
    { public: outofrange(); };
  class lengtherror
  : public exception
    { public: lengtherror(); };
  // Constructors
  string() throw(bad_alloc);
  string(const string& s)
                throw(bad_alloc);
error C2146: syntax error : missing ';'
before identifier 'string'
```

Commenting out the **throw** declarations does eliminate the problems. :-(

I'll try to get you a proper report early Jan (unless you have another, faster, source).

FYI, it looks from the header files and help as though the May '94 draft is the last one MS looked at. (e.g., *xalloc*, not *bad_alloc*). OTOH some of the library source mentions *bad_alloc* in comments. All very confusing.

Alan Griffiths
Senior Systems Consultant
CCN Group Limited
agriffiths@ma.ccngroup.com

Alan's report on using Microsoft Visual C++ v4.0 appears elsewhere in this issue.

The missing ';' bug can be fixed by putting ***two*** *semicolons after the throw spec of a constructor:*

```
string() throw(bad_alloc);;
```

I am indebted to Andy Sawyer <andys@thone.demon.co.uk> for providing this insight!

Hi,

What I have found is that the compiler that comes with Microsoft Visual C++ 4.0 produces an output value of 12. *[for the code below]* This is the value associated with the variable in the namespace. From your article, and the draft standard, I expected the value to be 67. Have I interpreted something wrong or is the Microsoft compiler failing to implement this feature correctly. Any and all information would be greatly appreciated!

```
#include <iostream.h>
namespace A
{
  int j = 12;
};

double j = 90.90;
void main()
{
  int j = 67;
  if (j)
  {
    using namespace A;
    cout << j << endl;
  }
}
```

Jay
jayc@smtpgate.tais.com

Microsoft have implemented the using-directive incorrectly. They are in good company as Metaware and Programming Research both implement the above example in the same way. However, their excuse is that they implemented namespaces nearly three years ago when the draft wasn't clear on this.

++puzzle;

In *Overload 11* Francis set a puzzle for everyone to have a go at. The winning entry appears below followed by a discussion of design issues by Francis.

Handling dates with locale based day and month information

by John Smart

As the only entry, though not what I was really after, John wins the copy of 'The Mythical Man Month' – Francis

This is an implementation of dates that allows them to be externally represented in a variety of locale based textual formats. The design could, I believe, be used as a model for a multi-calendar date system as noted below.

The classes used in this implementation are:

Dates	A concrete class holding a date as a serial day value. Provides compact internal storage for dates with fast arithmetic operations, e.g., comparisons, subtraction and adjustment by number of days.
DateText	A class holding a locale based set of constant Calendar texts.

DateFormat
: A class that associates a *DateText* instance with a date formatting string and the size of buffer required to hold an instance of a date in the supplied format.

CalendarTime
: A concrete class holding calendar time (a value of *time_t* from <*time.h*>). This represents time in seconds since 1/1/1904 (on a Mac)

DateIn
: A class supporting the input of *Dates* in a format specified by a *DateFormat* object.

Formatter
: A general purpose class that supports the interpretation of formatting strings. It generalises the implementation of *DateFormat*. This class is described in an issue of *Overload 6*.

The five date related classes do not use inheritance. The design of these classes is described below.

class Dates

```
struct tm;
class  DateFormat;

class  Dates {
       unsigned long serial_days;
public:
       Dates(int day, int month, int
year);
       Dates(int day_in_year, int year);
       Dates(void);
       const char * operator
                   ()(DateFormat&);
       Dates operator ++(void);
       Dates operator --(void);
       int  invalid(void);
       int  operator < (Dates rhs) {
            return (serial_days <
                   rhs.serial_days);};
       int  operator > (Dates rhs) {
            return (serial_days >
                   rhs.serial_days);};
       int  operator <= (Dates rhs) {
            return (serial_days <=
                   rhs.serial_days);};
       int  operator >= (Dates rhs) {
            return (serial_days >=
                   rhs.serial_days);};
       int  operator == (Dates rhs) {
            return (serial_days ==
                   rhs.serial_days);};
       int  operator != (Dates rhs) {
            return (serial_days !=
                   rhs.serial_days);};
       int  operator - (Dates rhs) {
            return (serial_days -
                   rhs.serial_days);};
       Dates operator += (int);
              //add days to a date
       Dates operator -= (int);
              //subtract days from a
date
       tm decompose(void);
};
```

It holds a date in its Serial Day representation (**unsigned long**). This representation has been described in a article in *.EXE* a couple of years ago. I have used an algorithm published in that article to convert between Serial Days and the Gregorian Calendar that is valid for dates between 1st March 1900 and 28th February 2100 (Year 2100 is not a leap year). The article also published algorithms that coped with a much larger range of dates.

Its constructors accept day of month, month number and year as integer values or number of days since 1st January and year as integers. Any integer values will be accepted and converted into a serial days value. However, if the resulting value is outside of the range of supported dates (1/3/1900 to 28/2/2100) the value held will represent the date of 29/2/1900 (an invalid date!) and this value, once established, will not change. The *invalid***()** member function can be used to identify an invalid date.

The default constructor creates the current date through the use of the ANSI C Library functions *localtime***(**&*time***(***NULL***))**

The arithmetic and relational operators are provided to support the use of *Dates* class objects. The **operator +(int)** and **operator -(int)** are omitted to avoid the use of friends to provide commutative addition and subtraction.

The *Dates***::***decompose***()** function converts a serial day value into the standard ANSI C **struct** *tm* (see <*time.h*>) so that the existing facilities of <*time.h*> may be applied to a date. There is no date constructor accepting a *tm* value since it is not necessary and avoids having to deal with possibly inconsistent data; the constructor *Dates***(***x***.***tm_yday***,** *x***.***tm_year* + *1900***)** can be used instead.

Note that all the private functions used to implement the public interface are simple non-class static functions since they only operate on built in types; they do not need pollute the class definition.

The member function, *Dates***::operator ()(***DateFormat***&)**, converts a date into a constant **char*** according to the data supplied by the *DateFormat* parameter. It allows any *Dates* object to be converted into any desired textual representation.

class CalendarTime

```
#include     <time.h>

class  DateFormat;

struct CalendarTime {
       time_t calendartime;
       CalendarTime()
       : calendartime(time(NULL)) {};
       CalendarTime(time_t t)
       : calendartime(t) {};
       const char * operator
                        ()(DateFormat&);
};
```

This class encapsulates a *time_t* value so that it may be output using a *DateFormat* object and read in with a *DateIn* object. A full implementation would provide arithmetic and comparison operations.

class DateFormat

```
class  DateText;

struct DateFormat {
       char const *const a_format;
       DateText &    the_text;
       char *const    buffer;
       const int     size;
public:
       DateFormat(char const *const,
               DateText&, int
buffer_size);
       ~DateFormat();
};
```

This has a single constructor that requires:

1 A date formatting string,

2 A *DateText* object

3 The size of the buffer that will accommodate a date formatted according to the formatting string and the text supplied by the *DateText* object.

All the members of this class are constants.

This constructor allocates a buffer of the specified size as a member of the constructed object; it is deleted by the destructor. This allocation means that the use of *DateFormat* objects is not thread-safe; see the description of the class *LongDate* for a solution to this problem.

The formatting string uses the format specifiers defined for the ANSI C function *strftime***()**, including H, M & S, plus the format specifier '%D' which appends an ordinal suffix to the day of the month e.g., 1st, 2nd, 3rd, 4th, 21st etc.

The formatting string also supports the width, justification, fill and repetition facilities implemented by the *Formatter* class.

Thus the formatting string: "%9A, %2D %9B %Y @ %2_0H:%2_0M:%2_0S" generates the text: "Thursday, 4th January 1996 @ 00:00:00".

The text to be used for the generation is specified by the *DateText* object.

struct DateText

```
#include     <time.h>

class  Dates;
class  CalendarTime;
class  istream;

struct DateText {
       const char *const
       full_month_names[12];
       const char *const
       short_month_names[12];
       const char *const
       full_day_names[7];
       const char *const
       short_day_names[7];
       const char *const
       ordinal_suffix[31];
       const char *const
       am_pm_text[2];
       const char *
              outForm(Dates
                 ,const char *const
format
                 ,char *const
buffer
                 ,int         size)
const;
       const char *
              outForm(CalendarTime,
                  const char *const
format
                 ,char *const
buffer
                 ,int         size)
const;
       int inForm(tm&
             ,const char *const format
             ,char *const       buffer
             ,int               size
             ,istream&     source)
const;
private:
       const char *
              outForm(tm&
                  ,const char *const
format
                  ,char *const buffer
                  ,int size) const;
};
```

By making the class members arrays I can ensure that the right number of values are supplied. The members are all public so that a *DateText* declaration may be simply initialised (see *CalendarDateNames.cp*). In case the user does not supply enough elements for any of these arrays the implementation that reads these values should recognise *NULL* elements and use a fixed default string.

The *DateText* class consists of constant arrays of constant strings which provide:

full_month_names[12]	The twelve month names
short_month_names[12]	The twelve abbreviated month names
full_day_names[7]	The seven names for the days of the week
short_month_names[7]	The seven abbreviated day names
ordinal_suffix[31]	The 31 possible ordinal suffices
am_pm_text[2]	The am/pm text for hours in the day

I have not found a way of inhibiting the user from declaring an uninitialised object of the class *DateText*. Is it possible in C++ to ensure that these constant objects are always initialised without the overhead of providing constructors (which may not be able to tell whether the arrays have enough members)?

The only solution I can think of is to provide a constructor that requires 71 **const char *const** parameters (one for each element of each array!). I don't like the runtime overhead this incurs compared with ordinary initialisation being done before execution.

Objects of class *DateText* can be declared for any locale that uses the Gregorian Calendar; the file *CalendarDateNames.cp* contains declarations for English, French and German names

For Calendars that are not Gregorian an equivalent class of constant arrays of constant strings could be defined. A class could then be derived from *Dates* that knows how to convert serial days into the numerical values used by another Calendar and these values can then be used to look up its external representation. One could even devise another form of date formatting string to parameterise the external representation. The same design model can still be used. All it relies upon is the fact that dates are always calculated in days.

The member functions *DateText***::***outForm***()** accept a *Dates* or *CalendarTime* object together with a formatting string and the address and size of a buffer into which the textual representation will be written and returned. The user is responsible for the management of the supplied buffer.

The member functions *DateText***::***inForm***()** translates an input stream into a *tm* structure according to the supplied formatting string and its member's text strings.

class DateIn

```
#include      <time.h>
#include      "Dates.h"
#include      "CalendarTime.h"

class istream;
class DateFormat;

class DateIn {
      Dates date;
      time_t time;
      int  input_error;
public:
      tm   details;
      DateIn(istream&, DateFormat&);
      operator Dates(void) {return
date;};
      operator CalendarTime(void)
                          {return
time;};
      int invalid(void)
                   {return
input_error;};
};
```

This class provides the interface whereby dates, textually represented in the format described by a *DateForm* object, can be converted into their internal representation.

The constructor reads the text from the *istream* using the formatting string and textual names supplied by the *DateForm*. The values read are placed in the public member 'details'. The conversion operators *Dates* and *CalendarTime* can then be used to convert the value into internal form.

The *DateIn***::***invalid***()** operation can be used to discover whether there were any errors during the input conversion. If there were then the corresponding members of details will be negative. The user of this class is thereby able to build appropriate diagnostics into the application. Converting an erroneous *DateIn* into a *Dates* values will always succeed but may generate the invalid *Dates* object; this is also generated when the input date is out of range.

The use of the above classes

The user of *Dates* may create initialised *Dates* and manipulate and store them very efficiently.

To handle the external representation for these *Dates* the user can declare a variety of date formatting strings and any number of locale based *DateText* objects. These may be combined into any set of *DateForm* objects of the user's choice. These objects and their members are prime candidates for inclusion in a namespace rather than being global objects.

The member function *Dates*::**operator ()(***DateForm***&)** can then be invoked to translate any *Dates* object according to any of *DateForm* objects. Similarly the constructor *DateIn*::*DateIn*(*istream***&,** *DateForm***&)** can be used to translate text into a *Dates* and/or *CalendarTime* object.

If, for a particular context, it is required that all *Dates* are externally represented in a single format then a class may be derived from *Dates* which encapsulates the required formatting interface. An implementation of this approach is shown by the class *LongDate* which provides the friend operations: *ostream***& operator <<(***ostream***&,** *LongDate***);** and *istream***& operator >>(***istream***&,** *LongDate***);** so that a *LongDate* can be used with the standard streams interface. The required *DateForm* is held as a static data member so that a *LongDate* object is just as efficient as a *Dates* object.

```
#include      "Dates.h"

class DateFormat;
class ostream;
class istream;

class LongDate : Dates {
       static DateFormat   longDateText;
public:
       LongDate(int day, int month,
               int year)
       : Dates(day, month, year) {};
       LongDate(int day_in_year, int
year)
       : Dates(day_in_year, year) {};
       LongDate(void) {};
       LongDate(const Dates& d)
       : Dates(d) {};

friend ostream& operator <<(ostream& os,
                               LongDate
ld)
          {return (os <<

ld(LongDate::longDateText));};
friend istream& operator >>(istream& in,
                              LongDate&
ld);
};
```

The **operator <<()** implemented by *LongDate* simply encapsulates the call of the *Dates*::**operator()(***DateForm***&)**. This is not thread-safe but the following would be:

```
ostream&
operator <<(ostream& os, LongDate ld) {
  char * buffer = new

char[LongDate::longDateText.size];
  os << LongDate::longDateText.
     the_text.outForm
           (ld

,LongDate::longDateText.a_format
           ,buffer
           ,LongDate::longDateText.size
           );
  delete[] buffer;
  return os;
}
```

This dynamically allocates the buffer before the call of *DateText*::*outForm***()** as an argument to *ostream*::**operator <<(***ostream***&, char*)**. After the text has been copied into the *ostream* object the buffer is released. For non-threading applications I don't think the overhead of dynamically allocating/deallocating a buffer for each date translation is necessary; that's why the *DateFormat* constructor allocates the buffer. If the users cares to use *DateText*::*outForm***()** directly they can even provide a statically allocated buffer explicitly to achieve even more predictable run time performance.

An implementation of these classes supported the following source code which generated this output text:

```
  Monday, 25th December 1995 ==
                      Mon, 25/Dec/95
  Lundi, 25me Décembre 1995 ==
                      Lun, 25/Déc/95
  Montag, 25te Dezember 1995 ==
                      Mont, 25/Dez/95
```

The code follows:

```
char const *const fulldate_format =
                        "%9A, %2D %9B
%Y";
char const *const shortdate_format =
                        "%a,
%2d/%b/%y";

extern DateText     EnglishDateNames;
extern DateText     FrenchDateNames;
extern DateText     GermanDateNames;

DateFormat Full_English
(fulldate_format,

EnglishDateNames,40);
DateFormat
Short_English(shortdate_format,

EnglishDateNames,20);
DateFormat Full_French (fulldate_format,
                     FrenchDateNames,
40);
DateFormat Short_French
(shortdate_format,
                     FrenchDateNames,
20);
DateFormat Full_German (fulldate_format,
                     GermanDateNames,
40);
```

```
DateFormat Short_German
(shortdate_format,
                          GermanDateNames,
20);

void xmas95(void) {
      static char is[] = " == ";
      Dates xmas(25,12,1995);
      cout << xmas(Full_English) << is
         << xmas(Short_English) <<
'\n';
      cout << xmas(Full_French)  << is
         << xmas(Short_French) << '\n';
      cout << xmas(Full_German)  << is
         << xmas(Short_German) << '\n';
};
```

These classes have been implemented using Symantec for Macintosh version 7. The following two problems (bugs?) were found during the implementation:

1. An *ostrstream* buffer initialised with the constructor *ostrstream*::*ostrstream*(**char*** *buffer*, **int** *size*); never has a '\0' character terminating the text inserted. One has to fill the buffer with zeroes before writing to it!
2. An **extern const char* const** *xx*; in one file does not achieve external linkage to the definition: **const char*const** *xx* = **"***text***";** in another file.

John Smart
smart@baesema.demon.co.uk

On John's second problem, the definition needs ***extern*** *in order to get external linkage (otherwise it has internal linkage because xx is* ***const****). On the first, although I'm not certain, I believe there is a call that terminates the ostrstream prior to extracting the* ***char**** *string.*

The code accompanying John's article will appear on a future CVu disk – Ed.

Making a date

by Francis Glassborow

Before we start

The following is not intended as a tutorial nor as a definitive way of tackling the task of developing a date class. It is (despite my writing style) a collection of partially organised thoughts about the problem. I do not care how many formal methods you have available to tackle design issues, you will still have to go through this kind of thinking before you start laying down an actual design. Unless you do so your work will be plagued with constant visits to early design decisions as a result of later ones (or even implementation problems).

Getting started

- Get little black book ... sorry, wrong kind of date.
- Grow palm tree ... closer.
- Create a kernel ... sounds better.

Deep in the centre of any date object is the concept of locating days in a time stream. The process of locating requires a reference point. This is rarely an easy point to tackle because we have to decide what this starting point will be. In everyday usage we often use context. When talking to you on the phone I might say 'I'll see you the day after tomorrow.' But if I wrote something like that in a letter, I better make sure that the letter is dated (similar problems arise when people leave messages on answer-phones, they assume that the message will be heard on the day that it was sent). Even if I provide a date this assumes that you will be using the same calendar that you are. There was a period in European history when it was vital to say whether you were using a Julian or a Gregorian calendar.

We will need some object that will store information about where a day is located in the time stream. This object should be conceptually independent of any calendar system (in so far as that can be achieved). This is the kernel of a date system. The following are some thoughts on designing such a date kernel.

The date kernel

Absolute dates

What we need is some absolute, universal reference point. The beginning of the Universe looks like a good candidate. It is, of course, completely impractical for our purposes. In fact all universal reference points that I can think of are impractical. What this means is that all practical dating mechanisms are relative to some arbitrary starting point. So choose one. Today's date (13th January 1996) is as good as any other. Reference all days as +/- from now. Some time we will need to tie down what a day is and when it starts. In other words we will need a time 0 point. Note that not all human calendars define the same point as the start of a day but we can leave that for later. For the time being let us use the Gregorian convention of starting a day at midnight.

Now we can identify any moment as being in day +/- x from now. So our date kernel object will store an offset from time 0 by any mechanism we finally choose.

Relative and partial dates

How should I handle such things as ‘three days from now’ if I do not know when now is? Or ‘the third of next month’? Or ‘June 22nd’ and so on? We often fix things on some local relative basis or provide only partial information relying on context to provide the rest. Sometimes only partial information makes sense. For example: ‘How many days are there between June 3rd and November 4th?’ You would be a little surprised if a year was included (unless the time interval spanned years). Note that this type of question is vulnerable to periods crossing leap days.

Let’s consider including a second data item to handle an offset and then we could do something like this:

Reference	*Offset*	*Description*
n	0	n days from base day (13th January 1996 – Gregorian Calendar)
0	n	n days from an unspecified local reference
1	n	n days from a context based reference

We are beginning to go from design issues to design decisions so I’ll break off here with the comment that I’m aware that there is an ambiguity for date (0,0) that will have to be resolved by a convention. And give you a few examples to help clarify the above.

- January 14th 1996 becomes (1,0),
- January 14th becomes (1, 13) – 13 days from the contextual reference of New Year’s Day.
- The 14th becomes (0, 14) – 14 day offset from an unspecified reference point.

And so on.

Note that this mechanism allows us to progressively refine a date by using the offset until we have a complete date when we can transfer the offset to the reference data.

Invalid dates

These are quite distinct from partial or incomplete dates. For example, 13th June is incomplete but 31st June is invalid. It is not part of the *Date* kernel to determine that a date is invalid but it must be able to store that state. We will need some Boolean value to hold this state information which can be set/reset by the concrete date object.

How should this information be made available to the outside world? The first reaction may be to provide an enquiry function. Not bad, I could certainly live with that. The intermediate class designer will want to provide conversion function to convert date kernel objects to **bool** so that they can write lines like:

```
  if (today) dosomething()
```

That has not been adequately thought through. Most unfortunately **bool** has an automatic conversion to arithmetic types so if you did that you would find all sorts of statements misbehaving (creating surprises). If you want to provide a mechanism for date kernel objects to return their status when used in a context where a Boolean value is required you have to provide a conversion to a **void***. Return (convert to) a zero (null pointer) for an invalid date and a non-zero (conventionally **this**) value for a valid date. This minimises unexpected behaviour because there are no valid implicit conversions from **void***.

*This is because the **bool** type in C++ is broken – Ed.*

Functionality issues

What should this kernel do? What access should there be to the constructors/destructor?

Such questions as ‘How many days between...?’ and ‘What is the date 27 days before...’ are not properties of a calendar system but of the time-stream. So that kind of functionality should be provided within the date kernel. (I’m fed up with this level of precision, let’s call it ‘the kernel’). But we should try to design it so that questions such as ‘Does this Jewish date come before this Islamic one?’ can be asked.

The constructor/destructor issue is a matter of deciding just how accessible we want our kernel to be. To answer this we need to move on to consider the fundamental requirements of a (calendar) date (let’s call it ‘a date’).

Putting meat round the kernel

Fundamentally we have two basic options. We can build a date from a kernel by inheritance or we can do so by ‘layering’. The question is, ‘Does a date contain a kernel, or is it a kernel?’ One issue here is automatic conversions – not forgetting the convenience this can provide for parameters. Many functions you write might just want some sort of date, any sort will do. This favours the inheritance route.

However we could possibly want to have the same date (kernel) held by several calendar objects. This mechanism favours using pointers or references to kernel so that several objects can all share the same kernel. There are a few advantages to this if we want to handle multiple representations of the same day.

Having thought about it for some time I think that inheritance is the way to go. I am open to persuasion and would be delighted to see other ways of tackling this problem because I feel that there must be alternatives that I have missed.

I have some reservations about using inheritance, not least because I think that the kernel design is too raw to be let loose on application programmers. This prods me towards having private constructors/destructors. However that mitigates against being able to have pure dates (kernel objects) passed to functions. It is pointless having an private ordinary constructor while having a public copy constructor if you are trying to prevent application programmers from creating raw kernels because that allows construction via self-copying. Don’t tell me this is stupid – it is, but it is guaranteed to work. For example for any type *T* where the copy constructor has not been explicitly declared as private/protected the following is syntactically valid though semantic rubbish:

```
T t = t;
```

Once we have determined that we are going to base a hierarchy on the kernel we then have to decide whether it should be polymorphic. It certainly cannot be a pure polymorph because such concepts as weeks and months are not universally applicable. On the other hand, such things as ‘get the date’ and ‘display the date’ are. By the way, it is because different calendar systems require different functionality that we cannot solve our problem by simply having the kernel contain a pointer to a date (many of you won’t even have realised that was a possible solution) which is a sort of inheritance in reverse and can be useful when you want to change the outward behaviour of an object – sort of polymorphic behaviour without using the type system, in essence you manage a virtual function table. (Aside: perhaps I can get my polymorphic object that way.)

Problems

Designing and implementing date classes for specific calendars can be very difficult. About the easiest is the Islamic calendar (I think, but I am not sure that it doesn’t have some nasty fiddles in some years). The Julian calendar is pretty straight-forward, with our current Gregorian one a little more awkward with its slightly more complicated leap year rule. By the time you reach the Jewish calendar you are beginning to enter tougher territory with inter-calendar months thrown in (sort of like leap days but whole extra months instead). At least all these have the concept of 7-day weeks, with months that have their days numbered consecutively. There are a couple of historical Indian calendars that have missing days (sort of like leap days in reverse) and the Aztecs used quite different religious and secular calendars, neither of which had any concept of either a week or a year.

Something to do

I would much like to see someone implement (having fleshed out a design) some kind of date kernel class which could be used to develop calendar specific classes. If done properly, inter-calendar conversions become easy.

Francis Glassborow
francis@robinton.demon.co.uk

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.

This information was taken from comp.std.c++

Working STL for VC++ 4.0 available

I followed Microsoft's instructions in the STL **readme** file for Visual C++ 4.0. I also added helper code and documented solutions to common problems.

- This code is unique in that it allows STL to work with *CString*.
- The code has been tested extensively with MFC applications.

The files are available at:
ftp.rahul.net/pub/terris/stl.zip

Terris Linenbach
terris@rahul.net

Credits

Founding Editor

Mike Toms
miketoms@calladin.demon.co.uk

Managing Editor

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

Production Editor

Alan Lenton
alenton@aol.com

Advertising

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

Subscriptions

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

Distribution

Mark Radford
mark@twonine.demon.co.uk

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 13* (April/May) should be submitted to the editor by March 24th. Note that this is an extended deadline due to the Editor attending the joint ISO/ANSI C++ meeting.

FULL PAGE ADVERT GOES HERE!