ISSN 1354-3172

# Overload

*Journal of the ACCU C++ Special Interest Group*

*Issue 19*

*April 1997*

Editorial:
Alan Griffiths
CCN Group Limited
Nottingham
Notts
NG1 5HF
overload@octopull.demon.co.uk

Subscriptions:
David Hodge
2 Clevedon Road
Bexhill-on-Sea
East Sussex
TN39 4EL
101633.1100@compuserve.com

£3.50

Contents

# Editorial

**Hello…**

Its been a while since I submitted an article for Overload or C Vu so you may not remember me. I claim the distinction (in Overload 7) of being the only person to provoke Sean into writing a commentary on an Overload article that was as long as the article itself. Surprisingly, in view of the provocative nature of the article Sean's commentary reached much the same conclusions by a different route. (That C++ placed heavy demands on the developer skills and that this was not widely recognised. We differed in that I considered this a problem with C++ and Sean with the expectations of management[1]) Little has changed and Sean and I have been debating again - but more on that later.

When I offered to edit this issue of Overload it was on the strict understanding that it would be for one issue only. This is not because I wouldn't like the job, its just that there are too many demands upon my time at present. Perhaps in a few years when the children are older. Circumstances conspired to make this issue possible for me, meanwhile more long term plans have been put into place.

**…and goodbye…**

I'm sure that I speak for the whole C++ SIG when I say "thank you" to Sean, he has transformed Overload during his time as editor and made it into an important resource for C++ programmers. In the nature of things there will be changes. We cannot expect to find another editor with Sean's grasp of the C++ language and ability to communicate it.

**…and hello…**

There is now a new editor waiting to take over now - John Merrells who will be editing Overloads 20, 21 and 22. (Unlike me he hasn't set a limit here, but this is all he's committed to at present.)

I'll let John introduce himself next issue, for now I'll explain the debate that has arisen between editors old, current and new. (Oh, yes! And Francis).

[1] Sean seems to have shifted ground in the last two years. "I'm increasingly disillusioned with C++ as a reasonable tool to implement OO designs. That's why I've expanded my horizons. I think it would be professionally irresponsible to discourage others from doing the same."

**…and what do you want?**

The highest priority for the future is that Overload keeps coming on a regular basis. We know from experience that if it is uncertain when (or if) the next issue will appear then the supply of articles disappears.

What is in currently in question is the nature of the material that should be incorporated, Overload's relationship with C Vu and what constitutes an acceptable standard of material.

- It will be very difficult to maintain the current standard throughout future issues. It could possibly be done by having articles reviewed before publication, but this would create delays, additional work, and would be hard to manage in a voluntary organisation.

- Overload is the journal of the C++ SIG. I feel that this limits the amount of non-C++ material it is appropriate to publish. In particular any C material goes to C Vu and by analogy this should apply to (for instance) Java.

  Sean feels that Overload should not restrict itself to C++, and has expressed a desire to submit articles on both Java and Smalltalk. (I for one would like to see them, but as editor for Overload would hesitate before accepting them for publication without a mandate from the membership.)

- At present the C++ SIG does nothing except produce and, presumably, read Overload. Should it do anything else? If so what? (The idea of a code library appears to be outdated, but the standard library has some notable holes that it would be nice to plug with *de facto* standard components.)

In any event I'm certain that John will be more confident of the way to take Overload forwards if a you write in and let him know what you want from Overload.

Similarly (since I was elected C++ SIG organiser at the AGM) I'd love to hear your ideas for what the C++ SIG can do for its members.

*Alan Griffiths*
*overload@octopull.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++.

## Observations on the Design of an Address Class - By Mark Radford

### Introduction

In his article in Overload 17/18 [1], Francis posed questions about the quality of the design of his `Address` class. He stated that one member of the `Address` and `Address_Data` classes is poorly designed and implemented, challenging readers to put the fault right. He also asked if readers agreed that the functions in the (surrogate) `Address` class were simple enough to be inline. This article presents my answers to these questions (which I have verified with Francis).

#### Question 1

The function which is badly designed and implemented is

```
make_address_data(
    const String& country)
```

The design problem is that the country is passed as a `String`, which is error prone because of the possibility of miss-spellings or even a country which is not catered for being passed (returning a null is not good enough given that such mistakes can, and should, be trapped at compile time!). (It should also be mentioned here that the `String` class is not standard. The forthcoming standard library provides a `typedef string`. See [2(i)])

The implementation problem is in expressions of the form

```
if (!strcmp(country, "UK")) {
    // ....
}
```

This highlights one of the many bad design features of the standard C library (by the way, I assume here that String supports a conversion to `const char*`, which is also bad; for an explanation see [2(ii)]). This expression (although it will work) is very misleading, as it looks like we're checking that they're not the same! Better would be

```
if ("UK" == country) {
    // ...
}
```

if `String` supports a suitable identity operator, *(I'd expect a conversion constructor for `String` to take "`const char*`" - Alan)* but if it doesn't, then

```
if (0 == strcmp(country, "UK")) {
    // ....
}
```

At least this doesn't encourage assumptions. This however, is speculation, as it doesn't help put the functions design right.

#### Question 2

On the question of whether or not the `Address` class functions should be inline because of their simplicity: I believe it would be better if they were not!

### Putting it right

#### Correcting the `make_address_data()` Function Design

One way is to delegate the instantiation of `Address_Data` to an abstract factory class [3]:

```
class Address_Data_Factory {
public:
    virtual Address_Data*
        make_address_data(
            istream& data);
};

class UK_Address_Data_Factory :
public Address_Data_Factory {
public:
    virtual Address_Data*
        make_address_data(
            istream& data);
};
```

```
class USA_Address_Data_Factory :
    public Address_Data_Factory {
public:
    virtual Address_Data*
        make_address_data(
            istream& data);
};
```

This will lead to a proliferation of classes but I don't think that is a problem in this case. The classes are all "weightless" and adding a new country is very simple. The address constructor which was

```
Address::Address(
    String country,
    istream& init_data=cin)
: the_works(
    make_address_data(country,
    init_data)) {
}
```

now becomes

```
Address::Address(
    Address_Data_Factory& f,
    istream& init_data=cin)
: the_works(
    f.make_address_data(init_data))
{
}
```

Now, all clients have to do is instantiate the correct factory, and pass it to the Address constructor: much safer than using a `String`. By the way, I'm not taking any account here of making parameters/functions `const`; it's possible that the factory class functions should be, but I'm not thinking through that kind of detail.

You may wonder, why go to the trouble of using an abstract factory; why not let the client code instantiate the appropriate `Address_Data` class? The answer takes us back to an important reason for using the `Address` class in the first place: to hide the implementation detail. Using the factories will not expose any such details.

### Inline Functions in the Surrogate `Address` Class

I disagree with Francis' assertion that the Address class functions can be implemented inline because of their simplicity. Functions should only be inlined if it proves necessary to achieve the required performance. In other words, inlining should, in my opinion, be treated as an optimisation technique, and optimisations should only be applied to working code which is not meeting it's performance targets. Some observations about this case:

1. they only complicate the class definition, making it harder to read

2. making them inline advertises implementation details that clients need not be concerned with.

3. performance will not be a problem. The time taken to call a wrapper function is insignificant compared to the time to execute a function like `clone_address_data()` (which will ultimately allocate memory - an expensive operation).

4. Any change to the implementation of any of these functions, requires recompilation of Address clients. Admittedly this is unlikely to be a problem: the functions are little more than forwarding functions.

## Conclusion

Many aspects of class design are subjective. It is important to remember that the above comments are my views. It would be interesting if other people were to follow this up and present theirs.

## References

1. Francis Glassborow, "The Uses and Abuses of Inheritance", Overload 17/18.

2. Scott Meyers, "More Effective C++" (i) Item 35 "Familiarize yourself with the language standard" (ii) Item 5 "Be wary of user-defined conversion operators", Addison-Wesley.

3. Gamma et al, "Design Patterns: Element of Reusable Object-Oriented Software" p87 "Abstract Factory", Addison-Wesley.

*Mark Radford*
*mark@twonine.demon.co.uk*

# The Uses and Abuses of Inheritance - Roger Lever & Mark Radford

*The following questions were posed by Roger. I asked Mark to address them as he had submitted the above article on this subject. - Alan.*

Having read Overload 17/18 and Francis Glassborow's article: "The uses and abuses of inheritance" I must confess to having some questions ☺.

### Copy Constructor and Assignment Operator

Is declaring a copy constructor private for an ABC necessary? The author states that this, along with the assignment operator, are removed in terms of functionality "...because you cannot copy an abstraction". However, an ABC (a class with at least one pure virtual function) is not instantiable, it cannot be used as a return type or parameter or a reference which requires the object to be instantiated.

It can be declared as a pointer. In short I do not see why they are declared private (and not implemented). In a concrete class I think that is good practice, and would advocate it, but this is an ABC?

*There is a need to do this with the assignment operator (if the class had any data members you would make it protected). The problem is demonstrated by the following:*

```
void copy_abstraction(
    Address_Data& to,
    const Address_Data& from) {
    to = from; // Oops!
}
```

*Making the assignment operator virtual doesn't help either: it wouldn't stop a* `USA_Address_data&` *from being assigned to a* `UK_Address_Data&`.

*The copy constructor must be private if you wish to prevent derived classes from generating their own copy constructors (remember that sub-objects are copied using their own copy semantics).*

### Is the Surrogate Really Needed?

make_address_data() seems very clumsy, is this a better approach? The design put forward is clearly trying to address design issues. However, it is not clear to me what exactly the cost/benefit is and why this is better than a straight polymorphic solution? Using a class `Address` that is an ABC and deriving UK, US... addresses which are handled polymorphically seems fine.

*There is a better approach: you decouple the instantiation of the classes from the classes themselves. See my article [above].*

*Note that Francis' address class not only hides the implementation detail, but also takes responsibility for deleting it. You can create automatic instances Francis' address class and not have to worry about managing pointers. Talking of automatic instances, that's another advantage of the surrogate class approach: you can have automatic polymorphic objects created "on the stack". This is in addition to the Address class removing the need for exposing the address implementation details.*

```
class Address {
    virtual makeAddress() = 0
    virtual cloneAddress() = 0
   // Whatever else is appropriate
   // to the abstraction
};
class UKAddress: public Address {
    // UK implementation
};
class USAddress: public Address {
    // US implementation
};
```

Therefore each concrete Address is handled via a polymorphic call which will behave differently dependent on the type of object instantiated. This approach will require the "client" to instantiate and delete the object afterwards but this could be handled via smart pointers. The assertion that this may be "...a feature of a poor design" I do not really understand. It appears that this may be a very neat solution and if resource acquisition/release strategies are employed it may be a very elegant solution.

*True, smart pointers could do the job, but this would require the concrete classes, implementation and all, to be exposed to the client code where the class is instantiated. One of Francis' objectives was to hide implementation detail from the client code.*

I remember (vaguely) a rule of thumb stating that any time you find yourself using a giant switch statement to decide the next action, like `make_address_data`, one should step back and see if a polymorphic solution is more desirable. This is in fact contrary to the emphasis of the article. Of course, as a minor point, given the multiple possible conditions of "country" it should use a switch

statement rather than `if`s, since this is then one evaluation of country rather than many.

*My article covers this.*

### Why Use Friend?

Why use friend class `Address_Data`? Clearly the implementation of `Address_Data` has been hidden however, it may be desirable that there is a public interface to some of `Address_Data`'s data. That question can only be answered in the context of other requirements and the design of a solution. However, I do not understand why `UK_Address_Data` needs to be a friend? I know that friends provide access to private data and what the principles are but why is that good or useful here in this example? It appears to me to introduce an unnecessary level of coupling when in fact `Address_Data` could have declared its data protected. Alternatively, it may have been useful for other classes to have access to the data via an interface.

*`UK_Address_Data` declares the base class `Address_Data` as a friend. That means that `Address_Data` has access to everything in `UK_Address_Data`. Therefore, I don't see why `Address_Data` declaring it's data as protected changes anything in this case. I guess the reason for the friend is that Francis appears to have made everything in `US_Address_Data` private (I must confess I overlooked this when designing my solution, which might need some adjustments). So, in order for the static member function `Address_Data::make_address_data()` to perform the instantiation, the friendship is needed to make the constructor accessible.*

### Checking for Self Assignment

Why does the `Address::operator=` not check for self assignment? It is possible that "`a`" is the same as "`this`" and should check for that condition prior to deleting "`this`"? Whether there is any real danger here seems to be dependent on how `clone_address_data` is implemented?

*It looks like self assignment should be checked for. Was this omission accidental or deliberate I wonder? Or am I missing something which makes it unnecessary. The way Francis has implemented the `clone_address_data()` function it looks like self assignment doesn't do any harm, but just wastes time and processing. I don't normally bother about performance unless I have to, but I think self assignment is so trivial to protect against, that this should be done.* ***(Francis provides a better answer below. - Alan)***

### Patterns

Why not use a Pattern style solution and/or presentation? Not having Gamma et al, I cannot quote which pattern this would relate to in their book, is there a surrogate pattern? However, it seems that the Address problem is evocative of a Factory or Bridge pattern solution. If the presented design is not a pattern it might have been

*Gamma et al give Surrogate as an alternative name for the Proxy pattern, where a Proxy/Surrogate (`Address`) class controls access to its "Real Subject" (`Address_Data`). Francis' design exhibits characteristics of both Bridge and Proxy patters. My attempt at improvement, uses the Abstract Factory pattern too. Design patterns capture known solutions to known problems, providing designers with not only standard solutions, but with ideas on which they can draw when seeking solutions. The Bridge pattern is about de coupling abstractions and their implementations. One of the applications of the Bridge given by Gamma et al is, in C++, to hide away a class' implementation details; in C++ there is no facility for doing this provided by the language. Giving the `Address` class responsibility for deleting the implementation is a characteristic of the Proxy pattern. Gamma et al call this application a "smart reference". The `Address` class in Francis' design simply forwards requests to the implementation object, but it is possible that `Address` could be extended to service requests which require more that one request of the implementation object; this would use the Adapter pattern.*

***(The pattern style of presenting "forces", "solution" & "examples" is very good at explaining the reasons for and applicability of a particular design idiom. It is also hard work! I suspect Francis was more concerned with presenting some design issues for consideration than delineating what he admits is a flawed design. - Alan)***

## *In Conclusion - Mark Radford*

*Let's state again, for the record, what we all should know by now: software design is hard! Anyone who thinks otherwise is just kidding themselves. The search for a solution to a design problem (in this case, the problem is to provide polymorphic objects without exposing their implementation details) in most cases will turn out to be much more involved than it looked on the surface. Each time we think of an idea, more issues turn up to tax our brains.*

*Francis' original article in Overload 17/18 and attempting to answer these questions has certainly made me think a lot, as did writing my own article. Roger has been thinking about it too, which is why the above questions were asked. Therefore, I believe something was achieved.*

*Roger Lever* *rnl16616@ggr.co.uk*
*Mark Radford* *mark@twonine.demon.co.uk*

## The Problem of Self-Assignment - By Francis Glassborow

A number of responses to my article in the last issue leave me rather concerned. I guess the fault is mine for not documenting some of my code better. The purpose of this brief article is to revisit the issue of writing an operator=() for a copy assignment.

In general you will only do this when the compiler-generated version will be suspect or plain wrong. There are several possibilities:

1. The class includes a pointer or a reference data member
2. The class includes a const qualified data member
3. The requires semantics is not that of an exact copy - for example there is a unique instance code, a copy count or some other piece of data that must not be copied.
4. You wish to provide a more sophisticated strategy for copying such as lazy copying.

The most books all teach you to start your copy assignment like this:

```
Mytype& Mytype::operator=(
            const Mytype& RHS) {
    if (&RHS == this) return *this;
    // code to copy as required
    return * this;
}
```

or the equivalent

```
Mytype& Mytype::operator=(const Mytype& RHS) {
    if (&RHS != this) {
        // code to copy as required
    }
    return * this;
}
```

Some programmers spend an inordinate amount of time debating the relative merits of the above alternatives. Instead they should be asking themselves about why they wish to use either of them.

In both cases the cost of making the check for self-assignment is some kind of comparison and branch statement. Branches are bad news on pipelined architectures. If we can write clean code with fewer branches we should do so.

By the way it is worth noting that switch statements can have a really bad effect in such environments. If you know that one case will occur more than 50% of the time and need maximum performance, it is worth thinking about making that a special case tested with an if statement before entering the switch. That helps by allowing the processor to speculatively process that branch. Such special treatment is only worth considering in low level code that will have a large impact on many applications (such as code for an operating system).

Usually the critical feature about self-assignment is the risk that data is deleted before it has been copied.

Suppose that I had written the following for my `Address` class (see page 7 of Overload 17/18):

```
Address & Address::operator=(const Address & rhs) {
    delete the_works, the_works = rhs.clone_address_data();
    return *this;
}
```

(By the way there was an error in the published code. See if you can spot it before I provide the corrected version.)

The above code falls over in the case of self-assignment because you just threw away the data for the left-hand side, which in this case was also the data for `rhs`. The second part of the statement now fails catastrophically because `Address_Data::clone_address_data()` accesses an invalid pointer. The naïve way to fix the problem is to check for self-assignment. E.g.

```
Address& Address::operator=(const Address & rhs) {
    if(&rhs != this)
        delete the_works, the_works = rhs.clone_address_data();
    return *this;
}
```

*(I feel that braces would make this code clearer than the comma operator! - Alan)*

Now consider the (corrected code) for the `operator=()` I wrote.

```
Address& Address::operator=(const Address& rhs) {
    Address_Data* temp = rhs.clone_address_data();
    delete the_works, the_works = temp;
    return *this;
}
```

*(This has the added benefit of leaving the object in a consistent state if an exception is thrown during the clone operation. I'd rate this as more important than worrying about the different number of processor cycles required for each version. Alan)*

I first copy the data handled by `the_works` to `temp` (note that I got the type of `temp` wrong in the original article, result of upgrading old code with better thought out names:) At this stage there are always two full copies of the data for `rhs`. I now throw away the data for the left-hand side. If that discards the data for `rhs` that is completely irrelevant - it can only happen when both operands are the same object (`Address`).

Finally I transfer ownership of the copy from temp to `the_works`. In the case of self-assignment this re-instates the data for `rhs`.

As long as cloning does the right thing this will always work. If this algorithm does not work (i.e. does the wrong thing; it won't produce undefined behaviour) for self-assignment then you have a fundamental design issue as to why the semantics of self-assignment is different from the semantics of other copy assignments.

The benefit of this algorithm is that there is no branch statement. On most hardware, the extra assignment of a pointer will produce smaller code than a comparison test. Even without the pipelining consideration a pointer assignment will normally run faster than a comparison and branch. The slightly larger stackframe (storage for `temp`) should be insignificant.

The cost is that you will take longer to do a self-assignment. But why should every assignment pay to allow you the rare benefit of doing nothing? If this cost (of self-assignment) is significant in your code perhaps you should consider testing for self-assignment before you make the assignment. For example:

```
    if (&lhs != &rhs) lhs = rhs;
```

Is this an attractive solution? Of course not, so why advocate the equivalent in the function called?

Remember that the critical pattern is using delete on a pointer before using it to handle a newed copy of something. Avoid the knee-jerk reaction and think about making the copy first and holding that copy with a temporary pointer while you delete the original.

# Borland C++ Builder for expert programmers - by Eric Richards



## Introduction

At last a Rapid Application Development tool is available for C++. *(This seems to ignore Blue Sky's WinMaker & VisualProgrammer, PowerSoft's Optima++, IBM's VisualAge C++, and possibly others - Alan)* This latest C++ compiler includes a C++ version of the class library developed for Delphi. All the features you are used to in C++ are still available with more additions than you may imagine. All of Windows functionality is available, with a far more intuitive event-driven interface, to provide graphical interfaces and system features such as drag-and-drop and OLE with extreme ease. Database functionality galore: handle either native databases (Paradox, dBase, Interbase) or many other databases in much the same way, with or without using SQL. A completely revised Integrated Development Environment that looks just like Delphi 2.0 brings genuine visual programming to C++.

## An important language development

Not having the resources to catalogue all the benefits of Delphi I will stress the importance of components. Components? In order to understand these you need first to understand a couple of extensions to ANSI C++ syntax that are absolutely necessary. The 1994 decision to stop adding features to C++ cannot stop genuine progress. Two vital new keywords are __published and __property . Being non-standard they are dutifully prefaced with a couple of underscores (but you could use the pre-processor to get rid of them) .

'published' is a privacy level that is similar to public except that run-time type information is available for an identifier so declared without needing to specify anything further. *(Is this anything in addition to standard in C++ RTTI? - Alan)* A 'property' is a class data member that can be accessed syntactically with assignment statements. But it actually uses appropriate getter and setter functions which are called automatically according to which side of := the property occurs.

These new keywords extend the C++ notion of a class, enabling its instances to be 'components' . Components are all derived from a powerful ancestor class with much of the functionality implied in the next section and they contain published properties.

A published property enables an end user, or a naive programmer, to supply a component object with parameterised data at run time or development time respectively.

## Borland C++ 5 comparison

The notion of a published property appeared in BC 5. But there it was not part of C++. The documentation involved an inelegant excursion into a scripting language, which I considered unnecessary hard work.

Many of the controls in C++ builder and Delphi were also introduced in BC5, but only as part of a dialog box editor. Thus the interface was not as smooth as in Delphi and C++Builder. (I believe a scripting language has been used internally to produce C++ Builder, but no trace of it remains visible.)

## A massively altered IDE

Like Delphi. the much enhanced Integrated Development Environment now contains three features new to C/C++ programmers, as well as the customary code editing area and menus :

1. 'Forms' which are versatile display windows capable of forming visual interfaces. They are a generalisation of the screens used by Windows programmers when using programs like Resource Workshop. Each screen is associated with some code. (Windows programmers: do not jump to the conclusion that it bears a resemblance to a windows procedure)

2. A multi-paged 'palette' containing around 100 iconised components that can be dragged onto the screens. Many of these components are little windows used for all the rectangular areas that you must have seen as a user of Windows even if you have not programmed them. But many more are nothing to do with display, they are actual programs which have been encapsulated as components, and their user interface is via the aforementioned properties.

3. The 'object inspector' can display both the properties (data with RTTI) and the events (triggered programmatic responses to real-time occurrences) of all the components you use, including the forms themselves. Dragging and dropping components and supplying their property values can take you a

surprisingly long way. A more advanced use of this class library is to write the real-time response procedures.

## Under the bonnet

When you create a new form from the menu you are in fact instantiating an instance of a new type derived from the parent class for forms. As soon as you do this, not only can you see a design-time onscreen representation of the form that will appear at runtime, but also the code which defines the class and its instance code in the Editor window. Then when you drag a component from the palette onto this form representation you actually include one object inside another in the declaration of the form's class , and you can see this in the code too. The code and the picture are automatically kept in step at all times through the compiler's background processing capability.

## What you do

C++ builder is an event-driven system, the event model is much more intuitive than the underlying Windows system. You can produce simple applications without defining any events yourself, this is a selling point for non-programmers. But usually you will need to define some event-handlers, if only to close the program neatly. So having got some components together by visual manipulations with your mouse, then just clicking on a so-far undefined event in the object inspector creates in the code editor the declaration of a suitable class method in a .H file and the corresponding outline definition in a related .CPP file. These function headers will look strange at first but to start with you can cheerfully ignore them ! You just insert your algorithmic ideas into these outline definitions.

To give a couple of examples: if you want something to happen when you click on a button, you supply the definition of that button component's OnClick event handler procedure, or if you want something to happen when you close a form you would supply the body of that form's OnClose event handler procedure. Previous arcane rules for Windows event handling are replaced by the intuitive application of common sense.

Various supplied components encapsulate input and output, all the controls you see in Windows 95 and Windows 3.1, system controls to implement OLE, DDE, Multimedia, and timers. There is a market in third-party controls for specialised applications. Some controls not mentioned here are provided in Delphi but not in the pre-release version of C++ Builder on which this article is based.

(Last-minute addition: My non-disclosure agreement re Delphi 3 will end on Feb 25, before this will appear. This contains among other things a new way of presenting multi-dimensional data and extensive support for Active X. Even if these topics turn you on, I would advise C++ programmers to start with C++ builder to minimise mental indigestion.)

## Databases

There is nothing to stop you handling your data with tools such as the template library, or writing directly to files. C++ Builder provides extensive alternative facilities for those wanting to interface with, or even to replace, many commercially available databases from dBase to Oracle. This is a huge topic which I cannot do justice to here.

Very briefly, twenty control classes are provided which encapsulate the functionality of the Borland Database Engine and also of a number of supporting tools for related activities like report-creation and database upsizing. The beauty of it is that nearly everything you do is independent of the particular brand of database, the expensive versions of C++ builder provide many different drivers which enable you to treat them all in a similar way. All versions provide everything necessary to create applications in two native desktop and one transaction-oriented SQL database.

## and C++ , similarities and differences.

Every one of the components is documented, adequately but concisely, and so you get to know the Standard Component Library which seems identical in C++ builder and Delphi 2 . Because source code is interchangeable between the two languages at .obj level no translation of the SCL will have been necessary, and C++ programmers will get a well proven system from the outset.

As soon as I saw a pre-release version of Delphi 1.0 I was sold on the system, even though as a C/C++ programmer it meant working in Pascal, but you don't have to do that any more.

For readers of this journal the benefits of working in C++ include familiarity with the language and the existence of a huge base of existing applications.

The benefits of working in Pascal have been getting a head start, and a faster compiler (claimed to be the fastest in the world). Pascal is not lumbered with

much old stuff kept mainly for backwards compatibility and its clean syntax is inherently quicker to compile. (and now Delphi 3 mentioned in above last-minute addition) However there is no difference at all in runtime performance.

The most significant syntactic difference for any C programmer who browses the Delphi literature for inspiration is as follows. Think of what happened to your C mindset when C++ introduced passing by reference. The new level of implicit de-referencing sets you back at first, but soon appears a good way to get rid of over-dependence on pointers. There is a very similar feature in Delphi Pascal whereby pointers to classes are implicitly dereferenced in such a way that you keep seeing dots linking variables where as a C programmer you would expect arrows, and this turns out to be much easier once you get used to it. But as a C++ Builder user you will not have to get used to this. Instead, just keep on using dots or arrows as appropriate like you have always done, realising that Delphi /Pascal sometimes does it differently.

## Conclusion

You should accept the fact that this way of programming is so much more productive and powerful that it is going to be the way of the future. Now that C++ and object-orientation have come of age, junior programmers and end users can easily use components to create simple but sound systems, on account of their intuitive interface, but few non-programmers will have time to learn all the techniques now available.

It takes programming skill to actually write components, using the notions indicated at the beginning of this article, and it takes wisdom to know when to bother. An advantage of Delphi and friends is that competent programmers can use the same skills to handle a variety of different commercial databases. There is not a lot to choose between using C++ Builder or Delphi, and a mixed team could work together.

*Eric Richards*

*I find that very few of the programs I develop would benefit from such tools (and have discussed this with Eric in the past)!*

*Certainly the when I reviewed the pre-release of Delphi1 it offered little for my needs that was not addressed by VisualProgrammer. (which had been around quite a while and allowed me to write the "real code" in C++.) Clearly, if you application is nothing but GUI & database access your mileage will vary. - Alan*

# Make a date with C++: In the Beginning... by Kevlin Henney

Many readers of *Overload* have identified the need for more introductory articles on C++. I hope this new series of articles will go some way to meeting that need – if not, please get in touch. One premise of the article is that readers have some familiarity with C or C++.

Given the year 2000 problem (farce?), date handling is very much a vogue topic and one I have found is a sufficiently rich seam to mine for examples on minor and major language features, method and technique, and common understanding (or, occasionally, misunderstanding). I will progressively try to build up the example by discussing some interesting implementation details – so hopefully this, combined with some language trivia, should still keep the interest of more advanced readers.

Having set the aims for series, and the prerequisites for the reader, it is worth making sure we agree on the ground rules: unless otherwise stated, the Gregorian calendar is being used for dates and we will quietly gloss over certain minor locale irregularities – such as the calendar system first being introduced to France in 1582, but only adopted later in certain countries (for instance, Britain in 1752 and Russia in 1918). Assuming that this calendar system has always been in existence, and contains a year 0 (which astronomical calendars, but civilian ones do not) will also keep things from getting needlessly complex!

### Leap years

Key to any date handling package is the classification of whether or not a given year is a leap year. Rather than embed the logic in every piece of code that seems to need it, we factor this out into a function with the prototype

```
bool is_leap_year(int year);
```

This would go into a header file. For C programmers an important introductory perspective on C++ is that in many ways it is a "safer C". It is more strongly typed than C, with a requirement that all things must be declared before use. In this case what is considered

good practice in C (i.e. declaring function prototypes before use[2]) is enforced in C++.

The only other thing to note is the return type of `is_leap_year`: `bool` is the built-in type for Booleans. `bool` takes the values `true` and `false`, but may also be mixed with other scalar expressions with predictable results (`true` maps to `1` and `false` maps to `0`). As an addition to the language it is not that recent (1993), but you will find a number of compiler vendors have been a little slack in introducing it – if your compiler does not support it, either use `int` or kludge the `bool`, `true` and `false` keywords with a `typedef` or macro. It is likely that `bool`, `true` and `false` will in some form make it into C9X, the next C standard, most likely as reserved library identifiers rather than as new keywords.

Function definitions are pretty much the same as in C, except that the old K&R form is not permitted – you must use the prototype form. The correct leap year algorithm is defined as any year that is divisible by 4, and either divisible by 400 or not divisible by 100. This means that 1984 and 2000 are leap years, and 1983 and 1900 are not.

```
bool is_leap_year(int year)
{
  return year % 4 == 0 &&
   (year%400 == 0 || year%100 != 0);
}
```

### Go faster stripes

Speed freaks have a nasty habit of turning all short functions into macros in C. With a bit of judgement and inspection (profiling, looking at generated code, etc.) some functions can be identified as time critical and in need of low level optimisation – such as losing the overhead of a function call – as opposed to high level optimisation – selecting different data structures and algorithms.

The problem with the optimisation approach adopted in C is the use of macros – if you are not familiar with the problems of writing and using macro functions I will assume you can't be using them, which is good thing! They do not look or behave like functions, which leads to careless mistakes such as terminating the macro definition with a semicolon, forgetting to use a backslash to continue onto the next line, and forgetting to place all uses of arguments as well as the whole macro in parentheses (by definition a well written macro is an unreadable one). Some of these mistakes are immediate compile time irritations, with associated cryptic error messages, whilst others can lie dormant in code for a long time, waiting to surprise and confound some poor unsuspecting victim.

And this list of problems does not even begin to cover the issues you can't work around: multi-line macro functions that return values, debugging, avoiding side effects in re-evaluating arguments, and the to convert a macro easily back into an ordinary function.

C++ offers an alternative and far superior mechanism in the form of inline functions. These look and behave exactly like ordinary functions with the minor difference that they are specified with the `inline` keyword:

```
inline bool is_leap_year(int year)
{
  return year % 4 == 0 &&
   (year%400 == 0 || year%100 != 0);
}
```

This gives the compiler a hint that the function code should be expanded out at the point of use and optimised accordingly. Note that this is only a hint, and the compiler is entitled to ignore it and expand the function out of line (i.e. traditional function compilation). This is not altogether a bad thing as many programmers (novices and the more experienced alike) tend to inline everything in sight out of a mixture of feature novelty and ease, ignoring the generally undesirable side effect that the overall size of generated code may well increase if all calls are expanded out in place. Code bloat can be a problem with careless and excessive use of inlines.

The only other thing you need to be aware of is that to optimise a function at its call site its definition must be available to the compiler. This necessitates one change to your code: place the inline function definition in a header file if you wish to make it available to all.

In theory in both C and C++, any `static` function can be optimised to be inline if the compiler sees fit. In theory this makes inline functions redundant. In theory there is no difference between theory and practice, but in practice there is: you will find few mainstream compilers that either do this or do this well. There is the argument that a good compiler should (so what do you do if your compiler doesn't) and that if you wait long enough all compilers will get there (a sort of "ideal world by bus" argument). Leaving aside the challenges of modern physics, it is fair to say that we live in the present. The `inline` keyword allows you to tag certain functions for

[2] If you are not already doing this in your C code, get with it!

possible optimisation and many compilers offer you a choice of command line strategies, such as:

- Perform no inlining at all;
- Inline only those functions specified as `inline`;
- Inline at will.

Explicit inlining can be a useful feature, especially as it displaces the addiction to macros that many programmers seem to have. It is also becoming an increasingly common extension in C compilers – as it codifies existing practice it is likely that some form of `inline` will make its way into C9X.

### The main thing

So now we can write code to use the `is_leap_year` function:

```
int main()
{
  cout << "Please enter a year: ";
  int year;
  cin >> year;

  if(is_leap_year(year))
  {
    cout << "Leap!!!" << endl;
  }

  return 0;
}
```

OK, so this isn't a very exciting program, but it shows the function in action. It also shows the use of the C++ I/O streams facility, for which you will need to include `<iostream.h>`[3]. The bitshift operators have been hijacked/borrowed/overloaded for shellscript-like I/O syntax, the gritty details of which I will not cover yet. The `cout` object is the destination for output, while the `endl` manipulator writes an end of line and flushes the output. There are also `cerr` and `clog` output objects for writing out or logging errors (unbuffered and buffered, respectively). Not surprisingly, `cin` is the source for input. Hopefully, you can see some correspondence with the standard C `stdin`, `stdout` and `stderr` streams.

Returning to the subject of good practice being enforced, the implicit `int` rule – where no declared return type implies `int` – has (relatively) recently been dropped from C++, so `main` must be declared to have a return type. Compilers may (may? will!) differ in their compliance.

[3] The draft ISO standard defines a slightly different convention, but you will be very hard pushed to find a compiler that does not implement the original `<iostream.h>`.

The physical structure of a C++ program is similar to that of a C program, with header files and source files. The main difference is the file suffices: where C uses `.c` and `.h`, there is no single convention for C++...

- Source file suffices include `.cpp` (de facto standard on PCs), `.cxx`, `.cc` and `.C` (only of any real use on systems with case sensitive file names).
- Header file suffices include `.hpp`, `.hxx`, `.hh`, `.C` and `.h` (a convention I'm personally not keen on, as it then makes it impossible to distinguish between a C and a C++ file without content inspection).

### Summary

- C++ is more strongly typed than C, requiring all functions to be declared before use and banning the implicit `int` rule.
- C++ has a `bool` type with `true` and `false` constants, although your mileage (or more specifically, compiler conformance) may vary.
- Appropriate use of `inline` functions obviates the need for the majority of macro functions.
- The I/O streams library provides an I/O facility with a shell like syntax.

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

# 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!

## C++ Committee Draft

The following announcement was made on USENET:

In January I posted the press release announcing the availability of the C++ Committee Draft for public comment. At the time, the only way to get a copy of the draft was to purchase one from ANSI or from Global Documents.

ISO policy as of last year was to prohibit free access to these documents except to committee members. I was assured at that time the although the policy might be reviewed and might be changed in the future, there was no possibility that the policy would change before the end of the C++ public comment period.

ISO has changed its policy sooner than anyone thought possible, and copies of the Committee draft are now available for downloading.

You can find complete instructions on downloading a copy of the draft and the procedure for submitting comments at:

<http://www.setech.com/x3.html>

<http://www.maths.warwick.ac.uk/c++/pub/>

---
Steve Clamage, stephen.clamage@eng.sun.com
chair, X3J16 C++ Committee

## The Casting Vote - by Sean A. Corfield

March, 1997. The second Committee Draft Ballot is in progress so, strictly speaking, WG21 can do nothing but wait for the National Body comments to come in with the votes in June. However, the second ANSI Public Comment period has just completed so X3J16 had something to do this week, reviewing what the public has said about C++.

Mostly, the public want typos or small editorial stuff fixed but we had the usual batch of extension requests. Some them were proposals that we have already considered and rejected while others were novel or downright weird. So I'm going to start by talking about some of things we didn't add to C++, just for a change!

### The : operator

Isn't it annoying how often you want to write `x ? x : y` but don't want `x` to be evaluated twice? Someone thought so and suggested we add the `:` operator so that `x : y` means `x ? x : y` except that `x` would only be evaluated once. They went on to suggest allowing `x ? y` to mean `x ? y : 0` and then to say that if `||` and `&&` worked 'properly' they would already have the behaviours proposed for `?` and `:` respectively. Rejected: too late for extensions.

### Binary literals

This proposed adding some minor lexical enhancements: binary numeric literals prefixed by `0y`, underscores in numbers (partly to make the proposed binary literals easier to read!) and the `\e` escape sequence to represent `ESC`. These had been proposed and rejected several years ago, along with `\dNNN` to represent decimal value escape sequences and a few other goodies.

### Cloning objects with `type_info`

This proposed adding the following member to `type_info`:

```
void* type_info::clone( const void* )
```

Then objects could easily be cloned by writing:

```
template<typename T>
T* clone_ptr( const T* p ){
  return (T*)typeid(*p).clone(p);
}
```

Despite its utility, this and many other additions to `type_info` have been considered and rejected over the years. At one point, implementations were at liberty to provide `extended_type_info` with functionality like this (e.g., one compiler vendor looked at providing function pointers for accessing constructors, destructors and so on), but that approach has fallen out of the draft at some point. I've raised it as an issue to try to reinstate it.

## Comments'r'US

The bulk of the meeting therefore was taken up with considering possible solutions to the US Public Comments. The intent of X3J16 was to produce a list of official comments with suggested resolutions where possible. Each National Body will be going through a similar process at the moment and in July we will have a more complete list of comments and suggested resolutions.

Most of the US comments relate to small fixes but there were a couple of issues which have broader impact: exception handling in the library, allocator pointer types, default arguments on template member functions and `void` return types.

## Exceptional policy

As several people have commented, the library effectively has no policy regarding exceptions: if a user type throws an exception while being manipulated by a library function, the program has undefined behaviour. Not surprisingly, many people would like to see some guarantee that a program won't fall over with its legs in the air when the first exception is thrown! Discussions in Nashua suggest that as long as your types don't throw exceptions during destruction and your iterators don't throw exceptions when "used in valid ways", then you stand a reasonable chance of the library behaving reasonably when your types do throw exceptions.

## Can you point to it? (again)

A major concern after Kona was the restriction that the `pointer` member of the allocator template had to be a real pointer type in order to work with the standard library. Matt Austern of Silicon Graphics worked hard between meetings to analyse exactly what the library assumed of allocators and steered the discussion towards a possible solution that would allow substantially useful user-defined pointer types to operate with the standard library components. More on this after the London meeting.

## Faulty defaults

Some time back, the Core WG reaffirmed the intent of the WP regarding default arguments: their semantics are checked at the declaration in which they first appear. This had the slight inconvenience of breaking the standard library!

The library currently has 54 occurrences of code like:

```
template<typename T>
class Thing {
public:
    void func(const T& = T());
};
```

Try instantiating this with the following type and see what happens:

```
class A {
public:
    A(int);
};
```

According to the library, that should work as long as you always provide an argument when you call `Thing::func`. Unfortunately, according to the language, it will fail at instantiation time (because `T()` is invalid when `T` is `A`). After some sparring between the Library and Core WGs, there appeared to be no consensus on where the change should be made. Again, we'll hear more of this after the London meeting.

## Nothing revisited

If you've tried to use the function objects in STL, you've probably hit the problems with trying to use the `void` type inside templates: you can't return a void expression from a function that has a `void` return type. If you haven't hit this, you just aren't using enough templates!

Stroustrup brought a proposal to Kona to relax the rules concerning the use of `void`. It was not accepted then but it was sympathetically received. The issue came up again in the public comments and now it looks likely that the resolution of the issue will involve at least some adjustments to the rules on the use of void expressions and `void` types. Personally, I'd like to see declarations allowed with `const void&` (as long as they are not "used") since that's the main problem I hit with templates and `void`.

## The future

The next joint meeting will be held at the BSI offices in Chiswick, London on July 13-18, sponsored by

Programming Research. It will be a busy meeting: we have to try to resolve all the National Body comments on CD2 (which we hope will pass).

*Sean A. Corfield*

*Technical Director*
*Object Consultancy Services*
*sean@ocsltd.com*

## New Keywords For…by The Harpist

Over the last few years the C++ Standards Committees have introduced several (actually quite a lot( ) of new keywords into C++. In general the motives for each addition have been very good. Now we are beginning to get compilers that actually support these new keywords it is time that the ordinary programmer started using them. In order to do so you will need to know what problems the keywords were introduced to tackle. The latest version of Visual C++ (5.0) seems to support most of the features of C++ though there are numerous deficiencies — we could do with reports of places where popular compilers do not comply with the language specification. This potentially helps in two ways. First the working programmer can learn what they should be getting as distinct from what is delivered (this matters because it helps programmers build the right mental models). Second we can expose the deficiencies and help bring more pressure on implementors to provide correctly implemented tools.

Before I get into the meat of this article, let me mention two deficiencies that are still present in VC++ 5.0 despite all that has been said and written before.

If you do not provide an explicit return from main the compiler still generates a completely incorrect warning. The language actually specifies that exiting main without a return statement shall be deemed as returning `EXIT_SUCCESS`. There is simply no way for a correct implementation to get this wrong. It maybe clumsy for a programmer to leave out a return statement (please no calls to `exit()`, that was fine for C but is quite inappropriate to C++ even if it seems to work.) but it is correct code and so does not merit a diagnostic (though one would be helpful for any other function that had a return value but no return statement—see sidebar). The actual warnings that VC++ gives (different depending on whether you do or do not explicitly specify `int` as the return type from `main()`) are actually either admitting that the compiler has wilfully miscompiled your code or advises you to write non-conforming code. This is highly significant because the C++ Standard can impose no requirements on a compiler that accepts code that includes `void main()`. The very existence of such a line makes your code non-conforming and hence all consequential disasters are your own fault. I strongly suggest that:you loudly and vociferously complain about this fault at every opportunity you never write code that includes `void main()`

By the way if VC++ reinterprets your code as if you had written `void main` (because you had relied on implicit `int` for your definition of main then you have them right in your sights. The compiler has blatantly changed your conforming code to non-conforming code.

The second irritating wart in VC++ 5.0 is that four years after the change of the scope rule for a variable declared in a `for` statement, VC++ still implements the old version. The longer this continues the more incorrect code will be written. I am getting more than a little tired of still having to write '`{for`' and closing with '`}}`' to ensure that my code behaves the way I want it to, and the way the language says it should. The four-year delay has probably increased the amount of suspect code by at least a factor of 10 (and more likely a factor of 100 or more). What is worse is that those making the mistake are exactly those that will find it difficult to detect and correct. Oh, for the benefit of those who do not know what I am on about:

```
void fn (){
    int i=0;
    {
        for(int i=1; i<10; i++)
            cout<<1/i<<endl;
        cout<< 1/i<<endl; // A
    }
}
```

This function was well-formed when the scope of a loop variable was from the point of declaration to the closure of the enclosing block (in such circumstances the value of `i` in line A is 10. In modern C++ line A results in undefined runtime behaviour (no diagnostic required) because the `i` is now the outer one that is still 0.

### Sidebar:

**Functions without return statements**

It is only an error to actually return from a function with a return type without an appropriate return statement. Read that carefully. Then think about the ways that it is possible to return without using a return value.

First of all, in K&R C there was no `void` return so it became idiomatic to omit the return statement from a function that would, in modern C, return nothing. It was only an error if the putative return value was used at the call site. In other words the following code was fine:

```
int fn (void) {
  printf("This is a stub procedure");
}

int main() {
    fn();
    return 0;
}
```

But this is not:

```
int fn (void) {
  printf("This is a stub procedure");
}

int main() {
    int i = fn();
    return 0;
}
```

Actually even then it is the passage of the flow of control through such a function at runtime that is the error, so like it or not, the following should compile and execute:

```
int fn (void) {
  printf("This is a stub procedure");
}

int main() {
    int i = 0;
    i ? i=fn() : i;
    return 0;
}
```

The critical part will never be executed so the program is defective in the sense that it contains some silly code, but there is no reason why a compiler should reject it, nor is there any reason why it should exhibit undefined behaviour at execution time.

There is another case when the lack of a return statement is only of academic interest and that is when the function will never actually return. This might be the case in C when a program finishes with an explicit call of exit() from somewhere beyond the call of the critical function. In C++ this can happen when an exception is thrown through the critical area.

To summarise, it is not the job of a compiler to double guess a programmer. However it is the job of good tools to identify suspicious areas and require some form of sign-of from the programmer.

## Now to some new keywords

### mutable

Sometimes an item of instance data in an otherwise constant object must still be alterable. For example, if you are using some form of delayed or lazy copying there needs to be a copy count to track whether an item is a singleton or currently represents several objects. This copy count must always be `mutable`. In other words, whatever else the compiler does it must not place the copy count into a write locked memory segment. The traditional solution of using a cast to ensure that the item in question was changed simply does not work. Writing to `const` protected data by using a cast results in undefined behaviour. I know that the language now explicitly allows you to change the *cv* qualification of an object, none-the-less doing so to an object that was declared `const` results in undefined behaviour if you then seek to modify the instance data. That is not the same as casting away `const` protection from a reference or pointer to `const` object because theoretically you might know that the underlying object had not been 'write' protected. Example:

```
class WithCount {
    // other data
    mutable int modifiable;
public:
    void increment_count() const
           { modifiable++ ;}

    // other member functions
};

int main () {
    const WithCount wc;
    wc.increment_count();// line B
    return 0;
}
```

Despite `wc` being declared `const`, you can call the `increment_count()` member function because it is a `const` member function. None-the-less it may change the modifiable instance data because that has been explicitly declared as mutable. With this feature in place it should never be necessary to cast away `const` qualification of an object, and if you do then any resulting undefined behaviour is entirely your fault. I strongly advocate that you use `mutable` where-ever it is appropriate as soon as your compiler supports it. Indeed you can sort of have your cake and eat it by using `mutable` where-ever it would be correct, using the pre-processor to eliminate it from actual code until your compiler supports it, and cast away `const` in the

meantime. If you leave that cast in for some time after you have a compiler that supports mutable no real harm will be done.

The golden rule is to write correct code now and, if necessary, use the pre-processor to provide a temporary fix. At the same time take every opportunity to demand that your compiler implementor supports C++ as specified and not some historical antecedent.

**explicit**

The process of constructing an object bears some similarities to that of type conversion. In both cases you start with data and use it to produce an object of some type. The difference is that for a type conversion you start with a single argument and —almost certainly— create an anonymous temporary of the required type. Constructing an object may have more than one argument and may —often— create a named object.

The similarity of process has plagued C++ for many years. Whenever you write a constructor for Y that can be called with a single argument of type X it automatically became a user defined conversion from X to Y. The compiler could use it whenever such a conversion seemed appropriate to the compiler. This licence to the compiler is at best unnecessary and at worst fatal. Various hacks —coding tricks— were available to restrain the compiler but when all was said and done they were ugly and made code that much harder to understand. Ideally we should have constrained constructors to be just that, but as so often happens hindsight makes it clear that the default was wrong but it is too late to fix it.

What the Standards Committees have done is the next best thing, they have provided a mechanism for limiting a constructor to being nothing more than a constructor. If you prefix the declaration of a constructor with explicit it can only be used for conversions if you explicitly cast the data to the type. For example, given:

```
struct  Ex{
// various
    Ex (int);
};

void fn (Ex);     //function taking
an Ex by value

int main() {
    fn(1);
    return 0;
}
```

will compile happily and the compiler will call `Ex(int)` to convert `1` into an `Ex` whether that was your intention or not. However if you change `Ex` to:

```
struct  Ex{
// various
    explicit Ex (int);
};
```

`main` will no longer compile unless you replace the call to `fn()` with:

```
    fn(static_cast<Ex> (1) );
```

Strictly speaking making such a change to the public interface violates the contract between class designer and user however it is the kind of interface change that we should accept. If you really cannot accept such changes you can always use the pre-processor to remove `explicit` (by `#define explicit`) from header files but I think that code that relies on such suppression of safety devices is of dubious merit.

Please note that you do not need explicit to qualify conversions the other way because you always have a choice between:

```
    operator int ();
```

and

```
    int convertTo_int();
```

Using the former when you want to provide a conversion operator for the compiler and the latter when you wish to keep control of the process.

While I am on the subject of constructors, a couple of other items. One was a question posed to me by Francis. Is the constructor in the above example (`Ex`) an anonymous function that returns an `Ex`, a procedure called `Ex` that has no return value or something else? Not that the answer matters as such, but perhaps writing `Ex(int)` is a mere lexical convenience and we could just as well have written `(int)` — and have written `~()` for a destructor. By the way many compilers internally provide names such as 'constructor' for the function.

The other issue is that of default arguments. C++ has no need of these for ordinary functions (global and member) because a simple wrapper will do the job. For example instead of

```
    void fn(int = 0);
```

we could write:

```
    void fn(int);
    inline void fn(){ fn(0); }
```

This just makes explicit what the compiler does for you anyway. Once you have overloading in the language you have no need for default arguments. Well not quite, we cannot use this wrapping technique for constructors. For example:

```
struct Ex1 {
    int m_i;
    Ex1(int i =0): m_i(i) {}
};
```

cannot be expanded to:

```
struct Ex1 {
    int m_i;
    Ex1(int i): m_i(i) {}
    Ex1() { Ex1(0) }    //ERROR
};
```

In other words you cannot wrap a constructor. Providing a common body as a called private inline member function may limit the code bloat to some extent but you will still have two (or more) Ctor/init lists to maintain. Not everyone is convinced that such things as default arguments were such a good idea in general and had we realised that constructors really were something special we might have chosen a different path. As it is we are now left with the problem of default template arguments, which in case you have not noticed, are used in the Committee Draft 2 library but nowhere (that I can find) does the clause on templates tell us how these should work. It seems the library writers must have generalised from ordinary default arguments to using them for templates. Sure, it seems reasonable that if one exists so should the other, but perhaps the reasonable thing is that neither should exist. OK, it is too late to remove them from the language but perhaps we should keep a clear view that constructors are nothing like any flavour of ordinary function.

### namespace and static

If you mentally completed the title to this article you might wonder where the old ones are. Wonder no longer. The dual use of static in C was probably a little unwise but can be explained by saying that it means that the object whose name is being declared has current scope is to be placed in static storage. The name is placed in the smallest enclosing scope (block or file). Where static is used at block scope it means just about what static means as a computer science term, but at file scope its significance is to over-rule the automatic exporting of names to global program scope.

C++ came along and during the first ten years of its development those with a C mindset resisted new keywords like the plague. The result was that static acquired new uses with extra significance. It is time that this nightmare was cut back.

When `namespace` was introduced to solve a rather different problem (providing a mechanism for controlling the injection of names from libraries into global namespace) it seemed sensible to consider if it could also be used to limit the pollution of the global namespace with names that were intended to be limited to file scope. To this end the C++ Standards Committees did two things. The first of these was to give a meaning to code such as the following:

```
namespace {
    // various declarations
}
```

The meaning is that names in such code belong to a namespace with a unique and arbitrary (i.e. even if you discover what it is for one compilation, it may and probably will change next time round) name. The purpose of this is that you can never refer to the names in such a block outside the immediately enclosing region (file or outer namespace) because you do not know what the name is. Confused? Well let me take another cut at explaining. Consider:

in *file1.cpp*

```
namespace {
    int i;
    void fn(int i);
 // other code
}

namespace X {
    int j;
    void gn(int i);
}
```

in *file2.cpp*

```
extern int i;     // declare i as
defined elsewhere
namespace X;      // declare the name
X
using X::j;
```

Now the `i` declared in *file2.cpp* cannot be the `i` defined in the anonymous `namespace` in *file1.cpp* because the link name for that `i` is prefixed by a compiler generated scope name and so will not match the global `i` of *file2.cpp*. By contrast the contents of `namespace X` can be made available in *file2.cpp*.

The result of this is that you get all the functionality of C's global static but without the confusing keyword. Having provided the functionality it only remained for the Standards Committees to encourage you to use it by deprecating the use of static at file scope. That means that they have given notice that future versions of C++ (though not the standard currently under development) might not continue to support that C usage.

Unlike classes, `namespaces` are extendible so you can wrap each block of file scope declarations in the anonymous `namespace` and all such blocks within a single file will share the same unknown qualifier. Like the other innovations above, I suggest that you start moving to this one as soon as you have a compiler that supports `namespace`.

### The New Style Headers.

You will see increasing use of lines like:

```
#include <iostream>
#include <list>
```

The lack of *'.h'* is not a typo. These are the new C++ system headers that make particular parts of the Standard C++ Library available for use in the following code. When you use these you need to know that almost all the Standard C++ Library is contained in `namespace std`. If you do not understand this, you will not be able to use these headers. For example, suppose that you wanted to use `cout` in your program. You can do one of the following:

- refer to it as `std::cout`. Unfortunately that probably means that '`<<`' will not work because it will be provided by `std::operator<<()` functions. I am not sure about this and perhaps one of the resident experts could write about such issues.

  *(According to CD2 3.2.4 the operator resolution begins with member functions of the LHS and then moves on to free functions in the namespace(s) corresponding to the operands. With a conforming compiler "`std::cout <<`" should invoke `std::operator<<()`. - Alan)*

- import the relevant items with using declarations such as `using std::cout`. Again I would welcome someone explaining how associated operators (and possibly functions) work in such circumstances.

- use the quick, magical fix, of the using directive: `using namespace std;`

By the way, do not get bitten in the way that Francis did by assuming that `cout` can be used in the dynamic initialisation of global variables (and in their destructors). The classic implementation of *iostreams* ensured that `cout` existed from the first to the last linked file that included *iostream.h*. This was done with an ugly hack. The C++ Standard will lay no such burden on implementors. The result is that you should not use *iostream* objects in constructors/destructors of global objects.

Moral: Do not use global data objects. If you need them use a global function instead. That way you can control the problem of order of initialisation of multi-file programs (if you include even a single library header, you are definitely into that region.

Get used to the new style headers because you will need them if you are going to use the full power of the Standard C++ Library. Not doing that would be silly. A lot of competent experts have worked long and hard to provide you with tools to make your life easier don't waste their efforts.

## Conclusion

If you want any of the above expanded, re-addressed etc. please email Francis and he will pass the request on to me. I do try to keep things simple but detail seems to run away with me.

# C++ Techniques

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

## OOD again: Some light, some shadows - by Graham Jones

First of all, many thanks to Kevlin (Overload 16) for replying in detail to my article in the previous issue. He certainly shed more light than darkness in my direction, though perhaps not always in the way he intended: one thing I now understand better is how the kind of programming I do is atypical.

I stand by my statement that "using OOA/OOD for the interface is fine, but a minor issue either way". Only about a quarter of the code in my application is do with 'presentation issues', and it was the easiest quarter to design, code and test. *(I'd say you have rather more "presentation code" than is typical The normal range is around 10%-30% - Alan)* No doubt this explains why most people find the examples in 'Design Patterns' to be balanced, whilst I do not. I should add that I like 'Design Patterns' in many ways. In particular, it focuses on fairly low level design, where it merges into implementation, and I am a good deal happier with OOD (or OOP) at this level than at the higher level tackled by Booch. Kevlin's comment that 'fine grained classes can have more of an impact on the construction of your application than the coarse ones' was timely. I was beginning to suspect that this was the case, and it helped crystallise some of my ideas.

### Neural nets

I was intrigued by Kevlin's description of the neural net program. Look at the order in which he approached the problem of modelling the net. A mathematical model, the choice of a data structure (matrix) and associated algorithms, then the neural net class, then its interface, then factoring out the UI. It is roughly the order in which I have tackled similar projects. It is also almost exactly the reverse order that the OOD paradigm (without modularity) recommends. *(I realise that authorities differ, but I'm not aware of any that would reject Kevlin's approach. There is always a "cut-off" where modelling further detail as object classes is counter-productive. Kevlin used his expertise to identify this point. - Alan)*

I'm afraid this example reinforces my view that OOD is of little help for this kind of program - and it is this kind of program I spend most of my time on. Kevlin quite rightly says that my views on design are incomplete. But unless there is more of a method to this modelarity business than Kevlin describes, it seems incomplete too. Does modelarity add flexibility to OOD? Or does it allow the designer so much freedom that there is no real guide except experience?

Much work in AI suffers from the 'solution first' approach. The most important design decision in Kevlin's example was to use a neural net in the first place, but this is just a 'given' in his description. Likewise, Booch decides pretty quickly that a 'blackboard' system is appropriate for his problem, and gets on with implementing it (Chapter 11 in "Object Oriented Design"). I am not criticising Kevlin, his friend, or Grady Booch, since AI is not their job, or even their main interest when designing these systems. However, if your job is to actually solve a problem you must be prepared to make radical changes in the methods used, and your program design should take that into account. When you discover that the neural net does not do what it is supposed to do, what is left of the design? Do you perhaps wish that you'd spent less time on presentation issues? *(No, I wish I'd identified and fixed the interfaces between the parts of the system that need to be flexible and those that I expect to be stable. Oddly enough OOD is an effective tool for doing this. - Alan)*

### Is interface or implementation most volatile?

I'd like to briefly describe an early stage of my OCR program. The input is a bitmap, one bit per pixel, and sizes up to 1Mb or 8 million pixels are common. This, surely, is an obvious object in my program. I want to find all the shapes (connected components) in the image, and there are various algorithms for this. I will sketch a few: the important point for my purposes is the way they need to access the bitmap.

1) Find a black pixel, and then any black neighbours, and then any of their neighbours etc. Pixel co-ordinates are stored while this searching is going on, and to avoid visiting the same pixel many times, the pixels are set to white in the image as they are found. `getpixel()` and `setpixel()` member functions seem sensible: they can wrap up the bit manipulations required to find the colour of a pixel at given co-ordinates. This is the simplest, and the slowest method.

2) Scan down the image line by line, find runs of black pixels in each, and build up a graph structure in which the vertices are horizontal black runs, and the edges connect runs that touch one another in adjacent lines. The graph can then be converted into a list of shapes. The best way for this to access the image seemed to be via a `getscanline()` function. This algorithm is potentially a lot faster than (1) but using `getpixel()` would nullify this benefit (`setpixel()` is not needed.) `getscanline()` can convert the scan line into a `char` array, and the algorithm can be written as though it was dealing with a 2D array of chars.

3) Like (1) but using horizontal black runs instead of individual pixels. `getscanline()` is no good because while finding a tall shape, many scan lines would be needed. Again `getpixel()` and `setpixel()` are too slow. A third way of accessing the image is needed.

I do not know which is the best algorithm without trying them. This is very often the case in my work: I know the input and required output, but not how to get there. The point I want to make about OOD is that it is aimed at cases where the way in which data is to be accessed remains constant, while the way in which the data is represented inside the class may change. *(I disagree, it is aimed at cases where interactions between parts of the system remain constant, but the algorithms and data structures used in each part may vary. In particular the above examples sound like three implementations of the same class - Alan)* As in the example above, the opposite keeps happening to me: the way in which the data is represented stays constant, while the interface changes. *(Then your object model is wrong. This is the problem Kevlin's friend encountered: getting the correct model - Alan)* A couple of sentences from Barton and Nackman's Scientific and Engineering C++ (p230) strike me as relevant:

"Information hiding allows us to exploit the possibility of having different state representations by picking the "best" representation according to criteria prevailing when we implement the object's behaviour. If these conditions change, as when problems or computers or compilers or users change, the representation can be altered and the behaviours reimplemented without altering other parts of the program that only call member functions."

During the 4 years that I have been developing my OCR application, the problem and the computer and the compiler and the users have stayed the same (from a design point of view) while the program has changed often and radically. I can only say that in my experience, functions and data structures survive these changes better than objects. This applies to relatively low-level changes like different algorithms for shape-finding and to more drastic changes like replacing your neural net with a blackboard. Perhaps I should make it clear that I am talking about functions as a design element - function-oriented design if you like - not about the capabilities of functions versus classes in C++.

*For a moment try thinking of objects a different way: take all the "functions" that act on a "data structure", put the data structure as a private structure inside a "class" and make the functions into member functions of that class. Any functions that are not invoked from outside the class should be "private", the ones invoked from outside are "public". By your own admission the functions and data structures are largely stable - apart from "low level" algorithm changes - which I'd expect to be in the "private" functions that do not impact the rest of the system. A class built this way should be more stable than the functions and data structures that comprise it.*

*In one sense this approach to constructing object classes is a "long way around" since it goes via your existing functional decomposition, but it allows you to make effective use of your existing expertise. (Someone is sure to point out that it also loses some of the potential benefits of OOD - but those can only be realised with a good deal of experience of the method.) Alan*

## Polar Complexities

In my last article I described a class for complex numbers and quoted Russell Winder to the effect that he thought that knowledge of whether the internal representation was Cartesian or polar should be hidden from the user of the class. In my opinion, that is a very bad design decision which only someone who was fixated on objects would even consider. Changing from Cartesian to polar co-ordinates will affect the speed and precision of practically every operation on the class. The idea that one could make such a change without affecting client code is ridiculous. I think that in this case, the nature of the internal representation (not the actual variables) should be exposed to clients. I note with relief that Barton and Nackman's classes for arrays expose their internal representation in their names at least: you know where you are with a ConcreteFortranSymmetricPackedArray2d<T>.

## Encapsulation

In my last article I also mentioned a program for converting between different image file formats. This is another case where the data representation should not (or cannot) be hidden *(Why not? - See* "Observations on the Design of an Address Class - By Mark Radford" *in this issue - Alan)*. Possibly I am beginning to make sense of what it is I don't like about OOD. Encapsulation is a wonderful thing but an 'object mentality' leads programmers to encapsulate the wrong thing in some situations. In particular objects hide the way that data is represented, and there are at least three reasons why this might be a bad thing: because the representation is more stable as the program develops than the way it is accessed; because the user needs to know about the time, memory, or precision implications of the representation; because the way the data is represented is already 'fixed' as some sort of external standard. *(I don't see why encapsulation is wrong in any of these circumstances - which I agree are common. Alan)*

## Final thought

In "Safer C" by Les Hatton, 1994, ISBN 0-07-707640-0, p98, the author says that he has seen "a number of C++ systems with ridiculously simple functional components and a labyrinthine class system which caused the mind to boggle. When plotting the class hierarchy of such a system recently from a major communications package, the lines joining the various classes were in such abundance that all the white pixels between the boxes representing the class names disappeared after about twenty minutes plotting on a blisteringly fast workstation, prompting the package's designers to burst out laughing, and confirming their view that things had got a little out of hand!"

I think Phil Bass (Letters, Overload 16) is right to fear a backlash against C++ and OO. The computer industry seems to have a sheep-like tendency to rush from one extreme to another. I did not intend in my last article, and I do not in this one, to put people off trying OOD - I know that my experience is limited, and specialised. What I would like this article to do is to provoke others to write in with their experiences.

*Graham Jones*

*Looking at the above I realised that I've added more than my intended proportion of annotations. I believe that OOD is a solution to the problems that Graham has identified rather than the source. I hope that the above comments are illuminating. If not, please write! - Alan*

# A model for backsolving by Richard Percy

*problem statement ◆ a generalised approach ◆ backsolving toolkit*

## Introduction

In the last article I presented a *Cashflow* template class that provides a generalised method for generating those projections that financial analysts like so much. I don't propose to take this model any further forward at the moment but just backward.

It is a fact of life in financial matters that we often know where we want to get and how we want to get there but we don't know where to start. The result of this, as far as financial applications programming is concerned, is that the method of calculating an end result from a starting position is often straightforward but only the desired end result is known.

For example, a company may have a known liability on, say, 1 January 2000 and wishes to invest in a stock whose dividends and redemption proceeds will cover this cash outflow. It would be fairly easy for the company to work out the eventual proceeds from an investment of a known amount now in a fixed interest stock, assuming that it knows the current stock prices and interest rates available on cash deposits. It is less easy to calculate the investment necessary to meet the liability and even less to compare the yields of two alternative investments.

Sometimes a mathematical formula can be derived to calculate this kind of problem but this may not be portable to other similar problems and can be time-consuming. The best approach is often a "trial and error" method, which normally involves sensible initial guesses of the answer that are refined successively until we are "close enough". The model I am presenting below is a generalisation of this method in the form of a tookit of classes that are bolted together according to the programmer's needs.

## A generalised approach to backsolving

The biggest problem with trying to generalise backsolving is the huge variety of data types and methods that programmers may want to use. Having said that, most of the time they will just want to use built-in types (e.g. double) and a bog-standard targeting method, such as interval bisection. So, it would be

useful to provide a range of standard components that can be used straight out of the box or replaced with something else provided by the programmer.

Because of these requirements the model I have developed is not fully object-oriented but rather splits the problem into elements that I think programmers will want to mix and match. These are:

- ***Seeker*** - Responsible for generating new guesses for the trial and error method. Also receives initial guesses supplied by the programmer and results of each trial.
- ***Tolerance tester*** - Decides whether a given guess and its result are close enough to the target value.
- ***Generator*** - Calculates the result of each guess.
- ***Solver*** - Controls the interaction of the above elements by generating successive guesses and finding their results until they are close enough to the target.

The ***Seeker*** provides the algorithm used to generate successive guesses and will normally need to keep a history of previous guesses. Therefore, it is often the most complicated part of the model. I intend that my library will provide one or two generalised seekers that will work for most purposes but the programmer can provide his own if he wishes.

The ***Tolerance tester*** should normally be very simple and, again, the library should provide some common general methods.

The ***Generator*** can be just about any "black box" that takes a source value and generates a result. Consequently, it is up to the programmer to supply it but the library will impose restrictions on its general form.

The ***Solver*** is simply a controlling loop and will be provided by the library.

## Implementing the elements of the toolkit

### Introduction

I have developed the following implementation using Borland C++ 4.02 but have generally tried to avoid compiler-specific code. The model makes heavy use of templates and uses the "function object" idiom in a manner similar to STL.

An important concept is that of ***Source*** and ***Result*** types because most of the templates are based on them. The whole point of this exercise is that the programmer provides a "black box" that converts a *Source* (any type) into a *Result* (any type) but only knows the value of *Result.* He wishes to calculate the *Source* value without tortuous rearrangement of a mathematical formula. The code sometimes abbreviates *Source* and *Result* as *S* and *R*.

The library is organised into two main modules: `UTARG` and `SAMPFUNC`. The first contains general definitions, the *Seeker* abstract base class and the *Solver*. The second contains examples of each of *Seeker*, *Tolerance tester* and *Generator*. Some of these would be provided by the library in practice, whereas others are purely for illustration and would normally be in a further module provided by the library user.

It is useful to define a special type of exception for backsolving and this is provided in `UTARG` by deriving from an existing exception class, *xmsg*.

```
// Borland string class
#include <cstring.h>
// Borland exception classes
#include <except.h>

class XNotConvergent: public xmsg {
public:
    XNotConvergent(string& s) :
        xmsg(s) {}
};  // XNotConvergent
```

### Seeker

As stated above, the *Seeker* is the trickiest part of the model to program because it will normally need to keep a history of guesses (of type *Source*) and their results (of type *Result*). Because of the order of processing required by the *Solver* (see below) the timing of the data transfer to and from the *Seeker* is quite rigidly specified and separate functions are required for data passing in and out. This sounds like a job for an abstract base class!

```
template <class S, class R>
class Seeker {
public:
virtual ~Seeker() {}
virtual const S& GetNextGuess() = 0;
virtual void SetLastResult(const R&)
                                  = 0;
};
```

The user's *Seeker* must be derived from the **Seeker** base class and must implement the functions `GetNextGuess` and `SetLastResult` for an object to be created.

I have provided an example class, **MyBisector**, which uses a bisection method and works on continuous monotonic functions (i.e. *Result* values always either increase or decrease with increasing *Source* values and there are no gaps). The class requires *Source* and *Result* to have some basic arithmetic operators defined (<. + and /) and is most suitable for floating point types. The constructor requires the target *Result*, two initial guesses and their results to get it started.

```
template <class S, class R>
class MyBisector: public Seeker<S, R> {
public:
    MyBisector(const R& target, const S& guess1, const R& result1,
                                const S& guess2, const R& result2)
    : gu1(guess1), gu2(guess2), targ(target), res1(result1),
      res2(result2), expRes(false) {
        if ((res1<targ && res2<targ) || (targ<res1 && targ<res2))
            throw XNotConvergent(string(
               "MyBisector:Invalid result arguments in constructor"));
    }
    // default copy & assign are OK
    virtual const S& GetNextGuess();
    virtual void SetLastResult(const R&);

private:
    S gu1, gu2, gBis;  // 2 old guesses & newest guess
    const R targ;
    R res1, res2;// 2 old guess results
    bool expRes; // true if expecting result of last guess
};  // MyBisector

template <class S, class R>
const S& MyBisector<S,R>::GetNextGuess() {
    if (expRes)
        throw msg(string(
            "MyBisector:Asking for next guess when result expected"));
    expRes = true;
    return gBis = (gu1 + gu2) / 2;
}

template <class S, class R>
void MyBisector<S, R>::
SetLastResult(const R& lr) {
    if ((res1<lr && res2<lr) || (lr<res1 && lr<res2))
       throw XNotConvergent(string(
         "MyBisector:Result outside previous results"));

    if ((lr<targ && targ<res1) || (res1<targ && targ<lr)) {
        res2 = lr; gu2 = gBis;
    }
    else {
      res1 = lr; gu1 = gBis;
    }
    expRes = false;
}
```

I'm sure that the reader can think of more widely applicable and efficient algorithms than this. That's the whole point of designing the library this way.

### Tolerance tester

The *Tolerance tester*'s job is to decide when to stop; so it needs to know the target *Result* and the criteria for a guess being close enough. This implies a function call that returns a true/false value for each guess. However, it would be nice only to have to supply the target value once at the start and a user-supplied routine may also want to keep a history of guesses. A function object design with an `operator()` returning `bool` is the best choice here. The function takes a *Source* and *Result* argument because a decision might be based on either

or both for a given guess. For example, we might want to stop when we have guessed the same value more than 5 times or if the result of the guess is within a certain margin of the target.

The example shown below illustrates the latter and requires some arithmetic operators defined for *Source* and *Result*.

```
template <class Source, class Result>
class MyTol {
public:
    MyTol(const Result& target, const double margin) :
        targ(target), marg(margin) {}
    // default copy & assign are OK
    bool operator()(const Source&, const Result& r) const {
        // Source isn't used
        if ((r<targ && targ-r<targ*marg) ||(targ<r && r-targ<targ*marg))
            return true;
        else
            return false;
    }
private:
    const Result targ;
    const double marg;
};
```

**Generator**

The *Generator* simply calculates a *Result* value from a *Source* value and, therefore, can be just about any function that takes a *Source* argument and returns a *Result*. As with the *Tolerance tester*, we are going to make repeated calls to the function and will probably want to initialise with some parameters that don't change with each call. Therefore, a function object design is specified.

I have provided two examples: a quadratic equation and the internal rate of return (IRR) of a simple financial project. In fact, both problems can be solved by simpler means but the simple examples are provided for clarity.

```
template <class T>
class Quadratic {
public:
    Quadratic(const T& a, const T& b, const T& c) : a_(a), b_(b), c_(c){}
    // default copy & assign are OK
    T operator() (const T& x) const  {return a_*x*x + b_*x + c_;}
private:
    const T a_, b_, c_;
};
```

The second example makes use of the **Cashflow** class that I developed in the last article. It calculates the net present value (the *Result*) of a series of regular payments at a given interest rate (the *Source*). Actuarial students might recognise this as the present value of an annuity certain calculated in a generalised way.

```
#include "cashflow.h"
#include <math.h>

class RegularCash {
public:
    RegularCash(double amount, unsigned int interval, unsigned long duration)
       :  amt(amount), inter(interval), dur(duration) {}
    // default copy & assign are OK
    double CalcNPV(const float rate) const;
    double operator() (const float rate) const {return CalcNPV(rate);}
private:
    class RcVec;
    double amt;           // regular amount of each payment
    unsigned long inter;  // no. of periods between each payment
    unsigned long dur;    // overall no. of periods in payment stream
};//class RegularCash
```

```
class RegularCash::RcVec {
public:
    RcVec(const double amount=0) :
        amt(amount) {}
    // default copy & assign are OK
    const double GetAmount() const
        {return amt;}
    bool RollForward(
        unsigned long newDuration,
        RcVec& oldRow) {
        // the same amount is paid
        // at regular intervals
        // in perpetuity
       amt = oldRow.amt;
       return true;
    }

    bool IsEqual(
      const RcVec& other) const
         {return amt == other.amt;}

    ostream& PrintOn(
        ostream& o=cout) const
          {return o << amt << endl;}
private:
    double amt;
};//class RegularCash::RcVec

double RegularCash::CalcNPV(
  const float rate) const {
  double npv(0);
  RcVec startPayment(amt);
  // first payment is at time 1
  Cashflow<RcVec> cf(1);
  cf.RollUpLim(startPayment,
  RcVec::RollForward, dur, true);
  for(signed long c(cf.BaseIndex());
     c<=cf.LastIndex(); c++) {
      npv += (cf[c].GetAmount()) /
             pow(1+rate, c * inter);
  }
  return npv;
}
```

**Solver**

The final part of the backsolving jigsaw is the *Solver*. This simply controls the process by performing the following steps.

1. Get a new guess from the *Seeker*.
2. Calculate the result of the guess using the *Generator*.
3. Test the values with the *Tolerance tester*. If close enough then stop.
4. Otherwise, tell the *Seeker* the result of the last guess and go again from the top.

The *Solver* is implemented as a template function and is surprisingly simple.

```
template<class Source, class Result,
   class Generator, class Tolerance>
const Source BackSolve(
  Generator gen,
  Seeker<Source,Result>& sk,
  Tolerance tol) {
  Result res;
  Source solution;
  while(true) {
    solution = sk.GetNextGuess();
    res = gen(solution);
    if (tol(solution,res)) break;
    sk.SetLastResult(res);
  }
  return solution;
}
```

Note that, although *Source* and *Result* are template parameters, they aren't supplied directly as arguments to the function. Instead, they are passed through as parameters to the **Seeker** argument, which lets the compiler find the instantiation of the template. This

handy feature eliminates the need to supply nasty *Source* and *Result* dummy arguments.

I suppose that calling the *Generator* is wasteful if the *Tolerance tester* only needs the *Source* value, but it can't be helped.

### Controlling the application

To perform the backsolving all the user needs to do is create the necessary objects and pass them to the *Solver*. The first example below solves a quadratic equation ($10x^2$-16x+5.2=400) with real roots and the second calculates the internal rate of return of a project with a down payment of £10,000 and proceeds of £1,000 p.a. for 25 years.

The implementation is for Borland's DOS or EasyWin platform.

The program calculates the positive root of the quadratic equation (7.13404) in 33 guesses and the interest rate (8.78042%) in 27 guesses. The results can be checked for the quadratic equation with the formula we all learned at school and for the annuity certain by using NPV=$[1-(1+i)^{-25}]/i$, where $i$=0.0878042.

In the next article I plan to look at generalised formatting and numerical precision issues for financial people. In the final article in this series I hope to cover updating the models I have developed to a more recent release of the C++ Standard Library.

*Richard Percy*

```
int main() {
  int retCode;  // {You ought to initialise this - Alan}
  try {
    // Example 1: quadratic equation
    const double qTarget(400);
    Quadratic<double> qGen(10,-16,5.2);
    MyBisector<float,double> qSeek(qTarget,0,qGen(0),999999.0,qGen(999999.0));
    MyTol<float, double> qTol(qTarget, 0.00001);
    cout << "Quadratic solution: " << BackSolve(qGen, qSeek, qTol) << endl;

    // Example 2: internal rate of return using cashflow class
    const double npvTarget(10000.0);
    RegularCash npvGen(1000.0, 1,25-1);
    MyBisector<float, double>
        irrSeek(npvTarget, 0,npvGen(0), 999.99,npvGen(999.99));
    MyTol<float, double> irrTol(npvTarget, 0.00001);
    cout << "Internal rate of return solution: " <<
       BackSolve(npvGen, irrSeek, irrTol) << endl;
  }
  catch (XNotConvergent x) {
      cout << "\nException: Non-convergence!\n\n" << x.why() << endl;
      retCode = 32767;
  }
  catch (xmsg x) {
      cout << "\nException!\n\n" << x.why() << endl;
      retCode = 32767;
  }
  catch (...) {
     cout << "\nException!\n\nProgram threw an unhandled exception" << endl;
     retCode = 32767;
  }
  return retCode;
}
```

# Corrections - Overload 17/18

*Adrian Fagg*
*adrian@rbaf.demon.co.uk*

## Using Objects for Background Tasks

Apologies to anyone reading my article. The final code box contains `<Test>`, which obviously should be `<MainWindow>`

Did I write that?

Yes, apparently.

## "auto_ptr || !auto_ptr"

I too (just like Adrian) feel I should mention a slight slip-up in my article. The default constructor for `no_copy` should really be protected and not public.

*Jonathan Jagger*
*jonj@dmv.co.uk*

# editor << letters;

## Further Thoughts on Inheritance for Reuse

*From Francis Glassborow*

In the last issue I only skimmed over this use of inheritance in C++. Sean's comment together with several emails makes me think that it would be beneficial to revisit the subject in a little more detail.

To understand what is going on you need a firm grasp of dynamic versus static binding. I know that some programmers get very confused by those terms. In the simplest form static behaviour is that which can be fully determined by the compiler whilst dynamic behaviour is somehow determined at execution time.

Objects have a static type. This means that an object has a well-defined existence at compile time. On the other hand objects that are handled indirectly via pointers or references have two types. The static type provided by the declaration of the pointer or reference identifier and the dynamic type of the object that they are referring to. Keep that in mind.

In inheritance hierarchies we talk of a function over-riding a base class version. By this we mean that there is a new definition of a base class function in a derived class. We also have the possibility that a derived class function hides a base class one. To try to make this clear consider the following very simple hierarchy:

```
class Base {
public:
    void fn (int);
    void fn (double);
    void gn (int);
};

class Derived : public Base {
public:
    void fn (int);
};
```

And some code to use that hierarchy:

```
int main() {
    Base b;
    Derived d;

    // calls fn(int) for Base
    b.fn(1);

    // calls fn(double) for Base
    b.fn(1.0);

    // calls fn(int) for Derived
    d.fn(1);

    // calls fn(int) for derived!
    d.fn(1.0);

    // calls fn(double) for Base
    static_cast<Base>(d).fn(1.0);
}
```

We say that `void fn(int)` in `Derived` over-rides `void fn(int)` in `Base` and hides `void fn(double)`. This is all to do with the way in which names are looked up. Nothing new in any of that and I only include it to remind you of the rules. In C++ behaviour of objects is statically determined based on the declared type of the object.

*Francis Glassborow*
*francis@robinton.demon.co.uk*

## Inheritance

*From Roger Lever*

In Overload17/18 "The uses and abuses of inheritance" by Francis Glassborow raises some interesting points. However, in the best traditions of these things I do have some points to make...

The coding rule: "If you want a variant of an existing type that has different behaviour then use either private inheritance or layering. Do not use public inheritance". This appears to be ambiguous in terms of what is actually meant by different behaviour. This is important as it is this distinction which will decide whether public inheritance is an appropriate choice or not.

Let us consider the interface specification of class Miss is:

```
// A is implemented by a mechanism X
    float do_something();
```

This is later changed to have everything the same except:

```
// B is implemented by a mechanism X
    double do_something();
```

or:

```
// C is implemented by a mechanism Y
    float do_something();
```

or:

```
// D is implemented by a mechanism Y
    double do_something();
```

This allows us to consider the issue as:

1. Interface specification has changed as in B and D

2. Implementation behaviour has changed as in C

3. Interface and implementation has changed as in D

### Different behaviour = interface specification

Comparing A and B, the behaviour is the same X in both cases, however, the interface specification has changed. This could well lead to either private inheritance or composition as a means of maintaining backward compatibility. The bottom line is that interface changes are very difficult to deal with, without either breaking existing code or forcing some kludge to provide the changes. This is a case where public inheritance is not a good idea, for the reasons stated in the article.

### Different behaviour = implementation mechanism

What if we wanted to implement A using a mechanism Y as in C? This is an implementation detail and as long as the interface specification remains fixed it has no bearing on the issue. In fact this is part of the reason for hiding these details - that they can be changed without affecting others dependent on the interface. So public inheritance does not enter the equation.

### Different behaviour = interface and implementation

What if we wanted to have `do_something` with both X and Y mechanisms, dependent on some condition, as in the Address example in the same article? As is pointed out there - virtual functions are a good solution, not overriding non-virtual functions. So public inheritance enters the equation here, probably via an ABC.

Comparing A to D, this may be a case where a new concrete class is required and as such public inheritance may well be a reasonable option. This is because D is exhibiting different behaviour to A. At this point usual public inheritance rules such as "Substitutability" come in...

## Summary

Therefore the coding rule: "If you want a variant of an existing type that has different behaviour then use either private inheritance or layering. Do not use public inheritance", should be modified to "If you wish to change only the interface specification of an existing type then use either private inheritance or layering".

*Roger Lever*
*rnl16616@ggr.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 Edito
r for inclusion here.

## The UML/OMT User Group

Following the release of version 1.0 of the Unified Modeling Language in January, the UK OMT User Group is changing its name and scope to more fully embrace the UML, to become the UML/OMT User Group.

Developed by leading methodologists at Rational in collaboration with other industrial partners, the UML is a notation set with semantics for representing and specifying software systems. It has evolved from a number of object-oriented development methods, notably Rumbaugh's OMT, Booch, and Jacobson's Object-Oriented Software Engineering.

The user group has been tracking the development of the UML from its initial development to its submission to the OMG as a standard.

Individual membership of the group is ?50 per annum. Corporate membership is ?200 with five named members, or ?300 with ten named members. All prices are exclusive of VAT. A web site is planned and corporate members will soon be able to take up the offer of a free link to their own site.

All members receive a quarterly newsletter, reductions on OO books from leading publishers, and a case book of articles on the techniques and notation of OMT, OMT-2 and UML. Each year the group also holds a seminar day, which is open to both non-members and members (at a reduced rate).

For further information about the user group and membership please contact either Jan Bevans (jbevans@qatraining.com) or Kevlin Henney (khenney@qatraining.com), on 01285 655 888 at QA Training Ltd, Cecily Hill Castle, Cirencester, Gloucestershire, GL7 2EF.

Credits

Founding Editor

*Mike Toms*
*miketoms@calladin.demon.co.uk*

Acting Editor - for this issue only

*Alan Griffiths*
*CCN Group Limited, Talbot House,*
*Talbot Street, Nottingham, NG1 5HF*
*overload@octopull.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

*David Hodge*
*2 Clevedon Road*
*Bexhill-on-Sea*
*East Sussex TN39 4EL*
*101633.1100@compuserve.com*



Copy deadline

All articles intended for inclusion in *Overload 20* (June/July) should be submitted to the editor by May 26th.

# ACCU and the 'net

## ACCU.general

This is an open mailing list for the discussion of C and C++ related issues. It features an unusually high standard of discussion and several of our regular columnists contribute. The highlights are serialised in *CVu*. To subscribe, send any message to:
accu.general-sub@monosys.com
You will receive a welcome message with instructions on how to use the list. The list address is:
accu.general@monosys.com

## Demon FTP site

The contents of *CVu* disks, and hence the code from *Overload* articles, eventually ends up on Demon's main FTP site:
ftp://ftp.demon.co.uk/accu
Files are organised by *CVu* issue.

## ACCU web page

At the moment there are still some problems with the generic URL but you should be able to access the current pages at:
http://bach.cis.temple.edu/accu
Please note that a UK-based web site will be operational in the near future and this will become the "official" ACCU web site. Alex Yuriev has done a great job supporting the ACCU web site from the US – thanks Alex!

## C++ – The UK information site

This site is maintained by Steve Rumsby, long-serving member of the UK delegation to WG21 and nearly always head of delegation.
http://www.maths.warwick.ac.uk/c++

## C++ – Beyond the ARM

Sean says he will have updated his pages by the time this is in print.
http://www.ocsltd.com/c++
Any comments on these pages are welcome!

## Contacting the ACCU committee

Individual committee members can be contacted at the addresses given above. In addition, the following generic email addresses exist:
caugers@accu.org
chair@accu.org
cvu@accu.org
info@accu.org
info.deutschland@accu.org
membership@accu.org
overload@accu.org
publicity@accu.org
secretary@accu.org
standards@accu.org
treasurer@accu.org
webmaster@accu.org
There are actually a few others but I think you'll find the list above fairly exhaustive!