ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group
Issue 20
June / July 1997

Editorial:
John Merrells
4 Park Mount
Harpenden
Herts
AL5 3RA
John.merrells@octel.com

m

Subscriptions:
David Hodge
2 Clevedon Road
Bexhill-on-Sea
East Sussex
TN39 4EL

101633.1100@compuserve.co

£3.50

Contents

Editorial

Hi folks. I've been appointed editor for the next three issues. If this role suits me, and more importantly suits you, then I hope I'll be here for some time.

Traditionally, our Editor has been one of our principle contributors. Inevitably, this changes the editing job from a responsible constructive hobby into a part-time chore. My stint as Editor may tease out some authoring tendencies in me, but I don't intend to let us to slip back into that old state.

Editorial Board

To spread the workload Overload is now 'managed' by an editorial team, consisting of an Editor and three Readers. Our Readers have quite varied technical backgrounds and interests. Their role is to review all submissions for technical accuracy, and correct use of English.

Ray Hall – Did an Artificial Intelligence degree at Imperial College with a thesis on 'OO Development'. He has experience in writing and running a magazine, and will be concentrating on OO development and techniques.

Ian Bruntlett – Has been mostly working with C for the past few years and is now brushing up his C++ and OO skills.

Einar Nilsen-Nygaard – An Electronic Engineer by degree. Currently working on network management software. Has a few years of C++ experience and is currently moving into distributed OO technology.

Me? I've been programming in C++ for four years working on games, terminal emulation, and voice mail servers.

Contributions

There are approximately 500 Overload subscribers, and yet only half a dozen are regular contributors. We need to encourage more of you to make an effort. It's rewarding to share your ideas with your peers, and you only really understand something if you can effectively communicate it to somebody else. Have you noticed that when you approach a colleague to explain an evil problem how you work out the answer before they've said anything. Your spiel of frustration is interrupted by the solution. That's because you haven't truly understood the problem until you needed to communicate it. So, communicate and learn.

Too Much C++

A problem I perceive, but with which many others contend, is that there's too much C++ in Overload. I expect you may be sucking air through your teeth at this point. I believe that Overload should be broadened to include more general OO articles. There are two reasons for this:

1. There is not enough hard core technical C++ material being generated.

2. The audience isn't C++ technicians but OO programmers who happen to be crafting their work in C++.

I'd rather have a magazine that promoted pragmatic ideas about problem solving in an OO fashion than one that concentrated on the squiggles of the C++ Standard. I find advanced articles on the dim and dark corners of C++ to be of limited use. It merely points out to me that the brightest minds find something hard to understand. So, the feature is unlikely to be implemented, or implemented correctly. My code is unlikely to be portable, and I might

not fully understand all the subtleties of the problem. During the past four years I've been bitten by various compilers over multiple inheritance, templates, and exceptions. These articles are like anti-idioms or anti-patterns. 'Ah, best avoid X, Y and Z for a couple of years whilst it matures.'

I'd like to encourage more articles on topics which address applying object oriented design and C++ implementation to common problems. Basically, pattern designs, and pattern implementations.

As ever, this magazine is for the membership, by the membership. We need your contributions and feedback.

Want to contribute? Can't find a topic? – Mail me.
Need a solution? Explain the problem! – Mail me.

John Merrells
john.merrells@octel.com

Software Development in C++

Whence Objects?
by Ray Hall

It seems inevitable that a frequently discussed topic such as object orientation will be misunderstood by some of the people some of the time, and discussed in buzzword-compliant terms rather than with understanding. A sequence of articles in issues 15, 16 and 19 illustrates this well. Given that the misunderstandings in the original were addressed by Kevlin in issue 16, it seems appropriate to ignore their repetition in issue 19, except to take up one point: "*The computer industry seems to have a sheep-like tendency to rush from one extreme to the other*". So, just in case anyone else does not recognise that object-oriented techniques have been around for 25 years or so, here is a brief sketch of what happened and of what it was that caught the attention of many programmers.

Functional decomposition

Presumably, program designers have always felt that the text of their programs reflected "the real world" in some way. Nevertheless, our view of "the real world" is mediated by the ways in which we can describe it, and in the early mainframe era, barely 40 years ago, problems most often deemed suitable for computation were those whose solutions could be expressed in FORTRAN or in COBOL.

The former being expressed in mathematical notation, it is unremarkable that illustrative programs are in domains such as complex numbers, conic sections &c. The obvious way to solve such problems is to find one or more functions (sub-routines, procedures &c) which handle the data, so that at the bottom of the hierarchy, existing functions can be used, and functional decomposition continues to dominate design thinking. In part, this was reinforced by "top-down" methodology, which came with block structured languages (especially Pascal) in the 1970s.

Business problems such as accounting and invoicing derived their solutions from punch-card origins, and the data definition phase of a COBOL program looked like a set of punch-card layouts. The prevailing design thinking here derived from data-flow diagrams, though, as COBOL acquired block-structure characteristics, designers could think about functional decomposition here too. As an intriguing aside, Admiral Grace Hopper (one of the principal designers of COBOL) said, in an interview in Byte some years ago (just before she died), that they had assumed that when the library facility was implemented in COBOL, it would be followed up by the sale of standard libraries so that programmers, as such, would rarely be needed!

Non-procedural languages

Pretty obviously, functional decomposition, as a design methodology, assumes that implementation will be in a language which implements functions. So, what other languages are there then? Well, the group discussed so far (which includes C) are often described as procedural languages; non-procedural languages include Lisp, Prolog and (many) others. Lisp is the granddaddy of them, and belongs to the mainframe generation (late 50s) with FORTRAN and COBOL and is particularly important now as the development language in Autocad. Prolog comes from the C and Pascal generation of the early 70s.

These languages were designed initially to give more intuitive solutions to problems where functional decomposition seemed artificial, and the group as a whole includes many languages which are entirely experimental. The problem domains in

which they are important include logic and symbolic equation solving; understanding natural language; expert systems; and many areas of artificial intelligence.

Here is a brief illustration of the way in which Prolog specifies the solution to a problem, and leaves the procedure to the compiler. The problem is to determine if an object X is a member of a list Y, which can be expressed as *member(X,Y)*. A list can be partitioned into a head (the first element) and tail (the rest of the list) by the notation [H|T], and there is an "anonymous variable" for which the notation is '_'; the final notation is ':-' which can be read as 'if'.

A solution exists if X is at the head of the list, or if X is a member of the tail of the list, and the program consists of two lines:

```
member(X, [X|_]).
member(X,[_|Y]) :- member(X,Y).
```

Readers should at least recognise the elegance of this, even if it takes some time to feel that it is intuitive. In a procedural language the recursion implied by the second line has to be written out.

Where did OO come from?

Another set of heretical thoughts was being pursued around 1970 in the Xerox Palo Alto Research Centre. Indeed most of the important heresies and innovations in desktop computing came from there. Firstly the GUI and, with it, the recognition that a complete hierarchy controlled by a *main* function is not the best way of allowing for user interactions. The language *Smalltalk* evolved to cope with graphic elements and with event-driven situations (amongst others).

Smalltalk incorporated earlier ideas, but still has the power to astonish by the completeness and integrity of its conception. You may know that all components of the system are objects ("an object consists of some private memory and a set of operations") that numbers are examples of such objects, and that computation is conducted by passing messages. An object responds to any of the messages that make up its interface by carrying out one of its methods. Thus the message

```
3 + 4
```

looks much like an expression in a procedural language, but it is actually a message to the object '3' to carry out the method '+' with argument '4' and return the result. No big deal really, except that it is entirely congruent with messages such as

```
SubTotal sqrt
HouseHoldFinances cashOnHand
HouseHoldFinances totalSpentOn: 'food'
```

Commercially, Smalltalk has been insignificant, by contrast with its huge influence. Language designers who wished to try out object-oriented features generally grafted class/inheritance pre-processors onto existing languages. Gradually fully-fledged languages appeared, including one totally new one, Eiffel from the much-quoted Bertrand Meyer. OO versions of Pascal, Lisp and C became established.

Initially the leading C version was Objective C but, as we know, C++ has proved more popular (though Objective C now seems likely to become the development language in MacOS [Morgan 97]). Much of the rest is history and the concepts are now embedded in C++ and other languages, and C++ has all of the features needed to write fully object-oriented programs. But it also has other features which could ensure that an apparently object-oriented program is something quite different. Even in a totally OO language such as Smalltalk it is possible to write programs in a style appropriate to FORTRAN. [Kaehler & Patterson 1986]

Swings and roundabouts

Why then do the OO enthusiasts enthuse? What gets mentioned in such contexts includes robustness and ease of maintenance.

One of the earlier enthusiasts (in C++ terms) spoke of the "software IC" concept so that program design would be similar to designing electronic devices [Cox 1986]. The importance of such goals is impossible to overstate, and the extra effort required to attain them is generally thought to be a fair price to pay.

The next thing to note is what reviewers say about Delphi, Optima, Visual Basic 5 and C++ Builder, where rapid development is well to the fore. C++ Builder, and its Delphi2 ancestor, represent some of what can be done to make Objects easier. Nevertheless, Delphi seems attractive to many of its adherents for its ease-of-use, and Usenet contributions suggest that knowledge of OO techniques is thinner on the ground in Delphi land than amongst C++ users.

Ease-of-use in these systems comes from having available a good repertoire of classes. However, the demands made on the designer and programmer in an object-oriented system are substantial, especially for retrieving and understanding classes which are additional to the base classes provided with the compiler. Again, this is not a new problem: "*Smalltalk is an environment*"; "*Smalltalk is a big system*". [Goldberg & Robson 1989] In other words, it is the total development environment that counts and, as yet, this has not been addressed by available C++ systems.

If this was remedied then the problem of knowing whether an appropriate class exists already would exacerbate an already steep learning curve. Whatever your perception of the slope of the C++ object-oriented learning curve, it is a ***long*** slope. Even if not all C++ compilers are being used for OO programs, it is worth heeding the observation: "To derive significant benefit from C++ requires a modification in one's approach to problem solving" [Wiener & Pinson 1988], but that is another story…

Ray Hall
Ray@ashworth.demon.co.uk

Further reading

[Cox 1986] Object-oriented programming, an evolutionary approach: Addisson Wesley 1986 *{Similar sentiments in Byte 11,8 Aug 86 pp161-176}*

[Goldberg & Robson 1989] Smalltalk-80, the language: AddisonWesley 1989 *{Smalltalk-80 was the basis of the first commercial releases of Smalltalk. A companion volume describes the development environment; this volume, on the language, provides an illustration of the emergence of object-oriented thinking.}*

[Kaehler & Patterson 1986] A small taste of Smalltalk: Byte 11,8 August 1986 pp 145-159 *{Related to the same authors' book A taste of Smalltalk: Norton 1986}*

[Morgan 97] An introduction to Objective C: Byte (22,6) June 97

[Wiener & Pinson 1988] An introduction to object-oriented programming and C++: Addison Wesley 1988

The Draft International C++ Standard

The Casting Vote
by Sean A Corfield

The Java Study Group is meeting in London at the end of June for two days to continue discussions on how and what the standards process should cover for Java-related technologies. The progress since my last column is that Sun Microsystems Inc (SMI) have applied for PAS Submitter status (Publicly Available Specification). If accepted by ISO, this would allow Sun to submit the Java specification books for rubber-stamping, effectively. Whilst this is clearly a step forward, several individuals and organisations have expressed concern

about allowing a single, for-profit entity to become an approved standards originator. My next column will report on the outcome of this discussion but personally, whilst I want to see Java standardised pretty much as-is and as quickly as possible, I am wary of approving SMI in this role.

The joint ISO/ANSI C++ committees meet in London in July to resolve the comments arising from the ballot on the second Committee Draft document. One of the overriding impressions gleaned from the comments I've seen so far is that it's still very badly broken and we can only fix a small number of the more critical problems before schedule forces us to 'ship and be damned'... and we will. Writing exception-safe code is still horrendously difficult, using STL in any but the most basic ways is fraught with difficulty and various individuals are uncovering outright errors, contradictions and omissions in the draft every day. Even as secretary of ANSI X3J16 I find it hard to jump to the defence of either C++ or the committee and, like many others, just look forward to the day we stop having to work on the blasted thing!

I'm currently using the standard library in anger – ObjectSpace's implementation - and my frustration with compilers simply increases to the point where I wonder whether we will ever see validated products... Java just keeps looking more and more attractive!

Sean A Corfield, sean@ocsltd.com.

Painting the Bicycle Shed by George Wendle

Imagine that you are on a board of governors of a school (or any other organisation). The agenda of the meeting contains several tough problems to which there are no obvious answers. It also contains a number of items that require painful resolutions. Now imagine that there is a small item on the agenda about repainting the bicycle shed. You might think that such an item would be passed on the nod. More often than not this item will generate more heat than all the rest put together. Everyone will have their own opinion. What type of paint, what colour and who should do it may be fairly obvious. But someone will want to pull the shed down, someone else will want to prohibit bicycles. The debate will rage on and on and on.

The problem is that such an item is simple and easily solved so it makes an ideal ground for everyone to work out their frustrations caused by the painful and/or intractable problems that make up the rest of the agenda. Simple problems that just need an answer are the bane of any committee's life. Worse, they often distract effort from the important things.

A Variation: The Multi-solution Problem

There is another kind of problem, closely related to the bicycle shed (which only has one real answer - just authorise the money to get it painted) and that is a problem with three or more perfectly acceptable solutions (two in a committee that decides by consensus). Inevitably, each solution will have its proponents who too often will become emotionally attached to their choice. The outsider can see that the problem is that of making a choice between equally valid solutions.

When faced with a problem that has several solutions the first question should be 'does the choice matter?' If not take a vote, select the majority decision (or that with the largest number of votes) and close the issue for all time. Of course there is then the problem of coming across a superior solution that was missed from the original set. The sane answer is to accept that the original choice was arbitrary and mark the new, better solution as the primary choice next time.

Committees do not behave rationally. They will reject an outstanding solution because they do not like its proponent. They will accept a crazy solution or reject a sane one just because they do want to upset an authority figure.

What Has This to do with C++?

The problem with C++ is that a Committee is *designing* it. C wasn't. C was *standardised* by a committee, because almost every part of it had already been tried and tested.

Whilst I am hardly ecstatic about the core of the C++ language, I am happy that it is good enough for me, and many like me, to use profitably. I wish I could say the same about the Standard C++ Library. There are two major criteria for judging the quality of a standard library, usability and portability.

Look at the C Library. The novice C programmer can easily write programs using features from stdio.h, stdlib.h, math.h, etc. I know that each of these has hidden traps for the unwary, and thought is required before using them in industrial strength applications. But, the C Standard Library also contains many functions that can be used by skilled programmers to write highly portable code that is substantially robust.

Now look at the C++ Standard Library. Even things like I/O have been redesigned, not just once, but twice. I constantly hear the cry 'Cannot change that, it will break existing code.' This is foolish in the extreme, anyone who has tried to write code to conform to the draft standard library has had their code broken numerous times. Worse still, bugs and poor design bedevil the current library. I keep hearing the claim that there isn't time to fix the problems so we must just ship as is. That sounds more like a certain well-known software company than a responsible international standards body.

Now, I do not think that there is anything we can do to actually fix the current version of the Library. It is a mess, though many of the ideas are good, and it has had a largely beneficial effect on the design of the language. The concepts of the STL are fine, though programmers really do need to understand that STL is a low-level component library that should largely be encapsulated in higher level application components. In other words the ordinary application programmer should rarely use STL components though class designers should consider them as part of their basic tool kit.

I have seen Francis and others decry the quality of the MFC and continually drive home the message that the MFC is not an object-oriented library. In my opinion the draft Library is subject to exactly the same criticisms.

What we need is for the C++ standard to be shipped with the proviso that the next work item for WG21/NCITS J16 is to develop a new set of standard libraries that are entirely distinct from those shipped with the language. Fortunately one of the wisest decisions made about the Library was to encapsulate it in the `std` namespace. A future standard library will not conflict with the existing one.

Let me be absolutely clear about this. The C++ community needs a robust and well designed standard library. The one they will get with the language is not that.

What has this to do with the PTBSP?

The Standards Committees are locked in a World-view that requires them to try to fix little problems while being unable to tackle the big issues. For example, the concept and design of 'string' is fundamentally flawed. It is barely usable and I am willing to bet that no serious application programmer will want to go near instantiating the `basic_string` template with anything other than some

variety of char. Those that want to use other strings will also want better targeted designs. They will write their own, and that is exactly what we do not want.

Sticking with this example, the string concept should be implemented by a loose cluster of template classes. Each should be slim enough to be attractive, and have an intuitive interface. We need components that meet well defined needs. Not a component that can behave as a screw, a nail, glue etc. and made from any substance (value based type) that happens to be to hand.

If a C++ Standard is not shipped soon then C++ will die. If the current Library is shipped without any promise of something better, C++ will bleed to death before the next standard. You cannot fix fundamental design flaws in response to defect reports.

Those that want C++ to survive must grasp the nettle. They must admit that the current state of the Library leaves much to be desired. They must demand that they be given the chance to design a new independent standard library that can be shipped in sections. They must avoid ever again being overwhelmed by a requirement for a single monolithic standard.

The STL was the first (widely used) library that specified performance constraints. This was a major and courageous step forward. Now the time is right to recognise the need for standard component libraries that are delivered quite distinct from the underlying language standard. If the C++ community can persuade ISO to issue a work item to do this then C++ will live, if not it will die, strangled by the crowd of non-standard libraries.

The UK pushed for a normative addendum when voting for ISO C. Now it should render the international C++ community a service by pushing for standardised component libraries.

George Wendle

C++ Techniques

Make a date with C++: Typing Lessons by Kevlin Henney

Introduction

In *Overload* 19 I introduced some of the features that distinguish C++ from C. Many of them might be said to be significant improvements, i.e. "C++ as a better C" or "C++ as a safer C" are phrases often used to describe these changes. The stronger and slightly more logical type system is one feature that sticks out. I will continue this theme, as we look closer at features to support date representation.

struct your stuff

One possible way of representing a date type is to use a `struct` with day in month, month number and year (4 digit, of course) fields:

```
struct date
{
    int day, month, year;
};
```

On the face of it this seems no different to C. The difference comes in the use:

```
date dob = { 14, 3, 1879 }; // C++ not C
```

The use of the `struct` keyword to prefix the tag is redundant. It has always been a quirk of history that the tags inhabit a different namespace to type names – it

should be as simple as "a type is a type is a type", and C++ makes this logical simplification. For backward compatibility, however, we can still optionally use the `struct` keyword although clearly the principal reason for doing so has been removed:

```
/* C or C++ */
struct date dob = { 14, 3, 1879 };
```

The C idiom of providing a `typedef` for a `struct` is redundant (and given that this is the case, it is a good way of spotting C programmers masquerading as C++ programmers):

```
typedef struct date
{
    int day, month, year;
}   date;
```

C++ automatically provides the `typedef` when you define the `struct`. An interesting question is what happens to code like this that already provides a `typedef`? Will C code written in the style shown break when compiled with a C++ compiler? No. In C++ there is now no harm in providing the same name for a user defined type as the one it already has:

```
typedef date date;
```

A little pointless, but certainly harmless. A bit like being able to use your own area code to dial a local number (at least in the UK).

One arbitrary restriction in C that C++ removed was that a `struct` need not have any members. This may seem odd at first, but it allows you to implement pure opaque handle objects. However, this does not mean that it has zero size; you might regard the non-zero size as "pure padding".

Comfortable *enum*

The same rules regarding tags for `struct` apply to `enum`s. Another of those oddities in C that got cleared up in C++ was that `struct` tags were not only in a different namespace to type names, but were also in a different namespace to `enum` tags. After the unification of namespaces in C++ the following is not possible:

```
/* C not C++ */
enum date { fields, day_no };
struct date { int day, month, year; };
```

The most significant change to enumerations is – yes, you guessed it – they are more strongly typed than in C. It is good practice in C to treat each `enum` type as a distinct type in its own right, and not as the ordinary integer type it truly is. This kind of thing is usually picked up by checking tools and your colleagues. It is recommended rather than required.

A C++ `enum` is more strongly typed – and is a genuinely distinct type – but not so strongly typed to be as infuriating as Pascal's enumerations. `enum` constants may still be given compile time constant values:

```
enum date_option
{
    ordinal_day = 1 << 0,
    month_name  = 1 << 1,
    short_year  = 1 << 2
};
...
date_option option = short_year;
...
cout << (option == short_year ?
         dob.year % 100 :
         dob.year);
```

In addition, an enum value may still be used as an integer value:

```
enum day
{
    sunday, monday, tuesday, wednesday,
    thursday, friday, saturday
};
const char *const day_name[] =
{
    "Sunday", "Monday",
    "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};
...
day today;
...
cout << day_name[today] << endl;
```

However, a variable of some `enum` type cannot be used as an integer lvalue, i.e. you cannot assign or initialise an `enum` from an integer:

```
/* C not C++ */
today = -1;
today = 2;
```

This is regardless of the fact that the integer may in fact have the same value as one of the enumeration constants. If you want this conversion to happen, you must do it explicitly:

```
today = (day) 2; // required in C++
today = day(2);  // alternative syntax
```

As shown, C++ also supports a function style cast which is a bit neater than the traditional cast form. You should prefer this new form to the old one where possible – it is not always possible, as types with more than one specifier in their name must use the old style, e.g. `char *` or `unsigned long`.

The only areas where C programmers are likely to find these changes restrictive is in iteration and bitsets:

```
/* C not C++ */
++today;
option = ordinal_day | month_name;
```

I will show a technique for getting around the first restriction in a future article. I'm not sure I have a whole load of sympathy with their use in bitsets as I regard this as a misuse of `enum`s, and a fundamental misunderstanding of the relationship between types: the type is being used both as the container *and* the contained type. The container should be an appropriately sized unsigned type, and the `enum` type should be used only with the values specified for it, otherwise there is not a lot of point in using an `enum` except to save typing (as opposed to enforcing typing).

That said, you are guaranteed that an `enum` can hold values up to the nearest whole power of 2 above its maximum enumeration constant value. Although in principle a C implementation is free to chose an appropriately sized underlying integer type, the norm is simply to use `int`. C++ implementations tend to opt for smaller sizes where possible. C++ also permits `enum`s with ranges greater than an `int`, for instance where `long` is larger than `int` and at least one of the enumeration constants can only be represented as a `long`.

An interesting point of trivia is that a C++ `enum` type need not have any enumeration constants:

```
enum noenum {};
```

State of the *union*

Unsurprisingly, the tag rules are the same for `union` as they are for `enum` and `struct`. The key extension is the anonymous or unnamed `union`. A `union` defined and declared inside a `struct` has always needed a name for the `union` as well as a name for its members. This is in contrast to variant records in Pascal in which the members of the variant are in the scope of the surrounding record. This is now possible in C++. The following example shows a more generalised date structure that one might use for a system in which the date may be represented either in terms of fields (e.g. *DD/MM/CCYY*) or as a day number (e.g. the Julian Day is the number of days since some time in 4713 BC):

```
struct fields { int day, month, year; };
typedef long day_no;
enum format { fields, day_no };
struct date
{
    format type;
    union
    {
        fields as_fields;
        day_no as_day_no;
    };
};
...
switch(dob.type)
```

```
{
case fields:
    cout << dob.as_fields.day   << '/'
         << dob.as_fields.month << '/'
         << dob.as_fields.year;
    break;
case day_no:
    cout << '#' << dob.as_day_no;
    break;
}
```

This simple extension will find its way into C9X. There is an even more elegant way of handling this sub-typing relationship in C++, which I will cover in a future article.

It can also be used to declare free-standing variables outside of a `struct` or a `union`. The constraint on this is that these variables must either have no linkage, i.e. they are `auto` variables, or internal linkage, i.e. they must be `static`. In short, no anonymous `union` may have external linkage and be accessible to other translation units.

C++ also allows a `union` to have no members. Hardly a controversial change, but if you still doubt that this may have any use consider justifying a comparable state of affairs in C: a `union` is entitled to have only one member....

Summary

- Tag names are type names in C++.
- C++ `enum` types are distinct types in C++. They are readable, but not writable, as integers.
- Function style casts often provide a more readable alternative to the traditional cast notation.
- The members of anonymous `union`s are members of the enclosing scope.
- C++ supports `//` line comments.

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

The Pitfall of Being Ignorant by The Harpist

The following article was published in an Internet Newsletter from a reputedly high quality start-up company. For various reasons I have deleted all identifying marks but otherwise this item is exactly as published. Read it carefully, then I will comment.

Note that this was a published article, not a private email. This is important because many of us are guilty of silly oversights when writing just for the benefit of a closed circle. But, when we write for Joe Public it behoves us to be much more careful, particularly if we are publicly criticising the work of others.

An Obscure Pitfall of C++ by XXX XXXX

I've been at XX for close to a year now, and like some of you, my first exposure to C++ was when I started messing around on the XXX. C++ is without question quite a confusing monstrosity of a language; sometimes I wonder what good ol' Bjarne and his friends in New Jersey were thinking when they included some of the more arcane aspects of the programming language discipline in C++. At any rate, it can't be repeated too often that programmers are well advised to be very careful when using some of the more unusual features of the language. This is all the more true in a multithreaded environment, where what appears to be a safe way of doing things actually isn't.

Since I'm responsible for maintaining the core libraries for the XXX, I've gone through quite a bit of C++ code over the past several months. Never has reading code been as daunting a task as when I had to pore through the implementation of the Standard Template Library that was at our disposal. It

was even more daunting when I had to narrow down some obscure bugs. Based on this experience, I'd like to discourage a practice that seems somewhat common in the C++ community.

Rule: Don't declare member variables of classes static, if the member variable is of a nonfundamental type.

This lesson may be hard to understand to those who don't violate it, so let me present the following situation. Sometimes you need a flag or counter that is common to all instantiations for a particular class; for example, a flag to indicate that some common initialization for all objects of this class has occurred. Often this flag is also a non-fundamental type, because it has some sort of mutual exclusion built into it, or some such thing.

I've noticed that many programmers are tempted to declare these flags and counters as static, private members of the class, so that only one such object is instantiated for all instantiations of this class itself:

```
class B {
public:
     B();
     ~B();
}

class A {
public:
  A();
  ~A();
  ...
private:
     static B flag;
}

static A::flag = 0;

A obj1, obj2;

A::A()
{
  if (A::flag == 0) {
    do blah;
  }

  A::flag++;
  ...
}
```

Now this is a perfectly fine idea, except that it opens one up to all sorts of nasty race conditions.

What tends to happen is that A::flag will generally be used in the constructor for A itself; after all, the purpose of the flag is to let A's constructor know whether it needs to do any other initialization.

However, notice the circular dependency -- the constructor for A depends on B. However, the C++ standard doesn't define an order in which global objects are constructed. A therefore depends on an object that may not yet have been constructed -- that is, B -- and then all hell breaks loose.

The value of the flag is unpredictable; it may or may not be correct at any given time. An identical problem occurs if there's a dependency between destructors.

This wouldn't occur if object B was of a fundamental type, such as int or char, since fundamental types aren't constructed in the canonical C++ sense.

In this type of situation, it's likely that this flag is used in the constructor for the class of which it is a member. Recall that the flag was declared static, meaning that it's actually a global object in its own right.

The solution? Declare the object static and local to the constructor:

```
class B {
public:
     B();
     ~B();
}

class A {
public:
     A();
     ~A();
     ...
}

A obj1, obj2;

A::A()
{
  static B flag = 0;

  if (flag == 0) {
    do blah;
  }

  flag++;
  ...
}
```

Now everything's fine. And as a side benefit, the code's even easier to understand.

A Critique

I often hear people claim that we no longer need edited, printed publications any longer because it is much simpler to publish electronically. Well maybe, but the evidence of the above does not support that contention. The electronic newsletter in which this article appeared was not a hobbyist's doodles but published by a company who would like you to be impressed by their products. Internal evidence suggests that the author is believed to be a company expert on C++.

He makes a number of assertions about C++. Some of us might agree with some of the things he says. C++ is a very large and complicated language but a genuine expert would raise direct examples of problems rather than putting up a completely bogus Aunt Sally.

Let me deal with the trivial to start with. His code is syntactically wrong, he always forgets to close his class declarations with semicolons.

Much worse than that, look at the line:

```
static A::flag=0;
```

What is the type of A::flag? OK, small problem. However, what is the meaning of static? A novice error that you should sort out on the first day that you declare a static member. Of course in a global context static means something quite different and hides the name in the file where it is defined. That might cause more than a little problem with linkage.

I will come back to some other coding problems in a moment, but before I do let us look at the design. Under what circumstances would a flag be anything other than an integer type?

I am also curious about the `flag++` concept. If flag is not an integer type, what will incrementing do? Also, I note that the writer uses an identity operator on `flag` and compares it with 0. It is clear that the only conceivable user defined type that would make sense in this context would be an enum.

I would have liked to have seen some genuine code, rather than this patently contrived example. It seems to me that the author is concerned with cases where something needs to be done the first time a class is instantiated. There are such cases; for example, you might want to provide a pool of dynamic memory. But in all such cases the use of a user-defined type would be bizarre. This kind of action would be better encapsulated as a member function called by the constructor.

Now what about 'non fundamental'? What does he mean? What he should mean is a type that has a non-trivial constructor.

Global instances of any type where the constructor has nothing to do will be statically initialised (to zero if not specified). Note that all this does is to guarantee that you will not get undefined behaviour. For example:

FILE1.CPP

```
extern int j;
int i=j+2;
```

FILE2.CPP

```
#include <iostream.h>
extern int i;
int j=i+4;
int main() {
  cout<<i<< "  " << j << endl;
  return 0;
}
```

This might not always result in `j=6`, but `j` will have a defined value (I think) but one that depends on the order of linkage of the files. Of course such circularities are stupid and are an excellent reason for avoiding global variables.

OK, C++ is complicated, but the author is claiming to be expert enough to locate subtle bugs in the STL implementation. There are some subtle problems with the STL, but they are largely in the specification and using it properly. It is far more likely that there is a flaw in the user understanding than a bug in the implementation.

Now let us move on a little. As long as you define[1] your static members in the file where they are first used and before their first use, the language requires that initialisation will happen. The problem is what constitutes a use? In the example code, is it the constructor that uses the static? Or, is it the definition of the global variables? The answer is, the constructor. This means any static member used in a member function should be defined in the same file as the function definition and precede it. But, that is exactly what you will have been taught by any competent trainer. Your class implementation file starts with definitions of the static members of the class and continues with the definitions of the member functions, constructors and destructor.

The code as written has no order of initialisation problem. But, there might be a problem if the writer moves the member function definitions of class A and class B to their own files (as he should), but forgets to move the static data definitions.

This has one small consequence; you must not use static data members in inline functions that might be used by constructors. For safety this should be an absolute rule, and not just for classes with non-trivial constructors. The following code has a potential for nastiness:

```
class Nasty {
  static int i;
public:
  static int ival () { return i; }
};
```

Of course, this will not result in undefined behaviour, but like the example above it just might result in unexpected behaviour. Not the same thing but still embarrassing. Of course, in order for this problem to surface you will need to use `ival` in the dynamic initialisation of a global object. As inline functions are always visible to the user of a class the problem can be spotted and avoided even if the server class has been carelessly implemented.

Now let us look at the solution proposed by the writer. This is fatally flawed. Take a moment to think about it. Go on, go back and look. Have you seen it? Exactly! What happens if class A has another constructor? His code is a perfectly correct solution for an entirely different problem

Now let me lead you again through the correct way to tackle the order of initialisation problem. Let's start with rewriting that unpredictable program above.

FILE1A.CPP

```
extern int& j();
int & i(){
  static int _i=j()+2;
  return _i;
}
```

FILE2A.CPP

```
#include <iostream.h>
extern int & i ();
int & j() {
  static int _j=i()+4;
  return _j;
}

int main() {
  cout<<i()<< "  " << j() << endl;
  return 0;
}
```

Now, by moving the static variables inside a function we have taken control of the order of initialisation. A consistent and predictable ordering has been imposed.

[1] As opposed to declare, which is what you will find in the class definition. Confusing I know, but remember that names can be declared many times and must only be defined once, which is why we have to define static members outside the class.

Now it should fail because of the mutually recursive initialisation. I believe that C++ has grasped a problem that C has never clarified by declaring that such code produces undefined behaviour. Which means a good code checking tool should spot the problem. If you want to see how the problem might surface in C, consider the following:

FILE1B.C

```
extern int* j();
int * i(){
  static int _i=*j()+2;
  return &_i;
}
```

FILE2B.C

```
#include <stdio.h>
extern int * i ();
int * j() {
  static int _j=*i()+4;
  return &_j;
}

int main() {
  printf("%i  %i\n", *i(), *j());
  return 0;
}
```

Now perhaps one of the C experts can tell us what the C standard says about such mutually recursive initialisation of local statics does.

To Summarise

1. Do not use global variables, wrap them in functions returning a reference. This idiom should always be applied to dynamically initialised global variables

2. Think very carefully before using static data members in inline member functions. There is probably a very subtle problem lurking for those that ignore rule 1.

3. Learn to do it properly before making unsubstantiated criticism of the work of others. Sure, I may be wrong in some of the above, but I have put it out front where you can tear it to shreds.

4. Work should at least have the more glaring mistakes removed before publication. Editors of traditional hard copy publications try to do this. It is rare that the editor of an electronic publication makes the effort to polish work before publishing.

5. Do not believe that just because the writer works for a named company that they know anything.

6. Companies would be well advised to get an outside editor to look at the prognostications of their local experts before letting it loose on an unsuspecting public.

Finally, there are many things wrong with C++ but the biggest one is a lack of knowledgeable training. The second biggest is failure by companies to get their programmers trained. In the UK, if a presenter of C++ training is a member of ACCU the odds are that you will get reasonable quality, if they are not the odds are very high that you will not. That is not speculation but a pragmatic judgement. At least ACCU members will tell you that they might be wrong.

The Harpist

Self Assignment? No Problem!
by Kevlin Henney

The First Rule of Optimisation: Don't do it.

The Second Rule of Optimisation (For experts only): Don't do it yet.

Michael Jackson

The classic problem of self assignment was revisited by Francis in the last issue [1]. The standard form was captured by Coplien as part of his *Orthodox Canonical Class Form* [2].

```
type &type::operator=(const type &rhs)
{
  if(this != &rhs)
```

```
  {
    appropriate copy and release actions
  }
  return *this;
}
```

This is the basic schema that you should seek to follow for all your assignment operators. Francis wants to call the general applicability of this into doubt for reasons of efficiency, but before I deal with that issue specifically I think it's important to establish what we mean by Orthodox Canonical Class Form and what we hope to achieve by it

Orthodox Canonical Class Form

> **orthodox** *adj*. conforming with established standards, as in religion, behaviour, or attitudes. [3]
>
> **canonical** *adj*. (of an expression, etc.) expressed in a standard form. [4]

The OCCF is a *recommendation*, not a *rule*:

> Programming standards must be valid both for newcomers and for experts. This is sometimes very difficult to accomplish. We have solved this problem by differentiating our guidelines into rules and recommendations. Rules should almost never be broken by anyone, whereas recommendations are supposed to be followed most of the time, unless there is a good reason not to. This division allows experts to break a recommendation, or even sometimes a rule, if they badly need to. [5]

It is not something to be followed slavishly, but it has an important property: it works, it is safe, and as an idiom, clearly communicates its purpose to readers. You can have greater confidence in something that is written following this form than in something that has not been.

Confidence and intrinsic quality

Confidence is not something woolly that should be underestimated or ignored in the process of software development; it is essential. I have been presented with a piece of code that looks like it grew on a spaghetti tree, greeted it with a perplexed expression, and then been told chirpily by its author "not to worry; it works" *. Great. Not.

Not only should a piece of software "work" (what this means is a whole topic in itself, but I'm sure you can come up with a number of plausible consensus definitions) but it should also look like it works. Commercial code is not written solely for the benefit of its author – although clearly the industry would empty out in the absence of any such gratification – and the idea that the only deliverable is a piece of executable code at the end of a waterfall development process should be greeted with the derision (as well as project failure) it deserves.

If you can understand the code by its form it will be easier for you to both have confidence in it and to spot any mistakes. If you cannot *see* that a piece of code is correct, how can you have confidence that it *is* correct? Executing it is not the answer: dynamic bug hunting and bashing is a poor substitute for code that is internally well structured – like other forms of hunting in a modern society, it is unnecessary and barbaric. The concept we are identifying here is that of *intrinsic quality* [6].

* I am reminded of an occasion when I was working onsite and needed some new code from someone back at work. I said that I didn't expect it to be fully tested as he didn't have the right environment in which to do this. I was a little perplexed when the code arrived and failed to compile. Looking at the code I then understood why: there were basic syntax errors all over the place, and the code could never have been compiled. This was confirmed when I rang him to discover that because I had not expected full testing, he had taken this to mean that he didn't need to compile it either. In his vocab "compile" and "test" had somehow ended up as synonyms! Confidence was *not* high.

Orthodoxy and heresy

So there is a great deal of benefit in following a standard form for something that could otherwise give rise to obscure and unsafe behaviour. One of the aims in programming is to be precise. If you are not being precise, you are being vague. If you are being vague, you don't need the help of a programming language – I often find that beer is a far better medium for this.

But as I said, this is a recommendation and not a doctrine or religious law. What to do if you feel an alternative solution is more appropriate? Will you be cast out from the gates of the C++ programming community and roasted over a code review? Should you just rebel outright, go off and establish your own orthodoxy? Nothing quite so dramatic in fact: a comment will do. Just show that not having an explicit self check was considered, but deemed unnecessary as the code presented is already safe.

The important property of the canonical form is that it is based on some guarantees of behaviour; a specification. Whatever code structure you settle on should satisfy this specification. To return to the idea of rules and recommendations, the OCCF is a recommendation but the spec it is based on is a rule:

> *Rule 5.12* Copy assignment operators should be protected from doing destructive actions if an object is assigned to itself. [5]

The code that Francis presents fulfils this criteria; although it departs from the standard form it fulfils the same set of requirements. In short, it works.

Equivalent Forms

The basic structure of the code offered by Francis can be summarised as

```
type &type::operator=(const type &rhs)
{
    take a copy of rhs's resources
    release existing resources
    bind copy to self
    return *this;
}
```

The ordering of copying and release are required as it is this control flow that ensures self assignment is not a problem. In the event of `&rhs` being the same as `this` we will waste a bit of time making a redundant copy of the current object, releasing current resources and then reassigning the copy. Perhaps the redundancy is not so aesthetically pleasing, but it is certainly safe and it will not be executed commonly enough to make it an issue. Perhaps the only thing missing is a comment (note: "comment", not "essay") stating that this code is self copy safe.

The applicability of this is for dynamically allocated representation, typically a single pointer to an object, that can be easily copied (where I mean a copy based on the statically declared type) or cloned (a copy based on the dynamic type – giving rise to the concept of *type shallow* and *type deep* copying).

What we have is the idea of behavioural (or *black box*) equivalence. Given the basic requirements we have outlined, this structure is substitutable for the OCCF.

Don't optimise

So in the name of overall efficiency and correctness no problems. Francis' motivation, however, is questionable:

> [T]he cost of making the check for self assignment is some kind of comparison and branch statement. Branches are bad news on pipelined architecture. If we can write code with fewer branches we should do so. [1]

Where did this sudden concern for efficiency come from? It certainly wasn't measured and was not found to be a bottleneck in a

real application. This misplaced concern for code level efficiency is the kind of thing that has been shown time and again as subordinate to optimisation through effective data structure and algorithm use. It is exactly the attitude and approach that is often held up as poor programming practice. I'm afraid in this case I am not going to contradict such received wisdom.

Let's take a look at some code:

```
type &type::operator=(const type &rhs)
{
    rep_type *new_body =
          new rep_type(*rhs.body);
    delete body;
    body = new_body;
    return *this;
}
```

In structure this is similar to Francis' code. No branches? Take a look at a pseudo-assembler output:

```
  push    sizeof__rep_type
  call    __op_new ; operator new
  move    new_body, result

  compare new_body, null
  jmpifeq postctor

  push    rhs + body
  push    new_body
  call    rep_type__ctorcp
                   ; rep_type::rep_type

postctor:
  compare this + body, null
  jmpifeq postdtor

  push    this + body
  call    rep_type__dtor
                   ; rep_type::~rep_type

postdtor:
  push    this + body
  call    __op_delete
                   ; operator delete

  move    this + body, new_body

  move    result, this
  return           ; return *this
```

That's right, there are two implicit conditional branches:

- A null return from a `new` should not have a constructor called on it; and
- A null pointer should not have a destructor called on it before being handed to `delete`.

In truth a case of two rather than three branches, as opposed to zero or one. How great was this saving? Look at everywhere there is a `call` instruction. This means we are calling four other functions, two of which we know deal with heap management. Against that backdrop, the extra couple of instructions from an explicit self check look even less clock threatening than normal:

```
  compare this, rhs
  jmpifeq wayout
                   ; if(this != &rhs)
  ...
wayout:
  move    result, this
  return
                   ; return *this
```

The level of optimisation we have achieved is what the phrase “a drop in the ocean” was intended to describe – if we used the word “optimise” anywhere near such code we would be deceiving ourselves.

Don't optimise yet

The next claim to investigate is that of branches – specifically conditional branches – on pipelined architectures. Eliminating them because of some hoped for optimisation is as rational as not walking under ladders based on superstition – there are times when it is unwise to do so, such as someone already up the ladder with a tin of paint, but that kind of judgement is not the same as superstition. So clearly we need to understand something about both conditional branching and pipelining before making a decision.

Some uses of conditional branching are simply the result of poor basic programming skills:

```
if(enabled)
    enabled = false;
else
    enabled = true;
```

Illustrates the weak grasp the programmer has of logic. You don't need to be a Vulcan to write and comprehend:

```
enabled = !enabled;
```

I agree with the basic tenet that we should write fewer control structures. A well abstracted system tends to encapsulate control flow within operations. Examples of this include polymorphism over explicit `switch` code, STL's combination of iterators and iterator algorithms, and the Enumeration Method pattern [7].

But many conditional branches are a fact of life: it is difficult to eliminate them if they are intrinsic to a problem description. How many branches are there in the following code?

```
if(year % 4 == 0 && (year % 400 == 0 ||
year % 100 != 0))
    cout << "Leap!!!" << endl;
```

Three. One for every condition: remember, C++'s built-in conditional operators are short circuiting.

An instruction pipeline contains instructions pre-fetched for execution. The many stages of an effective pipeline might include fetch instruction, decode instruction, calculate operands, fetch operands, execute instruction, and write operand result. Running these in parallel rather than in sequence is a very effective processor optimisation. The only fly in the ointment appears to be that a branch in the control flow may invalidate the instructions in the pipeline: one branch is pre-fetched. What if the other is taken?

It would be surprising and unfortunate if such an elegant architecture had not been fully thought out – and then it would be, as Francis suggests, "bad news" – but fortunately the impact of branches is anything but devastating, and pipelined chips sell and perform very well. One solution is to use a multi-stream architecture, i.e. you can hold more than branch at the same time. Branch prediction and delayed branching are more cerebral in their approach. Perhaps the simplest approach used is that the instruction stream following the branch instruction is loaded, i.e. what would have happened in the pipeline anyway.

How much of an impact does this last approach have on the code we have examined so far? None whatsoever. If you look at how the code is arranged, it is the common case that immediately follows the branch, and the uncommon one that must be branched on. If you wanted a rule concerning branches that took this into account it would be a simple one:

> Place the commonly executed code nearest to the condition that tests for it.

Interestingly, this is what many programmers tend to do already, but for readability reasons:

> Given an `if else`, the `if` body should deal with the common case code, and the `else` body with the more exceptional occurrence. If they are equally valid, i.e. neither is exceptional, then the order is best determined by the most positive phrasing of the condition, i.e. the equivalent expression with the least contorted logic.

It is often said that cleanly structured code tends to be more efficient than code whose guiding philosophy has been one of successive application of folklore optimisations. This case seems to vindicate that.

Relative merits

We have looked at behaviourally equivalent forms, but there is a stronger equivalence that is hinted at in the recommendation given above where I mention "equivalent

expression" for a condition. For built-in types (and, one would hope, user defined types) an example of strong logical equivalence would be that `!(a == b)` and `a != b` have the same meaning and are fully interchangeable.

One point that Francis raises in his article is the amount of time spent by people deciding on whether:

```
type &type::operator=(const type &rhs)
{
  if(this != &rhs)
  {
    appropriate copy and release actions
  }
  return *this;
}
```

Or:

```
type &type::operator=(const type &rhs)
{
  if(this == &rhs)
    return *this;
  appropriate copy and release actions
  return *this;
}
```

Is the better alternative. These are equivalent in the sense that they have identical meaning, and one can be transformed into the other by a good compiler. If you are wondering which way such a compiler would lean, look back at some of the points we have discussed. That's right, the scruffy multiple return version is less optimal than the version that uses the structured programming form †.

However, few compilers do that well so you are left with a separate set of concerns to balance. The common case is that the left and right hand side of an assignment are not the same, so if your interest is either pipeline efficiency or layout you would chose the first example. A direct translation of the second example into assembler tends to result in two jumps:

```
  compare this, rhs
  jmpifne postif ; if(this == &rhs)
  jmp     wayout
postif:
  ...            ; perform copying, etc.
wayout:
  move    result, this
  return         ; return *this
```

For C++ it is important that common function exit code is shared as this can involve destructor calls, which, if space is your concern, you would not wish to have duplicated at every return point. If you ask to optimise the second example for speed you will probably end up with duplicated code:

```
  compare this, rhs
  jmpifne postif ; if(this == &rhs)
  move    result, this
  return         ; return *this
postif:
  ...            ; perform copying, etc.
wayout:
  move    result, this
  return         ; return *this
```

It is interesting that we can arrive at the same conclusion from two completely different approaches; it says something about the relationship between forms at different levels. I personally side with those whose concern is the structure of the written code – my reasons for this are based on the belief that software development is an engineering profession, albeit an immature one. There are a few people who need to be concerned with the machine level, but that figure is far smaller number than the number who concern themselves with it.

Stable Intermediate Forms

Returning to Francis' proposed code structure: although arrived at from a faulty line of logic, it is sound. For those that are interested in patterns, what we have here is a language level pattern (better known as an idiom) that has a well defined context, i.e. C++ copy assignment operator for an object

† It has been said that in the light of modern optimising techniques based on data rather than control flow, he wishes he had not included any jump statements (a function return statement and a loop exit) in Oberon (MODULA 2's successor) as the discontinuities introduced into the control flow are not only inelegant, but they thwart a number of optimisations.

structured using the Handle/Body idiom [2, 8] (more generally, the Bridge pattern [9]) where the body is easily copied (either shallow or deep with respect to its type). The proposed configuration is something that works, meeting all the requirements for an assignment operator.

Exception safety

However, a pattern has three essential parts: context, forces and configuration [10]. The conflicting forces that are listed for this pattern are at fault, and hence it is not a pattern. But the context is valid and the configuration seems to have some merit, can we say something more about it? Alan Griffiths, in his role as editor, commented on Francis' solution:

> 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. [1]

My only caveat to this is, as we have shown, that exception safety is the *only* benefit of this approach – as an issue, processor cycles are not even on the radar. In addition to the usual forces describing the requirements on a copy assignment operator, exception safety is the most important force resolved. Let us examine the problem solved:

```
1. release existing resources
2. take a copy of rhs's resources
3. bind copy to self
```

What if an exception is thrown during step 2? The object remains in existence, but it now has a chaotic and unstable state: its resources have been released, but it still refers to them. What will happen on destruction of that object? That's right, destruction of a completely different kind! Objects in an unstable state cannot be destroyed without spreading that instability to the rest of the program. However, there is no safe and consistent way to stop an object from reaching the end of its life. To put it mildly, this is a non-trivial issue.

The solution is to ensure that at every intermediate step the object has a coherent state, i.e. not only is the result of every macro change stable, but each micro change from which it is composed is also stable. This principle of *Stable Intermediate Forms* underlies successful software development strategies [11] as well as other disciplines of thought and movement, e.g. T'ai Chi.

```
1. take a copy of rhs's resources
2. release existing resources
3. bind copy to self
```

This sequence resolves the forces. It is also sufficiently general that it is possible to use this with the original OCCF – for instance, when writing a copy assignment for a class whose objects have a mixed style of representation.

A pattern

In summary, the many concerns facing a developer branch into a myriad forces which fall somewhere between “challenging” and “daunting” in the software engineer's dictionary. Compared to other industries, software development sports a high number of people that can juggle. In this light it is perhaps easy to see why.

When it came to branches I believe that Francis was barking up the wrong tree. Closer inspection revealed a sound solution to a different general problem, and a documentable pattern:

Exception Safe Handle/Body Copy Assignment

Problem

- Ensuring copy assignment in C++ is exception safe.

Context

- A class has been implemented as handle/body pair.
- The body is copyable – type shallow or deep as appropriate.

Forces

- Any of the steps taken in performing the assignment may fail, resulting in a thrown exception. Partial completion of the steps may leave the handle in an unstable state.
- The result of assignment, successful or otherwise, must result in a stable handle.
- Self assignment must also result in a stable handle.
- After successful completion of the assignment the handle on the left hand side of the assignment must be behaviourally equivalent to the handle on the right hand side.
- Assignment, successful or otherwise, must be non-lossy, i.e. no memory leaks.

Solution

- Perform the body copy before releasing the existing body.
- Bind the body copy to the handle after releasing the existing body.

Resulting Context

- The existing body is not deleted before the body copy has been attempted. Therefore, a failed body copy will not result in an unstable handle.
- Failed body release may still result in an unstable or lossy handle. However, throwing exceptions from destructors is a practice commonly cautioned against.
- The ordering accommodates safe self assignment at the cost of a redundant copy.
- If the body copy preserves behaviour equivalence, a successful assignment will preserve it for the composite handle/body object.
- The solution can be used in conjunction with the schema for copy assignment from the Orthodox Canonical Class Form.

The issue of a failed deletion is an interesting one. It is left, as they say, as an exercise for the reader to resolve.

Kevlin Henney
kevlin@acm.org

References

1. Francis Glassborow, "The Problem of Self Assignment", *Overload* 19.
2. James O Coplien, *Advanced C++ Programming Styles and Idioms*, 1992, Addison-Wesley.
3. *The Collins Concise Dictionary*, 1988, Collins.
4. E J Borowski and J M Borwein, *Dictionary of Mathematics*, 1989, Collins.
5. Mats Henricson and Erik Nyquist, *Industrial Strength C++: Rules and Recommendations*, 1997, Prentice Hall.
6. Les Hatton, *Safer C*, 1994, McGraw-Hill.
7. Kent Beck, *Smalltalk Best Practice Patterns*, 1997, Prentice Hall.
8. James O Coplien, *Software Patterns*, 1996, SIGS.
9. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995, Addison-Wesley.
10. Christopher Alexander, *The Timeless Way of Building*, 1979, Oxford University Press.

11.Grady Booch, *Object-Oriented Analysis and Design with Applications*, 1994, Benjamin/Cummings.

Ed: For the word of lore on processor pipeline architectures and branch prediction statistics see: Hennessey & Patterson. Computer Architecture: A Quantitative Approach., 1990, Morgan Kaufmann Publishers Inc.

Lessons from fixed_vector Part 1
by Jon Jagger

When I read and think about Overload articles I always learn new things and am reminded of things I have already learned. That's one of the reasons I subscribe to Overload. In this article I'm going to recount some of my thoughts[2] as I read about the excellent `fixed_vector` class presented by Kevlin Henney in Overload 12. First, a quick recap of `fixed_vector`.

```
template<typename type, size_t size>
class fixed_vector
{
public:
       iterator begin();
       iterator end();
private: // state
       type base[(size>0) ? size : 1];
};
```

`fixed_vector` Implementation

`fixed_vector` provides a safer form of C arrays while at the same time being as close to an STL `vector` as possible (but no closer). An interesting question is 'How close should it get?'

`fixed_vector` is implemented using a plain old C [array]. This means that the template type must have a default constructor, which is not true of a `vector`.

```
class ndc     // No Default Constructor
{
public:
       ndc( const snafu& );
};

vector<ndc> good;

// COMPILE TIME ERROR
fixed_vector<ndc,16> bad;
```

`fixed_vector` also allows the creation of logically empty instances such as:

```
fixed_vector<int,0> empty;
```

A `vector` can be empty, so from that point of view it's desirable that a `fixed_vector` can be too. The mechanism by which Kevlin acheived this was to change a logically empty `fixed_vector` into a physically non-empty one. Hense:

```
    (size>0) ? size : 1
```

A perhaps non-obvious effect of this is that a logically empty `fixed_vector` will call the default constructor once.

You might take the view that a `fixed_vector`, while being modelled on STL, is basically a replacement for raw [arrays]. In other words:

```
template<typename type, size_t size>
class fixed_vector
{
public:
       ...
private: // state
       type base[size]; // [3]
```

[2] I feel articles like this, ones that recount the learning process, can make excellent subject matter for Overload. If you have a favourite article (not necessarily from Overload!) that you learned a lot from, why not write an article?

[3] A partial specialisation of this (with size==0) would provide the best of both

```
};
```

Copying a fixed_vector

I can easily imagine many C++ programmers writing the copy constructor for `fixed_vector` like this:

```
fixed_vector(
     const fixed_vector<type,size>& rhs
)
{
  for (int i=0; i < size; ++i)
  {
    base[i] = rhs.base[i];
  }
}
```

Would you? Consider this small fragment of code:

```
class accu {};
typedef fixed_vector<accu,1024> forum;
forum rhs;
forum lhs(rhs);  // copy construction
```

The `forum` copy constructor will call the `accu` default constructor 1024 times, and then the `accu` copy assignment operator 1024 times. If `accu` has non-trivial assignment/constructor semantics this is serious overkill. 1024 `accu` copy constructions would do better. For Instance:

```
fixed_vector( const
fixed_vector<type,size>& rhs )
              : base(rhs.base) {}
```

However, this won't compile since C and C++ don't allow array assignments.

```
              : base(rhs.base)
```

Arrays and member initialisation lists do not mix. However, there is a better way and it revolves around this:

worlds. Kevlin was well aware of this when he wrote his article. However, good articles remain focused and the focus of his article was not partial specialisation

```
typedef struct { int array[32]; }
segment;
segment x,y;
x = y;
```

Now the assignment *is* legal. Wrapping the array in a `struct` makes all the difference. Moreover, a class is just a "higher ranking" `struct`, which means the C++ compiler will provide the default copy assignment operator and copy constructor. What's more, they'll behave exactly as required. For example, the compiler generated copy constructor for `fixed_vector<foo,32>` will do 32 `foo` copy constructions rather than 32 `foo` default constructions plus 32 `foo` copy assignments. However, it's a little more subtle than that. You *have* to let the compiler write the `fixed_vector` copy constructor, since you cannot mimic its action because you cannot copy construct a plain array in a member initialisation list.

A better [array]

Here's the well known `find` function from STL:

```
template<typename InputIterator,
           typename Value> InputIterator

find(
    InputIterator first,
    InputIterator last,
    const Value& v )
{
  while (
    first != last && *first != value)
  {
    ++first;
  }
  return first;
}
```

and a typical use is:

```
typedef vector<int> container;
container v;
container::iterator iter1 =
           find(v.begin(), v.end(), 23);
```

STL was carefully designed so that another use is:

```
int raw[42];
```

```
int* iter2 = find(raw, raw+42, 23);
```

Once you've been using C++ and STL for a while, this somehow seems terribly low level. The `fixed_vector` solution:

```
typedef fixed_vector<int,42> segment;
segment cooked;
segment::iterator iter3 =
 find(cooked.begin(), cooked.end(), 23);
```

fits much more neatly into the STL framework, and it's more robust too. In the raw solution there are two occurrences of the array size, in the cooked solution only one. With `fixed_vector` fully inside the STL fold, you can write universal helper functions like:

```
template<class Container, class Value>
typename Container::iterator
stl_find( Container& c, const Value& v )
{
    return find(c.begin(), c.end(), v);
}
```

which you can use identically with `vector` and `fixed_vector`:

```
typedef vector<int> container;
container v;
container::iterator iter1 =
                    stl_find(v, 23);

typedef fixed_vector<int,42> segment;
segment cooked;
segment::iterator iter3 =
                    stl_find(cooked, 23);
```

That's all for part 1 as the copy deadline is looming. I hope to cover such goodies as writing a `reverse_iterator` in part 2.

Jon Jagger
jonj@dmv.co.uk

Shared experience: a C++ pitfall - By Alan Bellingham

Even in classic C++, without using templates or exceptions, there are some quite subtle pitfalls that one may encounter. The following problem is one I discovered in some third party library code. I'm thankful that I had the source, and was able to discover what was going on.

A class hierarchy using memory management

Consider the following skeleton class:

```
class exampleBase
{
  public:
    exampleBase () ;
    virtual ~exampleBase () ;
    void * operator new (size_t) ;
    void operator delete (void*) ;
    . . .
    void FnExtra (void*) ;
} ;
```

Now, you will notice that, apart from a constructor and destructor, (the latter properly being virtual), it has `operator new` and `operator delete` functions. This would indicate that some form of memory management is being done on a class basis, as we can see from their following implementations:

```
void *
exampleBase::operator new(
                    size_t allocsize)
{
 cout
  << "Base::operator new" << endl ;
 return AllocFromPool(1,allocsize) ;
}
```

```
void
exampleBase::operator delete(
                      void * deadMem)
{
 cout
  << "Base::operator delete" <<endl;
 FreeToPool(1, deadMem ) ;
}
```

So far, so good. We won't worry about the implementation of the `AllocFromPool` and `FreeToPool` functions except to note that the first parameter is the pool number, and that memory allocated from a pool needs to be released to the same pool. For the purpose of illustration, we'll just build versions which tell us which pool we're dealing with, and default to the global allocators:

```
void *
AllocFromPool( int v, size_t sz )
{
 cout
  << "AllocfromPool:" << v << endl;
 return new char[sz] ;
}
```

```
void
FreeToPool( int v, void * ptr )
{
 cout << "FreeToPool:" << v << endl;
 delete [] ptr ;
}
```

Well, we have a base class which presumably does something useful. Let's now consider a second class:

```
class exampleDerived : public
exampleBase
{
 public:
  exampleDerived () ;
  virtual ~exampleDerived () ;
  void * operator new (size_t) ;
  void operator delete (void*) ;
  . . .
  void FnExtra (void*) ;
} ;
```

Again, it appears to be doing its own memory management:

```
void *
exampleDerived::operator new(
                  size_t allocsize)
{
 cout
  << "Derived::operator new" <<endl;
 return AllocFromPool(2, allocsize);
}
```

```
void
exampleDerived::operator delete(
                      void * deadMem)
{
 cout
  << "Derived::operator delete"
   << endl ;
 FreeToPool(2, deadMem ) ;
}
```

Not much difference, you'll note. In fact, the only difference shown in the implementation is that we're allocating from and releasing to a different memory pool – perhaps because the elided parts of the class have created a class with a larger number of data members, and the second memory pool better suits this allocation size.

Now the classes as shown shouldn't cause too many problems. We have a virtual destructor (almost mandatory in a case like this), and if we were being coding this for real, we'd have covered the usual copy constructor and assignment operator issues.

So, what happens here?

```
int
main(int, char **)
{
 exampleBase * eb =
               new exampleBase() ;
 delete eb ;

 cout << "---------------" << endl ;

 eb = new exampleDerived() ;
 delete eb ;

 return 0 ;
}
```

Well, an `exampleBase` is allocated from pool 1, and then released back to it, and an `exampleDerived` is allocated from pool 2, and released back there. With suitable constructor / destructor tracing, we can see this:

```
Base::operator new
AllocfromPool:1
Base ctor
Base dtor
Base::operator delete
FreeToPool:1
----------------------
Derived::operator new
AllocfromPool:2
Base ctor
Derived ctor
Derived dtor
Base dtor
Derived::operator delete
FreeToPool:2
```

Exactly as it should. You'll notice that it doesn't matter that we delete an `exampleBase` pointer pointing to an `exampleDerived` – the correct `operator delete()` function is called because we have a virtual destructor. You can test this by changing the code so that the destructor isn't virtual and trying this.

Gilding the lily

Hmm. Sometimes programmers note common code, and abstract it into a separate function. There doesn't seem to be any reason why the memory freeing code has to be called *directly* from the `operator delete` functions. How about the following small amendment. It should still work, shouldn't it?

```
void
exampleBase::operator delete(
                      void * deadMem)
{
 cout
  << "Base::operator delete" <<endl;
 ((exampleBase*)deadMem)->
                FnExtra( deadMem );
}

void
exampleBase::FnExtra(void * deadMem)
{
 cout << "Base::FnExtra" << endl ;
 FreeToPool(1, deadMem ) ;
}

void
exampleDerived::operator delete(
                      void * deadMem)
{
 cout
  << "Derived::operator delete"
  << endl ;
 ((exampleDerived*)deadMem)->
                   FnExtra(deadMem);
}

void
exampleDerived::FnExtra(void * deadMem)
{
 cout << "Derived::FnExtra" <<endl;
 FreeToPool(2, deadMem ) ;
}
```

That's pretty horrible. Having to do those casts because we know that the dead memory pointers are actually pointers to the relevant classes is yucky, but we know that the only way to the `operator delete` functions is if that *is* so, we know that the memory hasn't been released *yet*, so it's OK, and we don't make use of any member variables *anyway*, so everything is fine.

Isn't it? I mean, it all works …

```
Base::operator new
AllocfromPool:1
Base ctor
Base dtor
Base::operator delete
Base::FnExtra
FreeToPool:1
----------------------
Derived::operator new
AllocfromPool:2
Base ctor
Derived ctor
Derived dtor
Base dtor
Derived::operator delete
Derived::FnExtra
FreeToPool:2
```

Lost in darkness

And yes, it does work. Unless you do what our original programmer then did. Noting that `FnExtra` has the same signature in both classes, and not thinking about the consequences, he made it virtual.

Oops.

```
Base::operator new
AllocfromPool:1
Base ctor
Base dtor
Base::operator delete
Base::FnExtra
FreeToPool:1
----------------------
Derived::operator new
AllocfromPool:2
Base ctor
Derived ctor
Derived dtor
Base dtor
Derived::operator delete
Base::FnExtra
FreeToPool:1
```

Suddenly, the `exampleDerived` objects are being allocated from one pool, and released to a different one. So what's happening?

What is happening is that the programmer has strayed into undefined behaviour. Although *he* knows that the memory pointed to is an `exampleDerived`, the system doesn't. Although *he* thinks that all the member variables are still as they were just before the destructor was called, the system has it otherwise. What has in fact occurred in this case (and using this compiler implementation) is that the compiler's construction and destruction of the object in question has changed the virtual function table pointer.

Now the implementation does this on construction:

1) Allocate sufficient memory for the derived class.
2) Call the derived constructor, which:
2a) Calls the base constructor,
2b) Calls the member constructors, in member order
2c) Executes the code within the derived constructor body.

and at destruction, it dies the following:

3) Call the derived destructor, which:
3a) Executes the code within the derived destructor body
3b) Calls the member destructors, in reverse member order
3c) Calls the base destructor
4) Releases the memory for the derived class

Now, it is a stricture of the language that within the bodies of the constructor and destructor for a class, the object in question must be treatable as an object of that class. In other words, within `exampleBase::exampleBase`, you *are* an `exampleBase` object. This rule has sometimes been stated as "Don't call virtual functions in constructors or destructor", but examination of 'The C++ Programming Language', § r.12.7 should make it clear that a virtual function call within a base class constructor will be routed to the implemenation available to that class, not that available to the derived class. Naturally, the same virtual function call within the derived class constructor *will* route to the derived class implementation.

So what this states is that, given a virtual function call between points 2c and 3a inclusive, we expect the derived class virtual function to be called. At point 3c, we expect the base class virtual to be in effect. Our code is calling it at point 4!

At this point, we've really reached the point of implementation dependence, but what has happened in this particular case is that the compiler, at the transition between point 2a and 2c, inserts code to adjust the virtual function pointer to point at the virtual function table for the `exampleDerived` class. Similarly, on destruction, between points 3a and 3c it adjusts the virtual function pointer to point at the virtual function table for the `exampleBase` again.

It never bothers thereafter to adjust the pointer back again. After all, as far as it's concerned, that memory is now raw storage and its *content* shouldn't be addressed by anyone (§ r.5.3.4). The result in this case was that the wrong function was called, memory was returned to the wrong pool (which rejected it), and a slow memory leak sapped the program.

The fix

The casts we mentioned should have been the warning – what it was telling us is that `operator delete` is effectively a static function, having no *this* pointer. The only other functions we should call from within a static function are also static, but in this case, the programmer *knew* the type behind the dead memory pointer, and cast that to the type. The better solution in the first place would have been this:

```
class exampleBase
{
 public:
  static void FnExtra (void*) ;
} ;

void
exampleBase::operator delete(
                      void * deadMem)
{
 cout
  << "Base::operator delete" <<endl;
 FnExtra( deadMem );
}
```

(with the same adjustments for the derived class too).

Epilogue

The code shown here isn't the original code, but a distillation of the problem: I have no

particular desire to cast stones at a product that has been rewritten since without this code in it. If you are worried that this might affect you, then I'll just say the following – it was version 4.0 of an xBase library, and this was fixed by version 5.0. I make no such guarantee for any *other* library that you may have.

Alan Bellingham
Alan@lspace.org

Further Thoughts on Inheritance for Reuse
by Francis Glassborow

In Overload 17/18, and a letter in Overload 19 I presented some thoughts on 'Inheritance for reuse'.

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 the terms 'static' and 'dynamic'. 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. We talk about binding a name (identifier) to a meaning or behaviour. So a parameter name is bound to the argument by the process of calling the function. A function call is bound to executable code at some stage. This can be at compile time (the default behaviour in C++) or it can be through some mechanism that enables selection at runtime. The latter behaviour is particularly important when the required behaviour is for an object (rather than a value) passed as an argument to a parameter.

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 and also note that parameters are declarations of local identifiers that are initialised by the argument provided at the time the function is called.

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);
};

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

This demonstrates public inheritance. Let's continue with a slight variation on the theme, private inheritance.

Privately Inherited Base

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

Note that this is private inheritance so the only functions that are publicly available are those declared in the definition of `NotBase`.

The first question that may arise in the mind of an experienced C++ programmer is why I would choose to use private inheritance rather than some form of layering or aggregation.

Let me deal with the major possibilities.

Contained Base Reference by Pointer

```
class Choice1 {
  Base * base;
public:
  void fn (int);
};
```

I hope that the problem with this choice is clear to all, I have to complicate `Choice1` with constructors, destructor and a copy assignment operator. Without the user providing these the compiler will generate its own and get it wrong. Replacing '`Base * base;`' with '`Base * const base;`' marginally improves things because now the compiler cannot generate those functions, but you would still have to write them if any were used.

Contained Base Object

```
class Choice2 {
  Base base;
public:
  void fn (int);
};
```

Is substantially better but it inhibits one choice you might wish to make, you cannot cast a `Choice2` object into a `Base` one. You might consider that an advantage but before you come to a final decision consider what a programmer will write if they decide that such a cast should be supported. They will insert something such as:

```
operator Base () { return base; }
```

into the definition of Choice2. Worse they might write:

```
operator Base & () { return base; }
```

Why, I hear you mutter, should a programmer want to do this? Well, knowing that they have implemented the object as a revised/reused `Base` they might also want to reuse some functions with `Base` type parameters. The solution via conversion operators is a disaster because their affect is to provide an uncontrolled conversion to `Base` (either by value or by reference). On the other hand a `static_cast<>` from a derived object to a base one works (certainly on the compilers I have tried though I must confess that I am not certain that it should do so.)

There is, of course, a better option to that of providing conversion functions, which should only be provided if you are sure that the conversion is both safe and desirable. Just provide a perfectly normal member function such as:

```
Base toBase() {return base;}
```

Of course this version:

```
Base & toBaseref() {return base; }
```

breaks data hiding and makes the data available to all and sundry. In other words you cannot do this if you intend writing

robust OO source code. The programmer who provides such conversion functions is the one who has breached the OO principles. Surely it is preferable to use the `static_cast<>` mechanism where the 'guilt' is placed firmly on the shoulders of the programmer who elects to treat a `Derived` object as if it were a `Base` one. In other words, using a private base enables a programmer to publicly 'cheat' if they wish to, but they cannot do so by accident.

Importing Overload Sets

Now let me move on to the problem of providing access to the member functions of a private (or, as you will see, public ones as well) base class. Writing pure forwarding functions can get tedious and the other mechanisms that were available in earlier versions of C++ were easy to get wrong. When the concept of namespace was introduced to the language the keyword `using` was part of the package. The first thing that you must understand is that using is about names, it is not about entities or objects. The second thing is that there are two distinct uses of using. A `using namespace X` directive means that all the names declared in `namespace X` are treated as if they were declared at the point of directive in the current scope. A `using X::name` declaration imports all declarations of `name` from `X` into the current scope. This is not the place to get into the details of namespaces, interesting though they maybe. However, the concept of a namespace bears a considerable similarity to the concept of a class scope. The need to move names from their declarative scope to another one is similar to that we have when we want to move names from a base class to a derived class. There are fundamentally two reasons that we might want to do this. The first is the problem of providing access to a hidden name.

Go back to the first example in this article. Supposing that I want to add an extra overload to a set in the base class. For example:

```
class Extra: public Base {
public:
       void fn(char);
};
```

The existence of that extra overload results in all the original (from `Base`) functions being hidden. What we need is a simple way of 'over-riding' the hiding process. In other words we want to use all the declarations of `fn` from `Base` as if they were declarations in `Extra`. We can now use a using declaration to do just that. So now we can write:

```
class Extra1: public Base {
public:
       using Base::fn;
       void fn(char);
};
```

What we cannot do is to selectively import some of the 'meanings' of `fn` while leaving some of them hidden. If you want to be selective you have no choice but to use forwarding functions.

The next step is to consider the case where we want to import an overload set and over-ride one (or more) of them. The language fixes that simply by saying you can (well it gets a bit more technical when you have to phrase that intent in standardese). So:

```
class Extra2: public Base {
public:
       using Base::fn;
       void fn(int);
};
```

The `using Base::fn` means import all the versions of `fn` from `Base` and then replace (over-ride) the `void fn(int)` version found in `Base` with a new definition provided by the implementation of `Extra2`.

This provides a simple mechanism for importing overload sets from base classes. If you always pair a function over-ride with a `using` declaration for the name involved you might avoid that particularly subtle change of a base class interface —providing another overload to a member function—

that can cause havoc with classes that have been derived to enhance a base class. For example, suppose that the provision of `void fn(double)` in `Base` was a late extra added to fix some defect in `Base`. `Derived` does not inherit this extra behaviour, but `Extra1` and `Extra2` do. In other words, if you want to ensure that your derived class inherits the entire base class behaviour both now and in the future, you need to include `using` declarations for all functions that you over-ride.

Now let me go back to `private` inheritance. You can use `using` declarations to import names from the private base class. If the `using` declaration is `public`, then all the imported names will have the same access as they do in the base class.

I think that this is a big bonus, and for me at least, swings the decision towards using private inheritance for reuse in C++.

Back to static versus dynamic

All the above is fine, but what has it got to do with the problems of reuse? Well the first point is that private inheritance inhibits derived to base class conversions. You can still do them via an explicit cast but they will not happen by accident. As the derived object is not intended to be a subclass of the base, you want that restriction. I hope the first part of this article has shown that private inheritance has some value for reuse but the constraint on derived to base conversions is necessary if the compiler is going to prevent misuse. To understand why this is true in C++ but not in Smalltalk (and Java) we need to look at the way these languages work.

Let us focus on references (pointers in C++ work in a similar way in so far as semantics are concerned). When we declare a reference parameter type for a function we are specifying the minimum requirement. When actually executed the parameter may be bound to any object that is either of the correct type or that has been publicly derived from that type. This can cause no problem in Smalltalk because all functions are dynamically bound. In other words the behaviour of an object reference in Smalltalk must be correct because the decision about over-rides is always delayed until runtime. The cost of this is largely that you almost always pay the performance price for dynamic binding (not a lot but it is there) and there is the potential for the detection of some errors being delayed till execution time. Of course there are ways of working in such an environment which come as second nature to good Smalltalk programmers, but experts in any language know how to cope with problem areas.

I do not know about Smalltalk, but Java has a mechanism by which you can declare a function as `final` and hence not over-ridable. This allows Java to use some static binding but the onus is on the programmer to enable this by explicitly marking the relevant functions as being the final version. This is of relatively little importance from the efficiency aspect because you would not be using Java if you were concerned about such minor performance issues. The intent of `final` is to allow programmers to determine that some behaviour is an immutable characteristic of a class and all its sub-classes. Enabling the static binding optimisation is a small side effect.

C++ is a rather different language in that the default behaviour is static binding. The programmer has to explicitly switch this behaviour off on a function by function basis. That is what the keyword `virtual` is for. The result is that you will only get correct behaviour for a derived object if the object declaration (not a reference declaration) is in scope or if the behaviour has been declared `virtual` in the base class. Maybe you think C++ should have had dynamic behaviour by default and used `static` to mark member functions that were to be bound at compile time. Had that

choice been made, C++ would be an arcane minority interest because there is no doubt that the original converts from C wanted their natural compile time binding to continue.

Conclusion

In my opinion, private inheritance is one of the best mechanisms available for reuse at object specification level. I think that the provision of using declarations has resulted in something that is more powerful than the more traditional route via layering. On the other hand public inheritance for reuse is ill-conceived and should be ruthlessly eliminated from respectable object based/oriented code. I have no doubt that the last statement will annoy some of you. Certainly the technique is better than the cut and paste of source code that riddled the work of the last decade, but why settle for less than the best? Using public inheritance for anything that does not support the 'is-a' relationship is a sure sign that the writer has not yet crossed the line into OO programming. However, you should note that I do not think that the various flavours of object programming technology are the only programming paradigms that should be in use.

Francis Glassborow
francis@robinto.demon.co.uk

Whiteboard

Welcome to a new Overload section. In the past, the established ones, 'C++ Software Development' and 'C++ Techniques', have tended to contain highly studied and polished articles. We hope that this new section will attract shorter discussion pieces, which will address common programming problems and solutions. This is to be a forgiving public forum where praise comes before criticism and there's no explicit guarantee of correctness. So, there's no excuse for not writing half a page about the latest cool thing you've done!

Next issue we'll be launching with a couple of articles exploring Finite State Machines.

In future issues I'd like to see some ideas for ensuring high quality software. The tale below, of my current software troubles, may trigger some thoughts.

For the past couple of years I've been working on a large complex server system. There are a couple of hundred thousand lines of code, with hundreds of concurrent co-operating threads. It's a Voice Mail system that allows you to send and receive voice messages just like e-mail. Since the server is connected to the telephone system our quality requirements have been very stringent. The emphasis has been on reliability above functionality, performance, size, and resource use. It must work for a couple of weeks at maximum load before the product can be shipped. Hurdling this final bar has proved frustratingly difficult. This has been because the quality of our own software was too low, and because we've employed many third-party components.

We ran our software within a number of debugging environments, and pushed the code through various code checkers. Having not implemented this regime from day one we got a lot of noise and little value from the exercise. We found that our own internal debugging solutions bore more fruit. What techniques have you used in your projects to ensure high quality? How have you reduced resource leakage, usage, and contention? How did you increase user responsiveness, and performance?

The foreign components we rely on are the Operating System, Message Store, Message Transfer Agent, Directory, Text-To-Speech Engine, Collection Classes, and Database Drivers. Our software can only be as reliable as the foundations on which it's based. We've experienced our threads entering an API and never returning, API's which take minutes to complete, sub-systems which leak memory, and sub-system threads which throw an exception once a week. What strategies have you used to deal with these sorts of problems?

John Merrells
john.merrells@octel.com

editor << letters;

explicit content

From Kevlin Henney

Referring to the Harpist's article in Overload 19, the issue about whether or not 'explicit' makes sense on a user defined conversion operator is an interesting, and not quite as clear cut as it might first appear.

The reasonsing that if you require a conversion explicitly you can have a named function to do it, and therefore you do not need an 'explicit' qualified UDC (and therefore this feature was left out), falls down on two issues.

One of these is a matter of style and consistency: there are conversions between existing built-in types that require explcit casts, and conversions to user defined types (inward conversions, if you like) can be made explicit. Providing no way to support and enforce this for UDTs breaks this consistency (an issue in that perpetual struggle between built-ins and UDTs for first class citizenship of the language). This kind of argument by aesthetic and consistency does not tend to hold a lot of water in the fierce heat of forging international standards!

The second more compelling reason is that templates were forgotten. It is remarkable how many simple rules of thumb disappear in the light of generic programming—a good example being the use of throw signatures, which were considered a *good thing* until it was realised that they cannot be used safely for a number of templated types. We might write in our template code something like the following:

```
f(x.as_int());
```

Where x has the template parameter type. The only problem is that none

of the built-in types support the as_int protocol! So you will end up

writing

```
f(int(x));
```

Which is what you would have done anyway (or (int), or static_cast<int>()). So you are left with the requirement that to be generic and support conversions you provide both as_int and operator int. And then you realise that you have one implicit form and two explicit forms, so the chances are that the spare form goes, ie as_int gets binned.

The realisation of the impact of templates came too late for the cttee to do anything about (it's not exactly a show stopper) -- everyone had previously bought into the theory that it was superfluous. One of the issues that apparently caused a bit of a bun fight in the original discussion of 'explicit' UDCs was the interaction with inheritance. This is not a hard problem (it has many easy solutions), just one in which many people

hold many different opinions. This climate probably helped get the issue dropped!

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

`auto_ptr` query

From Richard Percy
Replied by Jonathan Jagger

Firstly, I am astonished that we will not have a sensible implementation of a smart pointer in the standard library. I appreciate that standardisation and library design are not trivial, but how can we expect C++ to be taken seriously if our libraries are broken?

I have a lot of sympathy with this view. You are not the only one who is astonished. It is possible that the version of `auto_ptr` that ends up in the standard may be slightly different to the one in CD2.

Secondly, I was disturbed by the reference in Jon's article to sections of Stroustrup's two books. The sections that he lists are concerned with what Stroustrup calls "resource allocation is initialisation".

This means that resources (in this case, heap storage) should be acquired in the constructor of a local object and released in the destructor. This makes the code exception-safe, unless there is too much else going on in the constructor.

Right. And you can use auto_ptr to do this...

```
void fubar::method()
{
  auto_ptr<snafu> p(new snafu());
}
```

The problems I was trying to highlight were those encountered if you go beyond this basic use, namely if you try and copy an `auto_ptr` (either in a constructor or an assignment).

Jon's code example, however, does not follow this idiom because it expects the user of the smart pointer class to allocate the memory before calling the constructor or the reset function.

I'm sorry but I really don't understand what you're trying to say here. You have to allocate the resource before you call the constructor. You can't avoid that. You can couple them very closely...

```
auto_ptr<snafu> latest(new snafu());
```

...but the `new` still occurs before the construction of latest.

A few years back I read a book called C++ Strategies and Tactics, which had a very clear and helpful section on smart pointers, but I think that the author used reference counting. Can anyone follow up Jon's article with an analysis of different implementations of smart pointers?

Richard Percy

I have a library of smart pointers that I've written, `counted_ptr`, `cloned_ptr`, etc, etc. I use them primarily to store polymorphic surrogates in STL containers. I think that they might indeed make a good follow up. No promises, but I'll see what I can do.

Jonathan Jagger
jonj@dmv.co.uk

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!

Credits

Founding Editor

Mike Toms
miketoms@calladin.demon.co.uk

Incoming Editor

John Merrells
4 Park Mount, Harpenden, Herts, AL5 3AR.
john.merrells@octel.com

Readers

Ray Hall
Ray@ashworth.demon.co.uk

Ian Bruntlett
ibruntlett@libris.co.uk

Einar Nilsen-Nygaard
EinarNN@atl.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

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 21* (August/September) should be submitted to the editor, John Merrells <john.merrells@octel.com>, by July 21st.