ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 16

October 1996

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

Subscriptions:
Barry Dorrans
2 Gladstone Avenue
Chester
Cheshire
CH1 4JU
barryd@phonelink.com

£3.50

Contents

Editorial

A fast year

Since your overworked editor has finally got a grip on his personal life (at least temporarily), it seemed a good opportunity to try to catch up with the six issues a year I promised you.

If you all put on your writing hats, I should have enough material to get issue 17 out by New Year!

A lot has happened in the OO world this year. We've seen the C++ committee lurch much closer to a final standard and by the time you read this we may well have agreed to release the second "Committee Draft" – a significant step nearer the International Standard we all want and, in many cases, need.

We've also seen the OO methodology folks settle down with the makings of a Unified Method that incorporates the best of the disparate methods in use (and some of the worst – hey, no-one said it would be perfect!). This will help OO developers in the future convey their thoughts more precisely and reduce the burden on training and relearning. This is just as well given the widespread lack of training programmes in most companies these days.

We've also seen a new light in the OO world. A language that promises portability and simplicity, a faster way to build tomorrow's application: the distributed application. Of course, I'm talking about Java. Is it all hype? Is it the new saviour? Hopefully, you'll all be convinced that it is neither. It is, however, an extremely important development and provides us with yet another tool with which to solve the problems around us.

Although early days yet, Java too will need standardisation in order to "facilitate commerce" as they say in the standards' world. That effort is expected to begin shortly but we do not know yet how it progress. In issue 17 I shall be reporting on the November meeting of WG21 and X3J16 – the C++ committees – but in issue 18 I shall be reporting on the January meeting of SC22's Java Study Group.

English English

Recently I read a complaint about correct English in *CVu* and, by implication, *Overload*. I would respond much as Francis did and say that we as editors do our level best but with the task of technical proofreading as well, occasional errors in English slip past. Some a spell checker would catch and some they wouldn't. I actually don't use a spell checker for *Overload*, preferring to proofread and correct by hand. In this issue, it led me to replace "draw" with "drawer" which a spell checker could not have caught. If this issue contains more oddities than usual, I can point at least part of the blame at MS Word. After a fashion, that is. I've just upgraded my main system from a 68040-based Mac to a PowerPC-based Mac and therefore purchased another copy of MS Office (to obtain the 6.0.1 Word upgrade)[1]. This issue was therefore prepared with a "clean" installation of Word and has made me realise how much customisation I had done on the old system. In particular, I had fiddled with many of the "Intellisense"™ settings to suppress much of Word's "helpfulness". Some of its helpfulness may have caused unwanted "corrections" in this issue...

On the subject of English however, I will remark that probably the most irritating things I find in contributions are "ie." and "eg." when the correct forms are "i.e." and "e.g.", usually followed by a comma. Of course, this is something that a spell checker **will** correct but whilst readers might berate me for missing such an error (unlikely, I suspect), they forget that most of the spelling errors that creep in are actually because the original contributor does not run a spell checker prior to submitting copy!

That's rich!

Formatted contributions now seem to be coming in mainly as Rich Text Format as previously requested. Thankyou for that – it really does make my life easier.

The Editor
overload@corf.demon.co.uk

[1] Word of principle: I am one of those apparently rare home computer owners who insists on **buying** all their own software (and shareware!) to stay legal on all (three) of their systems. Many of us make our living from software which, ultimately, we expect someone to pay for. We should extend the same courtesy to the vendors of products we use. Of course, I'm sure that none of you reading this would "borrow" software from a friend...

Software Development in C++

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

Kevlin provides a detailed response to Graham Jones' questions from issue 15 which I hope you will all find very enlightening, Richard Percy continues his series on financial application programming by revising and enhancing last issue's model and Francis warns of the perils of trying too hard to take Java on board as another "C with classes".

Some OOD answers

by Kevlin Henney

In "Some questions about OOD", *Overload* 15, Graham Jones called into question many aspects of the object view of software construction. There were some good questions, as well as a few misonceptions. Here I hope to address some of both.

As a first point, however, Graham opens his article stating that he has little experience of writing C++: "not really enough to have sensible questions". It has been said that there are no stupid questions, only stupid answers. I don't believe that you have to pass through some acolyte stage of avowed silence before reaching a priesthood level where you are allowed to ask questions. It is often difficult to get answers without asking questions, and harder still to ask deeper questions without previous answers.

Modularity

Graham identifies that the definition of the word 'modularity' varies between OO design authors. This is a good observation, but the variation has little to do with object-orientation. Before I was involved in OO development I was keen on structured approaches, but the one word that seemed to vary in definition between authors was — you guessed it — 'modularity'. Some use it to mean procedures and functions, others use it to mean compilation units, others use it to mean information cluster, etc.

OO inherits (sic) this confusion, to some degree, as it is essentially an extension of previously held wisdom on software engineering. Booch takes the term to mean something akin to Ada's package and MODULA-2's module, although the definition he gives is in fact broader and fits more comfortably with my own and others' view that a module is a cohesive unit of decomposition. Hence modularity is the property something exhibits if it is composed of such units. And in real terms what are these units? At one level you have classes, and within them member functions are also modular, and above classes you have compilation units (which may or may not have strong language support) and then subsystems.

In answer to one of Graham's queries, "I find it much easier to think of objects as more flexible implementations of the compilation units in C than as of C `struct`s. Is this sensible?", with the caveat that we replace 'objects' with 'classes' the answer is "yes".

For one of the clearest and most comprehensive discussions on the subject, I would recommend the chapter entitled "Modularity" in Bertrand Meyer's *Object-Oriented Software Construction* [Prentice Hall, 1988, ISBN 0 13 629031 0], as well as Parnas' seminal "On the Criteria to be used in Decomposing Systems into Modules" [*Comms of the ACM* 15(12): 1053-1058, December 1972].

Meyer's comment about matching "modules" to linguistic constructs is particularly telling in this context – Ed.

Abstraction

Whenever discussing the concept of abstraction I tend to wheel out a couple of quotes. I include them here, along with the requisite dictionary definition:

> ***abstraction*** n. ***2***. *the process of formulating generalized concepts by extracting common qualities from specific examples.*
>
> The Collins Concise Dictionary

> *The purpose of abstraction in programming is to separate behaviour from implementation.*
>
> Barbara Liskov, "Data Abstraction and Hierarchy",
>
> *OOPSLA '87 Addendum to the Proceedings*

> *In this connection it might be worthwhile to point out that the purpose of abstraction is* not *to be vague, but to create a new semantic level in which one can be absolutely precise.*
>
> Edsger W Dijkstra, "The Humble Programmer",
>
> *Comms of the ACM* 15(10), October 1972

Abstraction, as discussed above, is not a concept unique to OO, and is indeed a concept at the heart of any technique that purports to simplify the management of complexity, e.g., high level languages as opposed to assembler. However, it is the case that with object-orientation we can take the abstraction further:

> *As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behaviour. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.*
>
> Luca Cardelli and Peter Wegner, "On Understanding Types, Data Abstraction and Polymorphism",
>
> *Computing Surveys* 17(4), December 1985

I have given you a definition by reference of what I understand as 'abstraction'. There are other definitions, but that some authors prefer to be either more specific or more general is again not a problem with OOD; it is a fact of life.

Encapsulation

Normally there is not much variation in the definition of encapsulation. It is the concept that the abstraction is placed, literally, "within a capsule". The implication is that once the abstraction has been made, it can be closed and treated as a unit. This we express in terms of protecting state attributes as private data members and using public members to describe the meaningful operations on our abstract type. There is a strong analogy with security, considering the public/private separation akin to a firewall.

I have not read Russell Winder's *Developing C++ Software*, but the snippets that Graham quoted do not inspire great confidence. However, to answer one of Graham's points directly, "why hide things from the programmer?": because that is what modules do. If you program in C and use functions you already believe in hiding things from the programmer.

The idea of restricting access is in line with the principle that we separate behaviour, i.e., interface, from implementation. Some programmers are under the illusion that they have some kind of right to tamper with all the internals of any abstraction. This is fine if you're playing around, but it is not software engineering and has no place as a commercial attitude. In other words it may considered a privilege, but it is certainly not a right.

When I create an abstraction I describe the ways that it is meaningful and safe to use it. I create a working vocabulary for that concept. I use the language to enforce the semantics; I am not interested in having people break the code through accidental misunderstanding or simple fraud (in a language with a preprocessor and direct memory manipulation it is difficult to protect against determined fraud). You may differ in this: having people come up to you and tell you that your code is broken, only to discover that they have used it in a way that you did not anticipate, sanction or guard against, may appeal to you.

A common misconception is that lone programmers do not need to worry about encapsulation, or even meaningful variable names and clear code. But it's worth pointing out that

> *You learn to write as if to someone else, because next year you will be "someone else".*
>
> Kernighan and Plauger, *The Elements of Programming Style*

The other reason you hide implementation is to allow and anticipate change. Changing an open data structure in a program or, especially, a library results in more of a tsunami than a ripple effect. Being able to retake certain design decisions is an important capability. Consider a date class. Francis asked a similar question a while back ("Date with a design", *Overload* 11), but I am going to restrict it to handling Western calendars. What is the best internal representation for it? I can think of a few depending on the principal ways in which I intend to use it, but in all

cases the general interface is fairly stable. Answers and suggestions either directly to myself or to Sean.

Note that there is a concept of abstract or implied state that should be distinguished from the idea of physical implementation. A common misconception with OO is that not only is all data private, but there is no way to access anything of interest through a class interface. If this were true then it would create a great tension within any design: given a person object, how would one find out the spouse or parents if you believed that all you could return were primitive types and that a pointer would break encapsulation? The idea here is that we are modelling associations and collaborations that are required for an object to exist. We are modelling implied assembly, but there is no requirement that we implement it in a simple and literal fashion (the *Six Million Dollar Man* provides us with a good example of this).

Modelarity

Where are the objects? Graham says that his OCR application seems to be rather short on stable object abstractions. "Using OOA/OOD for the interface is fine, but a minor issue either way" seems to be a somewhat hasty dismissal. The majority of code in any application is related to presentation issues such as user interface, persistence management, comms, and error handling strategy within these. OO certainly works for these facets of an application, and even if it could not be applied elsewhere I would have said that it had already proved itself — an estimated 70 to 90% of application code is devoted to such handling!

But we can take it further. Steve Cook and John Daniels coined the term "object myth" to describe the common fallacy that objects in your application correspond to real world objects. Your application will look very odd if this is your view. What we are doing is modelling the problem domain which, from a requirements and analysis perspective, may be rooted in the real world. We should not mistake the map for the territory: the model has artefacts of the modelling domain that are not in the real world, and vice-versa. In turn we build a machine — a program being an abstract specification of a machine — based on the model. Again, there will be artefacts in the machine implementation that are not present in the model, and vice-versa. For a good discussion of this philosophy I wholeheartedly recommend Michael Jackson's *Software Requirements & Specifications — a lexicon of practice, principles and prejudices* [Addison-Wesley, 1995, ISBN 0 201 87712 0].

So really, the message here is that we are interested in our design having 'modelarity' rather than reality. A good example of this is when a friend came to me recently for advice on building a neural network and that he wanted to "do it with objects". (As an aside, his intent was to win on the horses. I have another friend who came to me a few years ago with the aim of doing a similar thing with the pools. Whilst I can help them with the modelling dimension, plausibility and realisation is firmly rooted in reality: I do not find myself swimming in the sea of alcohol that is my promised share of the fabled Big Win.)

I gave him a hand with the design as I had some experience with neural nets a few years ago (including a very simple OCR engine, as it happens) and "doing it with objects" is both an interest and what I do for a living. I am not going to discuss the way that he originally intended to handle the problem domain category (horse, jockey, etc.) to feature (win/lose) mapping in terms of neural net design, except to say that initially he was quite far off the mark and would have required a vast array of processors to achieve a solution in anything short of geological time.

So, how to model a neural net? The net design he had chosen was a fairly standard back-propagation configuration with one hidden layer. Such neural nets are often illustrated in terms of three layers of interconnected nodes, with a given weighting on each link. This is visually appealing and is an appropriate model for our understanding. It is also the one that he had chosen as the basis for his code. He had node objects and layer objects, and lists of pointers connecting the whole thing together. This is a singularly inefficient and redundant way of implementing a neural net: the wrong model was used — never confuse the aim of simplicity with something that is simplistic.

The mathematical model provides us with a more appropriate starting point: the nodes are of transitory interest, it is the connections that are doing the work, and the whole system is best described in terms of matrices and their manipulation. Thus matrices are the lowest level objects of interest. In this context I would strongly recommend Barton and Nackman's *Scientific and Engineering C++ — An Introduction with Advanced Tech-*

niques and Examples [Addison-Wesley, 1994, ISBN 0 201 53393 6]. If you do it thoroughly, there is more work here than simply creating 2D arrays. My friend initially looked a little disappointed: he had hoped there would be, I guess, some kind of magic going on; a big secret that was to be revealed to him.

I then pointed out that all we had was a handful of matrices that enabled us to build the foundation of his system, but we hadn't even touched the bulk of his application. We then provided a neural net class that encapsulated this state. The visual view is a convenient one, so the interface to this class gave the impression that its abstract state was composed of nodes and layers. This meant that it could be interacted with directly via a command or GUI system, which in turn was factored out. There was the issue of persistence, there was a lot of training data and the need to save and load pretrained or partially trained nets: RDBMS, flat text file, or binary file? Why not all? The best model for this is an object one; functions are a singularly inappropriate method for expressing this.

And what about training and running the net? Functions look like good candidates for these jobs, until you realise that they are uninterruptable, stateless, cannot be combined with other functions to create alternative training programs and are non-persistent. Why are these features important? A simple data flow approach, i.e., function maps input to output, is not adaptable, and is exclusively sequential, i.e., it will hang the application for very long periods of time. Reifying functions as objects (variously known as functors, function objects, functionals or functionoids) is not, as Graham suggests, about being trendy; it is because ordinary functions are simply very limited in C and C++.

If you have any familiarity with functional programming or lambda calculus you will immediately recognise other areas of limitation. If not, I would refer you to *Introduction to the Theory of Programming Languages* (principally chapter 5, "Lambda calculus") by Bertand Meyer [Prentice Hall, 1990, ISBN 0 13 498502 8], *The Emperor's New Mind* (the last section of chapter 2, "Algorithms and Turing Machines") by Roger Penrose [Vintage, 1989, ISBN 0 09 977170 5], or "Can Programming be Liberated from the Von Neumann Style? Functional Style and its Algebra of Programs" by John Backus [*Comms of the ACM* 21(8): 613-641, August 1978] for good introductions to this field. The Barton & Nackman book and Jim Coplien's *Advanced C++ Programming Styles and Idioms* [Addison-Wesley, 1992, ISBN 0 201 54855 0] are good reading for functor concepts and implementations.

By this time classes were practically dripping off the wall. I am not going to cover all of the candidate classes that we discussed, but it is worth pointing out that the majority of them were small helper classes that reified relationships, strategies, control patterns, views, etc. In other words, not these big, chunky, clunky real world classes that many OO novices (and, sadly, some 'experts') believe are the stuff of OO programs. The fine grained classes can have more of an impact on the construction of your application than the course ones.

I have skimmed quickly through the design decisions that we took to create a stable layered architecture that would tolerate the uncertainties and inevitable changes as the system was improved and refined, or different strategies were adopted. I could spend a lot longer on this problem if that was the focus of the article, and the fact that structuring the system in layers now allows the domain classes — horses, jockeys, races, etc. along with a reified mapping to nets — to be expressed simply and effectively.

Patterns

I was a little surprised that Graham said that *Design Patterns* [Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1994, ISBN 0 201 63361 2] seemed only to contain patterns for GUIs and none for scientific/engineering programming. Again, I think this view results from a failure to generalise. It is certainly true that many of the examples used are GUI based, but the examples are not themselves patterns. The use of many GUI examples in the book is related to the broader base that GUIs have as a simple currency over worked scientific/engineering examples.

I have applied or seen applied every one of the patterns described in the book, and I can attest that not one single pattern is GUI specific. Earlier this year the *Journal of Object-Oriented Programming* ran a series of articles about the application of some *Design Patterns* patterns to financial programming. John Vlissides has also been applying a number of these, in his *C++ Report* column, to the construction of a file system framework.

In my review of "Design Patterns" in Overload 9, I commented on how well-balanced I thought the examples were – Ed.

If you still doubt their applicability in scientific/engineering programming, I would again recommend Barton and Nackman's *Scientific and Engineering C++*. The book is not a patterns book, but you will find yourself tripping over common idioms and patterns in every chapter.

Conclusion

Graham's view of design is a good one:

1. *Decomposition into smaller, simpler pieces, and*
2. *Finding and exploiting similarities among the pieces.*

But it is not complete. This is more of a meta-design approach. The advice, as it stands, does not provide criteria for decomposition, judging the goodness of relationships between and within the pieces (coupling and cohesion), language mapping, testability, etc. Most of the design methods I have come across use these two principles, but if that is all they offered they would be pretty hollow — for some more reflections on this kind of advice see "Elements of Programming Style", *CVu* 6(6). Object-orientation provides us with a set of criteria and some mechanisms for decomposing programs that is based on sound software engineering principles.

I hope this article has shed more light than darkness, and perhaps has inspired further questions. I will leave you with another quote to ponder:

> *Substance doesn't change. Method contains no permanence. Substance relates to the form of the atom. Method relates to what the atom does. In technical composition a similar distinction exists between physical description and functional description. A complex assembly is best described first in terms of its substances: its subassemblies and parts. Then, next, it is described in terms of its methods: its functions as they occur in sequence. If you confuse physical and functional description, substance and method, you get all tangled up and so does the reader.*
>
> Robert M Pirsig, *Zen and the Art of Motorcycle Maintenance*

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

Go with the flow - part II

by Richard Percy

Recap

In the first article in this series I outlined the requirements for a generalised cashflow projection model and a prototype solution involving a *Cashflow* template class and its clients. The initial solution is a good starting point but requires some refinements and enhancements to meet the requirements listed at the outset. More testing is required in the context of a simulated system to verify that it is usable in a real program.

Requirements revisited

The initial model met the following important requirements:

- Generation of a cashflow of any type from a given start position for a specified number of periods or until a certain condition occurs, if earlier.
- The option to choose at run-time the function to generate each position.

However, there were a few minor problems with the *Cashflow* class' interface and memory management. It also stored all rows in a cashflow with no option to discard the rows that are not required. The C++ maxim, "Don't pay for what you don't use," suggested that I should develop the model to deal with optional storage of intermediate cashflow positions.

After some amendments and enhancements the *Cashflow* class needs to be taken for a spin over more demanding terrain. I have provided an example to test its speed and capacity and to demonstrate its use with two different client classes. These *capacity* and *capability* types of testing are important when developing a prototype that may be extensively used by other developers. Other users will often try to employ reusable components in situations never envisaged by their author; so it is as well to test for any limitations and publish them!

Revisions to the model

The model remains fundamentally unchanged and consists of a *Cashflow* template class and its clients. A client class represents a "row" of a cashflow and is required to have a member func-

tion that can populate its object's data using the data in the previous row.

The *Cashflow* class interface

The original class declaration was the following:

```
template <class Vec>
class Cashflow {

public:

  Cashflow(Vec* pStartPos);
  virtual ~Cashflow();

  // RollFunc type is a pointer to a
member
  // function of class Vec
  typedef bool (Vec::*RollFunc)
         (const unsigned long, const
Vec&);
  // generate the entire cashflow using
  // duration-limited roll forward
  void RollUpLim(RollFunc,
              const unsigned long
duration);

  virtual ostream& PrintOn(ostream& =
cout)

const;

private:
  // disable copy & assign
  Cashflow(const Cashflow&);
  const Cashflow& operator =(const

Cashflow&);
  // data members
  typedef TIArrayAsVector<Vec> CfArray;
  typedef TIArrayAsVectorIterator<Vec>

CfIterator;
  CfArray huge* pcf;
}; // Cashflow
```

The most significant problem with this interface was that the start position for the cashflow was supplied as a pointer in the constructor. This impaired the flexibility of the model because it prevented the start position from being changed between projections and, therefore, a new *Cashflow* object had to be created for each projection.

More subtly it gives rise to a memory issue because the internal container of the *Cashflow* class demands that its rows are created on the heap. This put the onus on the client code firstly to remember to allocate the start row using the `new` operator and secondly to remember to `delete` the row after destruction of the *Cashflow* object.

Consider my original code that created a start position, constructed a *Cashflow* object and ran the projection.

```
int main()
{
  int retCode;
  try
  {
    …
    Cashflow<TestVec> t(new TestVec(0.0,
             .008, 0.0, 50000.0, 50.0,
0));
    t.RollUpLim(&TestVec::ProjectionRF,

25*12);
    …
    retCode = 0;
  }
  catch (...)
  {
    cout << "\nException!\n\n"
            "Program threw an unhandled"
            " exception" << endl;
    retCode = 32767;
  }
  return retCode;
}
```

I stated that if an exception is thrown back to `main()` then all dynamically allocated objects are deallocated by the *Cashflow* destructor. This was fine, provided that we ever get to the destructor, but what if an exception is thrown in the *Cashflow* constructor while allocating memory for the internal container? If so then the constructor is wound back and execution jumps to the `catch(…)` block. The *Cashflow* destructor isn't called because the object has not been successfully constructed. Therefore, the start position (TestVec object) that has been created on the heap isn't deleted and a memory leak results.

There are two solutions to this. One is to leave it up to the client class programmer to use a smart pointer and the other is to accept an object passed by reference as the start position and make a copy to add to the internal container. I judged that the second is tidier and would have a negligible impact on performance. The resultant code is shown below and has other changes to deal with the optional storage of intermediate cashflow rows.

It is not normally recommended to change the interface of a reusable class but it is better at this early stage than later. As one says in financial circles, the past is not necessarily a good guide to the future!

```
template <class Vec>
class Cashflow {
public:
  Cashflow(const signed long
                              baseIndex =
0);
  virtual ~Cashflow();

  // RollFunc type is a pointer to a
member
  // function of class Vec
  typedef bool (Vec::*RollFunc)
                (const unsigned long,
Vec&);
  // generate the entire cashflow using
  // duration-limited roll forward
```

```
  void RollUpLim(Vec& start, RollFunc,
          const unsigned long duration,
          const bool storeAllRows =
false);

  virtual ostream& PrintOn(ostream& =
cout)

const;
  // Get index of start row
  const signed long BaseIndex();
  // Get current number of rows
  const unsigned long Rows();
  // Get current upper bound
  const signed long LastIndex();
  // Get a row
  const Vec& operator []
                   (const signed long
row);

private:
  // disable copy & assign
  Cashflow(const Cashflow&);
  const Cashflow& operator =(const

Cashflow&);
  // data members
  typedef TIArrayAsVector<Vec> CfArray;
  typedef TIArrayAsVectorIterator<Vec>

CfIterator;
  CfArray huge* pcf;
  const signed long base;
  unsigned long size;   /* Must maintain
       own size count because Borland
array
       allocates new rows in lumps
       (cfGrowth).*/
}; // Cashflow
```

There are a few points to note in the above declaration.

The constructor now accepts an argument so that the row numbering doesn't have to start from zero.

The roll forward function `RollUpLim` now accepts an argument to specify whether all rows of the cashflow are to be retained. If this is false then only the first and last rows are available.

An overloaded [] is provided along with some functions concerned with row indexing.

Copy construction and assignment are disabled. This is partly laziness on my part but I doubt that copying an entire cashflow would be necessary. As usual it depends on how loud the users shout!

Some people (mentioning no names) don't like `typedef` statements on the grounds that they impair readability. I suggest that they can be very useful for long or complicated type names.

I needed to implement the Borland array container using a `huge` pointer to prevent the test programs crashing with long cashflows. Perhaps there is an expert reader who could explain why this is the case.

Implementation of the *Cashflow* class.

The internal workings of the class are not altered radically from the description I gave in the last article. There are some changes because of the altered interface and, of course, some additional functions. I will describe only a few of the implementation details and would be happy to supply the full source code on request.

The constructor allocates memory for the internal array. It would be quite legitimate to do this in the constructor initialiser list like this:

```
template <class Vec>
Cashflow<Vec>::Cashflow(
  const signed long bi)
: base(bi), size(0),
  pcf (new CfArray(cfInitSize - 1, 0,
                   cfGrowth)) { … }
```

However, I might want to do something in the constructor that could throw an exception (perhaps if the value of the argument were outside a certain range). If this occurred after the memory allocation then the exception handling mechanism would not recover the memory and a leak would occur. Of course, this is only a serious problem if it happens repeatedly and the amount of memory is large. A safer way is the following.

```
template <class Vec>
Cashflow<Vec>::Cashflow(
  const signed long bi)
: base(bi), size(0)
{
  …leave memory allocation as late as

possible…
  // Use 0-based array internally for
  // convenience
  pcf = new CfArray(cfInitSize - 1, 0,
                    cfGrowth);
  // array "owns" elements
  pcf-
>OwnsElements(TShouldDelete::Delete);
}
```

It is this kind of nit-picking that makes C++ such a challenge!

The roll forward function is considerably more complex now that optional storage of intermediate rows is allowed but the basic mechanism is the same. The start position is now passed as an argument and a copy is made to form the first element of the array. Each further element is added and then populated using the `RollFunc` member function pointer supplied. This function takes the previous element as an argument.

```
template <class Vec> void
Cashflow<Vec>::RollUpLim(
```

```
  Vec& start,
  RollFunc pfRollUp,
  const unsigned long dur,
  const bool storeAllRows)
{
  …integrity checks on arguments…

  // delete array members & free memory
  if (0 != size) pcf->Flush();
  // make a COPY of the start vector
  pcf->Add(new Vec(start)) , size = 1;
  bool cont=true;
  unsigned long c = 0;
  Vec* pNewRow;

  if (storeAllRows)
  {
    while (cont && c < dur)
    {
      pcf->Add(pNewRow = new Vec());
      size++;
      cont = (pNewRow->*pfRollUp)
        (c + 1,

*(*pcf)[static_cast<int>(c)]);
      c++;
    }
  }
  else
  {…}
}
```

Note the use of the operator `->*` to call a class member function using its address.

The code shown above applies when the user requires all rows to be stored. I have not shown the more complex section that applies when only the first and last rows are retained. It involves creating temporary rows and overwriting them to minimise the amount of memory allocation performed.

The client classes

A client of the *Cashflow* class represents a row of a cashflow and must itself be a class. It must also have a default constructor, copy constructor, overloaded `==` and `<<` operators and at least one projection function whose address can be taken.

My example is intended to show how a personal pension quotation might be made. The details are simplified in order to minimise the size of the code but the overall design is realistic. Any similarity of my fictitious pricing basis to that of a real insurance company, either living or dead, is purely coincidental!

The main classes provided are *Policy* (an abstract class), *ULPension* (a unit-linked pension contract that also provides life cover) and *Annuity* (an annuity contract providing benefits payable throughout life from retirement). *ULPension* and *Annuity* each contain a nested class that is used to make up the cashflow rows.

The abbreviated declarations follow.

```
class Policy {
public:
  virtual ~Policy() {}  // ensure
correct
          // destruction of derived
objects
  virtual double GetCost() = 0;
                  // find price of
policy
  virtual void DoPIAProjection() = 0;
                  // print projected
values
};  // class Policy

class ULPension: public Policy {
public:
  ULPension(double sumAssured, double
fund)
  : polSA(sumAssured), targFund(fund) {}
  virtual double GetCost();
                        // calculate
premium
  virtual void DoPIAProjection();

  class RfVec {
  public:
    RfVec(double U=0, double G=0,
          double S=0, double P=0)
    :  uv(U), g(G), qx(0), sa(S),
       p(P), md(0) {}
    // default destructor, copy & assign
    // are OK
    bool IsEqual(const RfVec&) const;
    ostream& PrintOn(ostream& = cout)

const;
    bool KeyFeaturesRF(
            const unsigned long
newDuration,
            RfVec& oldRow);
    double GetUnitValue() const
    { return uv; }
  private:
    …cashflow row data members…
  };  // class ULPension::RfVec

private:
  …policy data members…
}; // class ULPension

class Annuity: public Policy {
public:
  Annuity(double annAmount)
  : polAnn(annAmount) {}
  virtual double GetCost();
                        // calculate
premium
  virtual void DoPIAProjection();

  class PriceVec {
  public:
    PriceVec(double F=0, double E=0,
             double G=0, double A=0)
    :  f(F), e(E), g(G), npv(0),
       a(A), res(0), ifp(1),
disc(1/1.01)
    {}
    // default destructor, copy & assign
    // are OK
…functions similar to ULPension…

  private:
    …data members…
  };  //  class Anuuity::PriceVec

private:
```

```
  …policy data members…

}; // class Annuity
```

The *Cashflow* objects are created in the functions `ULPension::GetCost`, `ULPension::DoPIAProjection` and `Annuity::GetCost`. The function `Annuity::DoPIAProjection`, however, doesn't need to create a *Cashflow* object. The `GetCost` functions don't store intermediate cashflow rows because only the last row is used to determine the cost of the policy. The `DoPIAProjection` function, however, needs to retain all rows for a diagnostic trace and to provide projected values at various points in time for the client's illustration. An example follows.

```
void ULPension::DoPIAProjection()
{
  ofstream

outFile("c:\\temp\\cashflow.txt",
                 ios::out | ios::app);
  outFile << "PENSION: Constructing"
             " projection cashflow..."
          << endl;
  RfVec startPol(0, 0.0094, polSA,
                   polPrem);
          // 12% p.a. growth for
projection
  Cashflow<ULPension::RfVec> cf;
  cf.RollUpLim(startPol,

ULPension::RfVec::KeyFeaturesRF,
         25*12, true); // age 35->60
  outFile << "...finished roll forward!"
          << endl;
  outFile << cf << endl;
}
```

Incidentally, it's not just "toy" programs that use streams to output data to files. They are of immense practical value in this type of application both to testers and users with technical knowledge.

The roll forward functions within the nested classes are implemented similarly to the extract published in the last article. The example below is for the pension because the annuity code is rather longer and nastier!

```
bool ULPension::RfVec::KeyFeaturesRF(
  const unsigned long newDuration,
   ULPension::RfVec& oldRow)
{
  // fill in missing values in old
period &
  // get unit value at start of new
period
  oldRow.qx = exp( newDuration/100.0 ) /

50000.0;
  oldRow.md = max(oldRow.qx * (oldRow.sa
-
     oldRow.p - max(oldRow.uv ,
0.0)),0.0);
  oldRow.um = oldRow.uv + oldRow.p -
oldRow.md;
  uv = oldRow.um * (1 + oldRow.g);

  // set up parameters for new period
  g = oldRow.g;
  sa = oldRow.sa;
  p = oldRow.p;

  return uv > 0; // stop if policy
lapses
}
```

Note that this function is called on a newly constructed object. The function merely populates some of its data members and the remainder of the previous row's members.

Controlling the application

The layers of encapsulation illustrated above result in a very simple interface and the controlling code can be kept short. Of course, the disadvantage of this, as with any encapsulated design, is that one must provide plenty of functionality at the interface.

```
int main()
{
  int retCode;
  try
  {
    Annuity ann(8500);
           // we require an annuity of
           // £8500 per month
    double fund(ann.GetCost());
    ULPension ulp(96000.0, fund);
    cout << "Monthly premium needed to"
            " produce fund: "
         << ulp.GetCost() << endl;
    ulp.DoPIAProjection();
    ann.DoPIAProjection();
    retCode = 0;
  }
  catch (xmsg x)
  {
    cout << "\nException!\n\n"
         << x.why() << endl;
    retCode = 32767;
  }
  catch (...)
  {
    cout << "\nException!\n\n"
            "Program threw an unhandled"
            " exception" << endl;
    retCode = 32767;
  }
  return retCode;
}
```

In the example above the 35 year-old client's current salary is £36,000. He needs life cover of £96,000 until retirement and he wishes to retire at age 60 with a pension worth £2,000 per month in today's terms. Assuming 6% RPI inflation this would be £8,500 at age 60. The program calculates the fund required at retirement as £823,000 and the monthly premium required to produce this fund as £775. The program runs rather

slowly, though, because of the iterative method used to achieve the target values.

Summary

The generalised cashflow model is now well developed and has been thoroughly road-tested. However, the imaginary users are now crying out for some formatting functions for their huge cashflows and an easy, efficient way to carry out iterative targeting ("back-solving") for pricing and simulation runs. The Technical Director might also be wondering why we're still using an out-of-date implementation of C++. I will address these points in future articles and provide a range of examples.

Richard Percy
106041.3073@compuserve.com

Mixing Java & C++

Ruminations by Francis Glassborow

One of the talks I attended at the recent Object Expo Europe caused me grave reservations. The speaker was very keen to elaborate on how you could mix Java and C++ by using the Java native method facility. At the end of the talk I expressed the opinion that the speaker had just spent 100 minutes describing my worst maintenance nightmare. I was only exaggerating slightly.

Code that seeks to mix legacy C with object-oriented C++ often has problems because procedural C makes assumptions that are not always well supported by object-oriented code. However the problems are not implicit in mixing C and C++ source. The designers of C++ have made a continuous effort to keep C++ compatible with C. There are incompatibilities because things such as nested classes are desirable in C++ and are not supported in C (nesting a struct, enum or union inside a C struct results in the enclosed definition being exported to global space). The C++ object model is largely backward compatible with the C one. C++ provides a mechanism for linking C code with C++ code, the extern "C" provision. Even here some refinement has had to be added because linking these is not as simple as it was originally thought to be.

Java provides a mechanism for using C native methods. Note that that is C not C++. I am sure the designers of Java were well aware of the mare's nest of problems that would open up if Java programs tried to call native C++ methods. Before I go on, I think it is worth mentioning that there is an immediate price for using native C from within Java; the result will fail the Internet security firewall provided by Java. In other words an Internet applet that contains a use of native C will have a very limited use.

Of course there is another cost for using a native C method in Java, the result is no longer portable.

Now let me move on to trying to use C++ from Java. The first question that springs to mind is why anyone would want to do this. The problem is that the Java world is expanding very fast (for example one of the most experienced Siemens' development teams – including such C++ experts as Uwe Steinmuller and pattern experts such as Hans Rohnert – has been using nothing but Java for almost a year). It is not only book publishers that want to rush to get on the bandwagon, the developers of libraries also want to get in ahead of the opposition. Those with reasonably well developed C++ libraries would like to get out an early release of the Java version. In addition there are all those developers who already use a C++ library and would like to reuse it in their Java developments.

The idea that has crossed the minds of several people is that they could provide a Java wrapper class to encapsulate the C++ class. If you just consider simple classes, this is relatively easy to do by hand and there appear to be a number of simple rules of thumb to guide you in doing it. Of course this leads people to write programs to automate the generation of such wrappers. In the typical test cases this works quite well. Unfortunately the test cases are ones where it would be pretty easy to rewrite the original code as a Java class.

Now those of you who are used to writing C++ for event driven environments know just how difficult it is to co-ordinate your program objects with their corresponding environment objects. In this case most of the design decisions were made with the intent that two sides should be able to co-operate. In the case of Java there were no such intentions. Java was not designed to co-operate with legacy C++. The two object models are entirely different. C++ is designed to support static binding wherever possible. Indeed the default for C++ is static binding and we have to tell the compiler when we want dynamic binding by qualifying member functions as `virtual`. Java defaults to dynamic binding and the programmer

has to explicitly qualify a method as `final` to enable static binding. This is not the place to discuss the relative advantages of the two approaches, each has advantages and each has penalties.

The next, and more drastic, difference is in the management of dynamic memory. C++, by default, leaves the responsibility entirely in the hands of the programmer. It does not prohibit garbage collection but it does not provide any special support for such. Java uses garbage collection to manage dynamic memory. There is no mechanism in Java to return the responsibility to the programmer, the best you can do is to summon the garbage collector yourself. That would normally be a very silly thing to do. Incidentally it was for just this reason – that garbage collection cannot be retroactively removed – that C++ chose not to support it by default.

This difference in memory management makes it very difficult to ensure that C++ objects and their corresponding Java wrapper objects die together. One of the quickest ways to create a dangling object in Java is to have two references to the same C++ object and use one of them to destroy the C++ object. Remember that Java has no concept of a reference: all Java class instances are handled through pointers! In Java:

```
Mytype mt;
```

creates a pointer for a `Mytype` object and initialises it to null.

```
mt = new Mytype;
```

creates an instance of `Mytype` and saves its address in `mt`. This is quite different from the C++ concept of an object and a reference. In C++ a reference behaves like the original because it is only, semantically, an alias for the original. Let me try to elucidate:

```
int fn(Mytype mt1, Mytype mt2) {
  mt1=mt2;
  return 0;
}
```

behaves in the same way in C, C++ and Java. In each case the local variable `mt1` takes on the value of the local variable `mt2`. In C a bitwise copy of `mt2` overwrites the contents of `mt1`. In C++ whatever copy assignment for Mytype changes the local `mt1` to `mt2`. In Java `mt1` and `mt2` will both now point to the same object (whatever `mt2` was pointing to). The important thing to grasp is that this code does not change the external objects that were passed as arguments. If you want to understand Java you must think of a function such as the above as being like C/C++:

```
int fn(Mytype * mt1, Mytype * mt2){
  mt1=mt2;
  return 0;
}
```

and not like the C++:

```
int fn(Mytype & mt1, Mytype & mt2){
  mt1=mt2;
  return 0;
}
```

which copies the object `mt2` refers to into the object that `mt1` refers to.

There are those such as John Max Skaller (an Australian C++ expert) who believe that C++ should have used such a model for variables, but it did not and we continue to live with both the benefits and the penalties for such choice.

How does all this matter? Well it means that those writing Java do not expect to have to do any memory management, indeed they cannot in the Java environment. On the other hand C++ code expects memory to be managed by the programmer. Do not think that `finalise` will pull you out of the hole. All that does is to specify some action that should be taken before the garbage collector recovers the memory for an object, but unless you force garbage collection, it may never happen. What this amounts to is that once you start splicing C++ code into your Java you will have to manage the C++ objects' lifetimes... all of them because Java only has dynamic objects (well that is near enough true). You have just lost one of the major features of Java.

If you think this is bad, much worse is yet to come. Java does not have multiple inheritance nor any of the consequential baggage such as virtual base classes. I have no doubt that with sufficient persistence and hacking skills you might be able to cope with C++ objects with multiply inherited parents, but frankly, why bother. The presenter of the talk on mixed Java and C++ programming provided a truly ghastly hack to enable the Java programmer to extract relevant layout information for a C++ object. It would be bad enough if this hack actually worked but in the presence of multiple inheritance and virtual base classes you have no reason to expect consistent layouts for different derived objects.

In my opinion, the long and the short of it is that whatever short term gains you make by using an automated tool to allow you to reuse C++ code in Java will be lost many times over in future maintenance and debugging.

I am happy with Java programmers using C native methods, just as C programmers sometimes use assembly language to access some platform specific resource. I am also happy with Java calling an entire C++ application – after all that is one of the advantages of using Java on a server: my client Java application can address a server application without having to consider the nature of the client platform, the client application can use platform specific programs to service the client requests. What leaves me deeply worried is that anyone could seriously propose that Java and C++ should be mixed in a single application – that way lies madness.

Now over to you. Tell me why I am wrong.

Francis Glassborow
francis@robinton.demon.co.uk

C++ Techniques

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

Alec Ross continues his exploration of techniques for type evolution and Francis contemplates the semantics of the C++ standard library.

Circles and Ellipses revisited: coding techniques – an introduction

by Alec Ross

A previous article [1] described some motivation for techniques supporting type evolution, as originally raised in a series of articles in *Overload* last year.

There are several mechanisms available in C++ to implement such type evolution.

First, an object's behaviour can be viewed as a set of sub-behaviours, and these can be seen as associated with sub-type states of the object, even if the object is a simple, conventional C++ instance with a constant type. In other words the change in type is in the eye of the beholder seeing "different" behaviours being exhibited by a conventional C++ object. For example a member function can be seen as being composed of sub-functions for different parts of its argument domain. As the argument value changes, these distinct functions are brought into play. The "different" functions can be seen as giving the object different methods, possibly depending on the values of member data. This technique is described in greater detail below.

It is also possible to overlay an object in store with one of a different type which can have a different interface, and, in general, different members. This technique could be seen as providing a "real" change in type. It is described in a subsequent article.

Finally, Coplien's Envelope-Letter idiom with simulated virtual constructors provides a mechanism to achieve most of what is wanted. [12] This also is described in detail in a following article.

Circle to ellipse - by using a change of perspective

At a basic level, any perceived variation in behaviour of an object which depends on something - such as the value of a data member - could be viewed as a a change in type. For example, a given member function could be viewed as being made up of two mappings from different parts of its argument's domain. (Partial functions).

e.g., (omitting inlines for simplicity):

```
class X { ... int f();  ... int n; ...};

int X::f()
{
  return (n % 2);  // sub-domains: n
odd,
}                  // n even
                   // f() is an isodd(n)
```

A change in the value of a member `n` changes `f()`'s return value: thus a change in `n` could be viewed as changing the type of the `X` object concerned.

The division of the argument domain might be arbitrary, but would most reasonably correspond to two sub-mappings with different expressions, e.g.,

```
 int X::f() { return ( n % 2 ? 1 : 0 );
}
                        // f() is an
isodd()
```

At this slightly more elaborate level, the mapping could be viewed as bringing two or more different behaviours into a single function, and the required execution path selected by an `if ( ) ...` or `switch ( ) ...`, a conditional operator (`? :`) as above, or possibly indexing into a table of pointers to functions. (Yet another mechanism would be to use conventional polymorphism with a function argument of polymorphic type.) Often member functions will be made up in such ways Ð but the aggregation and choice of different behaviours from within a single function is not usually perceived as a switch which changes the object's type. For example, for a (malleable) conic type, we could have:

```
void conic::f()
{
  if ( e == 0 ) circle_f();
  else if ( e < 1 ) ellipse_f();
  ...
}
```

Methods can also be changed dynamically on a per class or per object basis. When this change is expensive to make at run-time, when the aggregation of functionality results in relatively complicated code, where the change can be seen as persisting with an object or class over several subsequent uses, and when transitory event(s) trigger the change, then there may be a greater tendency to regard the change as a change in the type of the object rather than simply as a choice in behaviour (and data) within a given type. As noted above, the behaviour can be given a switchable indirection via a pointer to function, e.g.,

```
X::f() { p(); }
  // f() now invokes afunction
switchable
  // via p
  // (Note: "p()" is equivalent to
  //        "(*p)()" )
```

Paradoxically, idioms such as `foo() { ... if ( ... ) f1(); else f2() ... }` could be seen as late-binding, leaving the choice of behaviour as late as possible at run-time, whereas the use of a pointer to function could be seen as early/ier binding in some sense, where `p` is set and then (at lease potentially) left untouched for several invocations of foo().

The examples below illustrate some syntaxes which can be used to select a function implementation on a class or per object basis, using indirection from a pointer to a (member or non-member) function. This allows objects to switch their behaviour based on arbitrary criteria - such as the value of a data member, where any write access to this member could potentially change the pointer.

The following show the use of directly set member pointers to functions to achieve object polymorphism.

1) Generalised idiom/syntax to pick up free functions.

```
// idiom for choice of behaviour
// which can be regarded as
// involving a type-change
// (also appropriate for non-
morph)

void f();
void g();

typedef void (*PVF) ();

PVF p;

p = f;      // set p

foo() { p(); }
    // foo now always invokes
    // f() (until p changed)
```

2) Specific example.

```
// Illustration of use of pointer
to
// function members pointing to
free
// (i.e., non-member) functions

typedef void (*PVF) ();

class Conic
{
public:
  ...
  void Do()  { p(); }
             // invoke free
functions
  void Do_s()  { ps(); }
private:
  PVF p;               // per
object
  static PVF ps;       // class
wide
};
```

3) Use of pointers to member functions

```
// Illustration of use of pointer
to
// function members pointing to
// member functions

class Conic
{
public:
  Conic(double e = 0);
  virtual void Show();
  virtual void SetEccentricity
                       (double
e_in);
private:
```

```
  double e;
  void Setup();
  ...
  void (Conic::*ShowPtr) ();
  // pointer to member function
  // used to select display
function

  void ShowShape();
  void ShowCircle();
  void ShowEllipse();
};


Conic::Conic(double ei) : e(ei)
{
  Setup();
}

void Conic::Setup()
{
  if (e == 0)  // see note (*)
below
  {
    ShowPtr =
&(Conic::ShowCircle);
  }
  else
  if ( e < 1 ) // see note (*)
  {
    ShowPtr =
&(Conic::ShowEllipse);
  }
}  // Note (*): no allowance for
   // error margin

void Conic::Show()
{
  // will call appropriate fn
  Setup(); // ensure pointer
current
  (this->*ShowPtr) ();
                       // do the
deed
}
```

The pointer to function could be set-up in the constructor, and then maintained with all code which altered the eccentricity - or simply set up as needed. The above code illustrates both approaches, though one is sufficient. (The comment about the lack of error margin indicates potentially naive switching, where misclassification could occur due to errors in calculation or accuracy of representation. The code is thus oversimplified for the purposes of illustration.)

This example can even be stretched to use a member to point to a function member of another class.

4) Using pointers members which point to functions in another class:

```
class C
{
public:
  void f1()   { ... }
  void f2()   { ... }
};

class conic
{
friend class C;
public:
  ...
  ...
  void Show(C& c0)  { ((&c0)->*pc) (); }
private:
  typedef void (C::*PVFC)();
  PVFC pc;
  void setup()
  {
    pc = (e == 0.0) ? &C::f1 : &C::f2;
  }
  ...
  ...
};
```

An overall view

All of the above techniques have considered that an object could be regarded as having morphed when the behaviour of a given member function has changed. Obviously one can add techniques to change the sets of data members involved, such as the use of a handle-body idiom, with the data in appropriate bodies. Changing the accessibility of members is more problematic. The problem is perhaps not too severe for data, if it is at least protected, and possibly private as is normally the case - one simply does not need to change the accessibility. If one wanted to guard against function calls which were inappropriate for the object's state, one might have to resort to run-time tests, perhaps including the use of exceptions.

The approach described above has the merit of simplicity, though in some cases it could lead to objects bloated with data, complex function definitions, and fat interfaces. In other words, many of the benefits of OO - and in particular those due to encapsulation and inheritance - could be lost. Also, although the effective type of an object could be changed, the result of `typeid()` would be constant - which could be seen as a problem.

Alec R L Ross
alec@arlross.demon.co.uk

References

[1] Alec Ross, "Circles and Ellipses Revisited", *Overload*, Issue 15,

[2] James O. Coplien, "Advanced C++ Programming Styles and Idioms", Addison-Wesley, Reprinted with corrections 1992, especially Section 5.5, pp 133 ff (Envelope and Letter Classes),

pp 148ff, (virtual constructors using globally overloaded operator new), and Section 9.2, p 311 ff, (a canonical form for the Envelope-Letter idiom).

Why is the standard C++ library value based?

by Francis Glassborow

I was recently asked the title question in an email from someone who was struggling to use the RogueWave version of STL as shipped by Borland. I wonder how many of you have ever considered the question. I certainly had not, so let me make amends by sharing some thoughts with you. In doing so I am sticking my neck out and speculating because I have largely left library issues to those that specialise in that area – recent experiences suggest that this might not have been an entirely wise decision even if quite a few others took the same position.

I too made that decision and have similar misgivings – Ed.

I think that the first major issue is that C++ is not an object-oriented language. That is a very important issue because it strongly influences many design decisions. The Standard C++ Library is intended to be a tool for all C++ users, not just the object-oriented ones. It is very difficult to use an object-oriented library in a procedural or functional programming style. Some might even claim that it is impossible.

The second issue is that of efficiency. The Standard C++ Library is an inherent part of the language. Very few programmers are going to be writing code based only on the kernel language, other than the designers of the SC++L that is. This means that almost every other piece of code is going to depend on aspects of the design of the SC++L. We all know that the place for efficiency considerations is at this deep level. The foundations must be strong, efficient and suitable for use by almost everyone.

Object-oriented libraries are inherently less efficient. The reason is that there is a price to pay for late binding. At the higher levels of programming this is a price that we are often willing to pay. But not all programmers are in a position to accept the efficiency penalties that come with OOP.

It is worth noting that the most OO part of the SC++L is that related to i/o. There are already high overheads in implementing any support for i/o so the cost for making it OO is a relatively small element. Unfortunately the design of iostreams is highly complicated, involving twin hierarchies (the i/o class hierarchy itself and a hierarchy of buffers) as well as multiple inheritance and virtual base classes. Making sure that it was compatible with the C stdio specification as well as alternative characters (for example input might be from Unicode stream) has further complicated it. While it is technically possible to derive new polymorphic types of iostream objects doing so is definitely expert territory. I wonder, given a clean sheet, what we might design for i/o today.

There are many examples of object based design in the SC++L. By object based I refer to types that lack the polymorphic feature so that the code can be bound statically – that is, bound at compile time. Indeed one of the attractions of templates is that it helps with writing general methods without introducing an artificial type dependency. Compare the STL containers with earlier methods that relied on contained objects all being derived from a common base class (such as Borland's `TObject`).

You should consider the SC++L as a box of components, most of them are loose though a few are substantial constructs. Rather like my spare parts drawer that contains nuts, bolts, bits of wire, etc. as well as spare power units, plugs and cooling fans. I have to understand the higher order components so that I do not break a DC 12V fan by plugging it into a 230V AC supply. The low level bits are generally more robust and I can see what they do. Of course if I insist, I can strip the threads on a bolt by using the wrong nut.

It is usually possible to encapsulate a non-OO item in a wrapper. For example, suppose that we have a simple class:

```
class example {
public:
  example& fn ();
  // rest of interface
private:
  // interface
};
```

Now I can write:

```
class oo_example : example {
  virtual oo_example& fn()
  { return example::fn(); }
  // provide the rest of the interface
  virtual ~oo_example();
private:
  // probably nothing
```

```
};
```

I am not sure whether private inheritance is appropriate here rather than using the 'Cheshire Cat' mechanism:

```
class oo_example1 {
  virtual oo_example& fn()
  { return smile->fn(); }
  // provide the rest of the interface
  virtual ~oo_example();
private:
  example * smile;
};
```

I would welcome your thoughts on the issue. I guess there are times when one is better than the other and perhaps some of you can come up with good examples so that we can all learn to make better choices.

Conclusion

Avoid trying to make the SC++L into something that it is not. There are lots of examples being thrown around at the moment aimed at demonstrating how powerful the STL is. The problem is that these examples often make the STL seem to be something that it is not. It is possible to use raw STL for small programs but it is really a set of simple components from which you can develop domain specific tools. The application programmer will often be at least one more layer away from the underlying procedural code.

I am reminded of my days as a Forth programmer where everything was built on what went before. All that you needed to implement for a new platform was the native code primitives, everything else would port easily. If you view the Standard C++ Library plus the C++ kernel as the native code primitives then the programs built on them will port easily. Of course what we need are some other special standard libraries such as a graphics API so that other parts of our programs can be built with standard components. Of course that will require other abstractions, or rather extensions to the C++ abstract machine.

Which reminds me, one of the strengths of the Java programming language is that it is written for the Java Virtual Machine, if you start using Java divorced from the JVM you will find that you no longer have quite such a portable language. Those seduced into using J++ as an ordinary programming language would do well to remember that.

Francis Glassborow
francis@robinton.demon.co.uk

editor << letters;

Hi Sean,

Great issue, *Overload 15*. In the /tmp/late/* column, "Constraining template value parameters", I illustrated an interval checking class that could be used as follows:

```
static in_range<id, 0, max_id>
id_in_range;
```

This asserts that the first parameter is in the inclusive range defined by the following two, i.e., in the range [minimum, maximum]. You commented

> *Perhaps, given STL's practice for intervals [minimum, maximum) might be more in the spirit of C++?*

The reason I did not exclude the upper bound is the loss of domain: I would never be able to check that something was in the range that included INT_MAX, so as a complete range checker the class would not have been fit for purpose.

Now on to a genuine erratum. I wrote the specialisation

```
struct compile_assert<0> {
compile_assert(); };
```

And commented

> *... for 0 the constructor is inaccessible, and hence objects of this type are undeclarable.*

Which is pretty much nonsense, I'm afraid. It's certainly true that the code wouldn't build to completion as no constructor definition is provided – the linker will cheerfully inform you of this, but that was not my intent. What is missing is a private access specifier, or use of the word class rather than struct. Either will have the desired effect, but I tend to use struct for this and related concepts such as traits, preferring instead to use class for more conventional OO and ADT definitions.

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

> *Your comment about ranges is a good point! As for the "private" constructor,*

well I misread it as you had intended too...

Sean,

In his discussion of the Boolean type in *Overload 15*, Francis states that they should not support any operators other than (in)equality, assignment and inversion. What about the logical AND and OR operators (`&&` and `||`)? These are absolutely essential for compound conditions, like

```
if (a==1 || b==2)
```

This brings me on to a point that I have been meaning to write to you about for some time. Francis also suggests a user-defined type can support a Boolean type - meaning, I think, that the restricted number of operators available can be ensured. However, the two logical operators `&&` and `||` acting on built-in types have special behaviour, insomuch as they evaluate their operands conservatively, and this is not reproduced in the class equivalents as far as I am aware. Take this simple program:

```
typedef int Bool;
Bool cmp_fn(int a, int b)
{
  cout << "Compare " << a << " & " << b
       << "\n";
  return 0;
}

main()
{
  if (cmp_fn(1,2) && cmp_fn(3,4))
    cout << "EEK!\n";
  else
    cout << "OK\n";
}
```

The output is:

```
Compare 1 & 2
OK
```

Now replace the `typedef` with the class:

```
class Bool
{
public:
  Bool(const int){};
  ~Bool(){};
  operator int() { return 0; };
  Bool operator &&(Bool&)
  { return Bool(0); };
};
```

The output is now:

```
Compare 1 & 2
Compare 3 & 4
OK
```

So we see that with a built-in type/operator the second comparison is optimised away, but this cannot occur in the user-defined version because both operands to the `operator &&()` function need to be evaluated before it is called.

This may seem somewhat esoteric, but there are instances where a Boolean-like type is required, consider fuzzy logic for one, and having such a type behaving differently from the built-in that it is modelled on is undesirable. The case of the `++` operator, which used not to distinguish the pre- and post- versions for user types is a case where the language changed to be more consistent. I wonder why the same consideration has not been given the the logical operators.

Colin Hersom
colin@hedgehog.cix.co.uk

Why not omit `Bool::operator&&()`? It isn't needed since you have a conversion from `Bool` to `int`. However, you are right that there is no way to simulate the McCarthy-`&&` semantics in C++ (without resorting to all sorts of expression classes to provide delayed evaluation semantics!). I believe the possibly of "fixing" `operator&&()` has been discussed but it would require extensive changes since at present the rules for function calls cover `operator&&()`.

This is the second part of a letter sent to me as editor of CVu. As it concerns material published in Overload, I have extracted it and added my response – Francis

Dear Francis,

In *Overload 15* Kevlin Henney describes some interesting ways of using templates for compile-time tests. In C, you can use the pre-processor, or runtime tests, both of which have disadvantages. But what about using `enum`, as in the following?

```
enum { ensureintsare32bits = 1 /
            (sizeof(int)*CHAR_BIT == 32)
};
```

Changing the subject again (and no longer writing for *CVu*), it seems to me that your code for `setname()` (*Overload 13*, page 7) is still unsafe. I wrote to *Overload* about this, but I was a bit confused, and I managed to confuse Sean too. I was confused because my (very out of date) version of `strrchr()` returns a non-const `char*`, and because `setname()` is declared as

having a non-`const char*` argument. While correct use of `const` prevents the second problem I illustrated, it is not much help for the first.

I presume you would like `setname()` to have a `const char*` argument. You can then cause `setname()` to fail like

```
record.setname( 1 +
          strrchr(record.getname(), ' ')
);
```

I am unclear how general this sort of problem is due to my lack of knowledge about C++. But it seems to me that whenever a part of an object can be understood as a whole object there might be problems, e.g., if I wanted to take the real part of a complex number, (but still regard it as a complex number) I might use

```
Complex a;
...
a = a.real();
```

If the class Complex uses dynamic memory, and its assignment operator follows the usual

```
if (this != &rhs)
 {destroy, create}
```

pattern, is there a problem? I am unclear as to whether this is exactly what you were getting at in *Overload 13*, or just that the code for doing it should be in one place. Why is a "create, copy, destroy" pattern not the norm, as in the following?

```
void setname(const char *s)
{
  char *p;
  p = new char[strlen(s) + 1];
  strcpy(p, s);
  delete [] name, name = p;
}
```

You could pass this on to Sean, or incorporate any of it in a reply, as you see fit.

Regards

Graham Jones

Francis responds:

Graham raises a number of interesting points and illustrates the danger of some of the current idioms being used in C++. Most objects cannot be created from parts of themselves. However the standard C++ idiom assumes that this will be the case. Graham demonstrates that this assumption is unsafe.

Many of the idioms of C++ are intended to provide efficient low-level code. This is important so that the high-level programmer does not pay the price for reusing low-level code. Elsewhere in this issue I examine another aspect of this problem.

The low-level pattern Graham suggests is certainly reasonable (and preferable) for functions such as `setname()` where it will be rare that the original is already the required new version. It is also more efficient because it replaces a decision with a local pointer variable (small space use, no extra instructions) and a single assignment to a pointer. It will be less efficient when it unnecessarily creates a new copy. Note that dynamic memory allocation is potentially a very expensive process, many programs spend over 30% of their time in dynamic memory allocation/deallocation.

Where the copy creation is more frequently unnecessary then the initial if statement may still be desirable. Where creation from a part object is impossible the standard idiom will do, but again where creation is rarely unnecessary Graham's alternative becomes a front runner again.

Francis Glassborow
francis@robinton.demon.co.uk

> *You didn't confuse me Graham, but your question was rather vague! I assumed your problem was const-correctness because `strchr` is different in C & C++. The other problem to which you alluded passed me by because I would use { create copy, destroy } for strings anyway (rather than { destroy, create }) or copy-overwrite if I had a safe way to do it, e.g., `memmove`.*

Sean,

Graham Jones' article makes depressing reading. I don't know how OCR systems work, but I have found that an object-oriented approach always leads to a better design. I accept that there may be cases where it is inappropriate (the Standard Template Library has been quoted as one example), but I am sure these are rare. I get the impression that Graham is entirely self-taught. If so, I would suggest that he tries to find a seasoned practitioner of OOD to help with his program. There is something important in OO methods, it does work in practice and I would hate to think that *Overload* readers might be put off by articles such as this.

And so to Peter Moffat's linked lists... When I first started reading about object-oriented pro-

gramming it was frequently said that "you'll never write another list". Lists (and similar basic constructs) would be standard library objects. So, when I got my first C++ compiler, I tried to write a List class to see if I had grasped the concepts. And, like Peter, I spent many hours trying various different strategies. I was never really happy with the results. Some versions were easy to use but expensive on space or CPU time, others were efficient in space or time but cumbersome to use. Of course, this was before I had heard of templates and long before the Standard Template Library.

I think the most important point about Peter's article is that there will be many, many programmers out there in commerce and industry following similar paths through the C++ jungle. Most will get there in the end, but it will be a long and expensive journey for those setting out without a guide. Some won't make it at all. I have no doubt that there will be a backlash against C++ (and perhaps OO in general) in the commercial world when the first wave of projects is over and companies look back at what has been achieved. There will be a lot of legacy C++ code - objects everywhere, but all inextricably tied to each other so that changing one has a knock-on effect on far too many others. The trouble is it takes a long time to learn how to use C++ properly and commercial organisations don't have that time. Software engineering is still a very young discipline and growing up is a painful experience.

Circles and Ellipses... Alec Ross's article has some intriguing ideas. I can't wait for the follow up articles.

My "suspected bug in Visual C++ V4.1" was confirmed as such by Microsoft. I suspect it's a problem with namespaces (in that case, a class as a namespace) because we found another one that might be related. It manifests itself in code like this:

```
// Suspected bug in MS Visual C++ V4.1.
// Watcom 10.6 reports no error.

template <class T> class set;

struct Junk
{
  void set();
};

void Junk::set() {} // error C2955:
'set' : class template name expecting
parameter list
```

The error message goes away if the member function is defined within the class. This has also been confirmed as a bug in VC++ 4.1 and 4.2 by Microsoft.

> *I'm astonished that a compiler could get this wrong! I wonder what on Earth they're doing here?*

Phil continues:

Stop Press: I think we've found another bug in VC++...

```
    class Outer {
    protected:
        int i;
    public:
        friend class Inner;
        class Inner {
        public:
            void f (Outer& outer)
            {
                outer.i = 3;    // error
C2248: 'i' : cannot access protected
member declared in class 'Outer'
            }
        };
    };
```

There is a similar bug in the Knowledge Base (Q115854) which "has been corrected in Visual C++ version 2.0".

Phil Bass
pbass@rank-taylor-hobson.co.uk

I replied:

> *Nope. By chance it's correct. The friend declares (injects) a name at file scope which is therefore not the same as the nested class. Try adding "class Inner;" ahead of the friend declaration – that should forward declare the nested class and then the friend declaration will refer to the nested class. I suspect VC++ will still choke on it.*
>
> *Note that the committee have recently removed name injection from the language but I'm not sure of the impact on this construct.*

Phil responds:

My apologies to Microsoft, VC++ gets it right and your suggestion works just fine. Usually I check suspected bugs against another compiler, but I was a bit hasty this time. Worse still, I didn't really understand how friend declarations work. Just shows how valuable membership of ACCU is.

I must credit Stuart McGregor, a contractor working with us, for discovering all three bugs reported to ACCU recently. In each case, it was code using or in the style of the Standard Template Library that tripped us up.

Phil Bass
pbass@rank-taylor-hobson.co.uk

Which just goes to show how much STL stresses current compilers!

Hi Sean,

I was told that in C++ the `switch` statement is a no-no. But in *C++ The Complete Reference* Second Edition by Herbert Schildt (ISBN 0-07-882123-1) it says:

"Virtual functions and dynamic binding enable polymorphic programming as opposed to `switch` logic programming. C++ optimizing compilers normally generate code that runs at least as efficiently as hand-coded `switch`-based logic."

And that was a *Performance Tip*. To me that says that it's still ok to use switches and that they are just as efficient as using polymorphic programming (Which I don't know how to do as of yet) but if polymorphic is more efficient then I think I'm going to learn it ASAP. What are your thought about this?

Thanks again...

Steve Mertz
smertz@direct.ca

Steve is a novice C++ programmer who has been emailing me quite a bit. Originally he had problems with a (poorly designed) string class and I helped him work through that. Eventually he said it was from a book and then went on to the above issue from the same book. I have clarified the issues for Steve in private email but thought I would bring this to the attention of the readership as a warning: many of the books out there are written by authors who do not understand C++! Schildt's annotated ANSI C Standard book is (in)famous for having nonsense commentary but it's one saving grace is that it's the cheapest way to obtain the ANSI C Standard (even though one page of the description of printf is missing!). The string class was full of simple mistakes which will unfortunately go over the heads of most novices - such mistakes are doubly dangerous for they mislead the very people they should be helping.

And what are my thoughts?Don't buy any of Schildt's books!

`switch` has it's place in C++ because it and polymorphic method calls do not do the same job and each is suited to solving different problems.

Sean,

I have recently discovered your "C++ – beyond the ARM" page, and have found it very helpful. Thank you for making it available.

However, I have a question I have not seen directly addressed. It relates to a class declaration nested within a template class, e.g.,

```
template< class T >
class enclosing {
private:
  class inner;
}; // enclosing<T>
```

It seems clear that such a nested declaration is allowed. It is less clear, however, how then to define the inner class. (I know that I could place the definition entirely within the scope of the enclosing class, but choose not to do so because of the complexity of the inner class.)

By analogy with the syntax used to define member functions, I would guess the following approach:

```
template< class T >
class enclosing<T>::inner {
}; // enclosing<T>::inner
```

However, I have found no compiler that likes this at all, nor any of several variations on this basic theme.

If you have a moment, I would appreciate your comments on this puzzle.

Thank you.

Walter Brown
wb@fncrd8.fnal.gov

Glad you found my pages useful. I've recently updated much of the template information.

You are absolutely correct about defining the nested class: forward declaration of nested classes – even without tem-

plates thrown in – is a relatively recent resolution of the committee and many compilers simply haven't caught up yet (MS VC++4 doesn't even get the non-template case right).

questions->answers

by Kevlin Henney

After a rather unexpected and unplanned break, *questions->answers* is back. Fortunately, the break was not forced through lack of questions — or even lack of answers — so please continue forwarding any C++ and OO questions either to myself or via Sean.

Access, your flexible friend

Peter Pilgrim asks how it is possible to create a class that may only be allocated with `new`. This question has come up a couple of times in different places, including recently on `accu.general`. As with many questions, the answer to this one mines a rich seam of techniques based on a single principle. The principle here is to control — restrict in fact — the access of default features. Along with the explicit member functions you provide for your class the language provides you with the implicit capability to do the following:

- *Default construction*, if no other constructor is provided and if none of the non-`static` data members are references or `const`;
- *Copy construction*;
- *Copy assignment*, providing that none of the non-`static` data members are references or `const`;
- *Destruction*;
- *Taking the address* of an object of that class using `operator&`;
- *Dynamic allocation* using the global `new` operator;
- *Deallocation* using the global `delete` operator;
- *Dynamic allocation of arrays* using the global `new[]` operator;
- *Deallocation of arrays* using the global `delete[]` operator.

There is an additional rule that for derived classes these operations are not implicit if they were not accessible in the base class. The list above forms what I sometimes call the hidden interface and it is worth keeping in mind that the way to prevent any of these operations is to declare them explicitly as `private` and not provide a definition:

```
class mutex
{
    ...
private:
    mutex(const mutex&);
    mutex& operator=(const mutex&);
};
```

This fragment illustrates the common technique of preventing default copy behaviour in a class. Anyone attempting to construct a copy, such as passing by value, or assign one object to another will be greeted with a compile time error telling them that those members are not publicly accessible. This is much better than being greeted with an obscure runtime error that comes from resource aliasing and accidental multiple release. Use this idiom when copying behaviour is

- meaningless (e.g., copying a container of raw pointers that owns, i.e., will delete on destruction, its contents),
- not something you want to provide in casual syntax because of its hidden overhead (e.g., duplicating a file object), or
- simply too difficult and not felt to be necessary (e.g., duplicating large data structures of which only a few instances ever exist in a program).

On the issue of inheritance it is possible to capture this idea in a base class:

```
class nocopy
{
private:
    nocopy(const nocopy&);
    nocopy& operator=(const nocopy&);
};
```

This confers non-copying on its derived classes as their default:

```
class mutex : public nocopy
{
    ...
};
```

This particular mix-in style potentially has no space overhead as the base class is empty, and can be optimised out of the size of the derived class (see *questions->answers*, *Overload* 13). On high warning levels some compilers or checkers may issue a warning that the copy operators cannot be generated for derived classes. Since this is more often the intention than not (in truth, I don't seem to be able to recall any circumstance in which it has not been expected) such warnings are a pain and potentially deter programmers from using a sound technique.

But watch out for programmers who then try to pass objects round by `const nocopy&`! Consider the implications of making `nocopy` (a) a private base class (b) a virtual base class. What is the impact? – Ed.

This is all well and good, but how does it help with Peter's original problem? I've reviewed a common example and asserted that the principle is general. But what do I declare `private` to restrict creation to `new`? This requires a little lateral thought. Consider the following:

```
void dummy()
{
    dialog_box unused;
}
```

This function declares an automatic variable. What is being executed? A constructor and a destructor. Clearly, to prevent such declarations (and objects with static storage duration), one or both of these must be made `private`. If we make the constructors `private` we will prevent the following expression from compiling:

```
dialog_box *q_and_a = new dialog_box;
```

Which is unfortunate as it is the only one we wish would compile! In other words, declare the destructor `private` and the previous expression will compile, as we intended, and the `dummy` function will not.

Suicide objects

So that's it is it? Well, not exactly. How do I get rid of such objects? I can't use `delete` as the destructor is `private`. The incorrect answer is "just leave 'em, the runtime'll pick 'em up when it's finished" and is neither necessarily true — users of 16 bit Windows can spend time amusing themselves watching the resources in their system leak away every time they run Word or Excel — nor safe — they will fail to release cleanly any system resources they grabbed, and these may be more significant than memory.

Some objects govern their own lifetimes, i.e., the stack, static storage or another object are not responsible for destroying them. They will disappear of their own accord, based on some event:

```
void dialog_box::on_completion()
{
  delete this; // be afraid, be very
afraid
}
```

The `delete this` is the reason that you wish to prevent non-heap creation. You must also be sure that no other object holds a dangling reference to the object. As suggested, this is potentially dangerous code and always deserves a comment. You may wish to chose something slightly less flippant, but I will confess to having put this comment and similar into production code — it has the desired effect of grabbing your attention and being memorable!

The modeless dialog box is the most common example of the need for self determination; you also see it with daemon threads and similar examples. In making the destructor non-`public`, as opposed to the functions in most other access limiting techniques, you need to provide a definition of the destructor. You may wish to make this `protected` if you have a hierarchy of suicide objects in mind. Note that making the destructor in a derived class non-`public` once it has already been made public in the base is next to useless — *next to*, but not *exactly*, as there are a couple of techniques that I may discuss in a future column (if asked) that work on this principle.

There are other techniques that do not play with fire and constrain the semantics of the class so heavily, involving the addition of manager objects to govern knowledge and lifetimes of other objects. Consider a document view window that kicks off a modeless search dialog box. When the user is done with it they dismiss the dialog. In the `delete this` scheme it commits suicide and that's it. A better scheme is that the document view remembers the dialog box. If it has disappeared from view and the user asks again for a dialog box, the same one is pulled forward rather than another one created. Similarly, on dismissal the dialog box informs its owner that it has been dismissed and the owner takes appropriate action: either it deletes it (sort of a callback for self deletion), or it hides it and retains it for the next time it is used, so that the cost of window creation is not paid again and the dialog's state persists. This design is both cleaner

and more flexible; the fun is at the design level rather than at the language feature level.

Threadbare

A question from Jon Jagger asks how to create thread objects given a C function based threading API. There are a number of threading APIs that share generally the same characteristics. In addition to a number of flags and other control data, there are four key components to a thread creation function's prototype:

- A function pointer that is executed as the newly created thread's main function;
- Some user data, normally a `void *`, that is passed into the thread function;
- A handle returned to identify the handle created;
- An indication of success that is returned to verify that the handle was successfully created.

Often the last two are rolled into a single return value, but this is not always the case: e.g., both the Win32 and POSIX.1c thread creation functions separate them out. Here is the prototype for the POSIX function:

```
int pthread_create(
  pthread_t*thread_id,
  pthread_attr_t
            creation_attributes,
  void*     (*start_function)(void *),
  void*     start_function_argument);
```

All this is fine for C: we create a function that is to be executed as the thread's lifecycle, and pass in some data that we want the thread to operate on or use as context. A little bit of casting here and there for tidiness sake, and everything seems OK:

```
void *thread_main(void *argument)
{
  thread_data *to_do =
                (thread_data *)
argument;
  ...
}

int main()
{
  ...
  if(pthread_create(&tid, attr,
                thread_main, &data) !=
-1)
    ...
}
```

But this is clumsy and error prone, and certainly some way removed from the object model that we would like to be using consistently within our C++ code. There are a couple of issues here:

- We would like a more toolkit base approach in a thread class, i.e., that it wraps up the complexity of initialisation, parameters, launch, failure, etc.
- The function based approach shows high coupling between arguments and weak cohesion with the function.

The second point is a software engineering issue, and is equivalent to saying that the arguments to our function are not normalised. You may not have thought of applying the concepts of coupling, cohesion and normalisation to function signatures — I'm happy to explain this in more detail if prompted. Effectively the function and the user data are a single unit and should be expressed as such. The way that we resolve this is to invert the relationship between the function and its data, defining a class that contains the data, has a function defining the lifecycle, and hides the plumbing of the C API:

```
class thread
{
public: // usage interface
    void run(); // kicks off thread
    ...
protected: // lifecycle
    virtual void main() = 0;
private: // plumbing
    ...
};

class needle : public thread
{
    ...
protected:
    virtual void main();
private: // data for use by thread
    ...
};

needle hey_stack;
hey_stack.run();
```

I will say that there are a couple of things I do differently, but those would probably raise more questions than answers! I will ask one question that you might like to ponder: why don't I automatically run the thread when I construct it?

So far so good, but `main` is a member function and the thread creation function is expecting an ordinary function. Some people try all manner of casts to get this to work, but it isn't worth the trouble: it's not supposed to work. There's a lot more to member functions — member function pointers, in particular — than meets the eye. There is a mistaken belief that `this` is passed in as a hidden first argument to a member function. If this were true then there might be some reason to believe — ignoring the not insignificant `virtual` for the moment — that the member func-

tion pointer technique would stand a chance of working. However, the truth is that it is *as if* `this` were passed in as a hidden first argument: there are few compilers now that implement it like this.

I won't dwell on member function pointer issues:

1. There is probably a whole (at least) `questions->answers` in that area, and
2. We don't need them to solve the problem.

By turning the problem inside out again we get our solution:

```
class thread
{
    ...
private:
    static void *runner(void *);
};
```

A `static` member function has no `this` pointer and is, to all intents and purposes, a global function so we can cheerfully pass it to our thread creation function without worry[2]. What is the data item of interest? The current object:

```
void thread::run()
{
  ...
  if(pthread(&id, attr,
              runner, this) != -1)
    ...
}
```

To get the behaviour we desire we perform a simple piece of unpacking in the `runner` function:

```
void *thread::runner(void *data)
{
  thread* self =
         reinterpret_cast<thread
*>(data);
  self->main();
  ...
}
```

I have used the newer cast notation as it is cleaner and clearer in intent — you try grepping for parentheses in your program to try and find casts, and see how far you get!

If you have not come across this technique before, follow it through carefully and you will see how we achieve type safe and type dependent behaviour in a simple form. The adaptor function allows us to use a C function to build an OO framework — polymorphism, encapsulation, and all. I have glossed over a couple of issues to get to the point, but I hope that there will be some questions about these in future.

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

[2] Well, sort of without worry. Few things in life are easy and, I'm afraid to say, there is another issue here that I am going to gloss over. Sean, how about a whole *questions->answers* issue of *Overload*? Err, actually no, I didn't say that — one must be careful what one asks for; one might get it.

News & Product Releases

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

The OMT User Group

a correction from Kevlin Henney

The mailer that I used to send the OMT User Group announcement was a little generous in reinterpreting currency symbols: all the pound signs were converted to dollar symbols. This resulted in membership details for the user group being quoted at much less than cost price in the last issue of *Overload*!

Corrected, excluding VAT one year's membership is £39 for an individual, corporate membership is £129 for 5 named individuals or £199 for 10.

Additionally, a web site is now being planned and corporate members will be entitled to a link to their site, where applicable. For details please contact one of:

Kevlin Henney
khenney@qatraining.com

Jan Bevans
jbevans@qatraining.com

Note that these new email addresses supercede the older (qatrain.mhs.compuserve.com) ones which remain compatible, but use a slower and less reliable connection method.

> *I should apologise to Kevlin for not picking up on this! Perhaps I have become so used to American prices with spending so much time over there...*

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

My C++ pages. The template section has had a major overhaul recently.

```
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

Managing Editor

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

Production Editor

Alan Lenton
alenton@aol.com

Advertising

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

Subscriptions

Barry Dorrans
2, Gladstone Avenue
Chester, Cheshire, CH1 4JU
barryd@phonelink.com

Distribution

Mark Radford
mark@twonine.demon.co.uk

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 17* (December/January) should be submitted to the editor by December 7th.