ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 10

October 1995

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

Subscriptions:
Membership Secretary
c/o 11 Foxhill Road
Reading
Berks
RG1 5QS
pippa@octopull.demon.co.uk

£3.50

Contents

Editorial

A history lesson

In 1993, there was the Turbo C++ SIG founded by Mike Toms and from it came *Overload 1*. By the middle of the year the SIG had absorbed the European C++ User Group, causing a five-fold increase in membership and thus losing the compiler-specific nature with which it was conceived. For the first three issues, Mike was pretty much single-handedly responsible for writing the bulk of the articles that appeared in *Overload*. Issue 4 saw a shift with many different contributors beginning to appear.

Issue 4 also saw Mike appeal for writers on a wide range of subjects. An appeal that has largely remained unsatisfied. I also note, somewhat distressingly, that the number of current contributors is in danger of settling back down to a handful of dedicated regulars – as noted by Francis at the end of *CVu7.6* – so I shall republish Mike's "call for articles" in the hope of spurring some of you into writing for future issues. You don't have to be a C++ expert – we need the novice view too! You don't have to be a great writer – *Overload* is about C++ rather than literature. See if anything on the following list takes your fancy:

1. commercial experiences of C++
2. streams
3. exception handling
4. templates
5. memory management
6. RTTI
7. STL
8. OOA/OOD
9. troublesome keywords or other language 'corners' (e.g., **const**, **volatile**, **static**)
10. C++ development tools

So get writing! Or I'll send the boys round... Remember, we know where you live **:-)**

Submissions

Nearly all the submissions for *Overload* arrive by email now. Most arrive as plain text which is my preferred format. If you send me *Word 2.0 for Windows* files, I print them, save them as plain text and then reformat them from scratch so don't spend too long on formatting your articles!

I've been using the free version of *Eudora for Mac* for some time and have been impressed enough to buy the commercial version. This provides fairly sophisticated mail filtering so I have taken this opportunity to change the submission address for *Overload* so that *Eudora* can file submissions away automatically. In future, please send mail to:

overload@corf.demon.co.uk

if it is intended for publication and

sean@corf.demon.co.uk

if it isn't!

Coming online

I would like to thank Adrian Wontroba for the amount of work he has put in to produce an HTML version of *Overload 7* that is accessible from Alex Yuriev's ACCU WWW site. One of the administrative issues

involved is getting written permission from authors to make their copyright material available in such a public manner – an additional thankyou then to all the contributors who gave that permission.

It has always been my intent with my own contributions to make them available on my www site. I recently took a week off work and spent several days working on HTML versions of my past and current contributions to *Overload* as well as updating my "*C++ – Beyond the ARM*" pages. The last two *Casting Vote* columns, all three *So you want to be a cOOmpiler writer?* columns and this issue's *What's in a name?* are now online with earlier columns to follow. I was pleased to discover that my "*Beyond the ARM*" series gets 10-15 hits every day!

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

Those URLs in full

http://bach.cis.temple.edu/accu
Alex Yuriev's ACCU home page.

http://uptown.turnpike.net/~scorf/overload.html
Overload index page.

http://uptown.turnpike.net/~scorf/cplusext.html
C++ – Beyond the ARM index page.

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++.

Ulrich Eisenecker examines the different uses of inheritance in modelling relationships and Alan Griffiths approaches the problem from the opposite side by showing how a single relationship can be modelled in several different ways. My compiler series continues by taking a brief look at the preprocessor.

Multiple inheritance in C++ – part III

by Ulrich W. Eisenecker

In the last few issues of *Overload* many articles were about multiple inheritance. Therefore I changed the stuff I wanted to write about, to avoid boring the readers of *Overload* and to join the actual discussion!

Very interesting are the – partially philosophical – remarks made by The Harpist and Kevlin Henney for instance. I feel challenged to comment on these and I want to start by quoting Sean Corfield in the editorial of *Overload 8*. He writes: "*... would you say you inherit characteristics from both your parents? ... A clear case of multiple inheritance – what could be more 'real world' than that?*"

Of course, this is multiple inheritance in "real world"! And therefore I will scrutinise the meaning of "multiple inheritance" itself. In the context of object-oriented programming languages this term is used in a way which leads to severe misunderstanding. Provoking? Fine – that is my intention. Let us go on!

Beginning to have doubts...

Considering inheritance in biology it is obvious that both single and multiple inheritance exist. The most common form in nature is to inherit from two parents. The main objective of inheritance is to preserve the ability of a species to adapt to changes in the environment. Only from a very distant point of view could one argue that in software development inheritance serves this objective too. The next great difference between inheritance in nature and (most) programming languages is that every biological individual carries its own plan for construction: class and instance are unified in the individual. Classes and instances are very different at least in C++. A class is a lifeless plan for constructing objects, and objects do not know about their class. Since the introduction of RTTI this has changed a little, but it is still not possible to access all the information related to a class, for instance the inheritance lattice. This is very different in Smalltalk. A class can be accessed via an object during runtime and all class information can be retrieved and manipulated. It is even possible for an object to change the relation to its class and to become an instance of another class. This feature has no model in nature. Maybe genetic engineers will even abolish this invariant of biological organisms.

I hope you now agree to view inheritance in C++ as something quite different from inheritance in nature. I prefer to look upon multiple inheritance as a pure means of language. It expresses the property that the description of a new class is related to the description of an existing class. Special restrictions can apply which are expressed by **public**, **protected** or **private** inheritance.

One syntax, but many semantics...

Inheritance relations can be used for expressing type relationships between classes. In modelling type hierarchies, it is generally important to obey the principle that a type can always be replaced by one of its direct or indirect subtypes (Barbara Liskov's type substitution principle – see Jim Coplien's *Advanced C++ programming styles and idioms*). Clearly multiple inheritance adds some complexity in type hierarchies since one always has to decide carefully, if a class is really a subtype of each of its parents. By the way: C++ is commonly said to be a strongly-typed language. This is true with respect to type checking of function parameters and so on, it does not generally apply for inheritance semantics.

Another possible use of inheritance is to model *aggregation relationships (has-a relationship)*. Of course this sounds horrible to all those who have ever heard of type theory before, but it was a practised style in Lisp-based OO languages – as far as I can remember many years before. To-

day this fashion is deprecated very much and I do not want to explore it further.

Some variant of the former is to use inheritance for modelling *re-use relationships*. That is, when implementing a new class, the implementation of an already existing class is used. It is now a question of design and effort (for duplicating part of the class interface) how such a relationship is handled. It can be done by aggregation and defining new methods which delegate to methods of the aggregated or **private**/**protected** base class object. If the re-used class provides exactly the desired interface no practical obstacles exist to prevent **public** derivation of the new class.

Another important semantic of inheritance is modelling *has-property relationships*. Obviously properties such as being printable, persistent, relocatable and so on are not classes which can be instantiated as objects. They are clearly only properties which – ideally – can be attached to every kind of object and be removed if desired. It would always be fine to keep such properties orthogonal to the type system. Unfortunately in C++ (and other languages too) the inheritance mechanism is used to express them all: type hierarchies, re-use hierarchies, has-property relationships and all the other relationships I am going to tell you about below. The interesting thing about has-property relationships is that in contrast to types and re-use properties they do not necessarily form a hierarchy or a lattice. Typically they divide a hierarchy into three parts. The first part contains the property classes which are mostly not related. The second part contains the classes for instantiable objects which may be related. The third part contains the 'mixin' classes combined from instantiable classes and property classes.

Another possible use of inheritance is to employ it for expressing *value relationships*. You wonder what that is? Okay, that phrase may or may not be well chosen, but the fact exists. What would you call the derivation of an *EmptyString* from an abstract *String*? What about matrices with only null elements? Value relationships can often help to express restrictions primarily attached to values in combination with positive effects especially for memory allocation or to describe algorithmic restrictions. An empty string does not consume memory space for representing any internal data. The same applies for a null matrix. Furthermore, some computations are not allowed or are only allowed for null matrices or triangular matrices and so on. If value relationships are directed so that derived classes widen the set of applicable computations there are mostly no problems. Things become difficult when restrictions apply for derived classes (see also The Harpist's remarks about circles and ellipses in *Having multiple personalities, Overload 8*). Such restrictions can regularly not be expressed statically during compile time (if one does not like to overturn the hierarchy). Therefore methods must be overridden to maintain certain restrictions and to issue an error at run time if necessary (common practise in Smalltalk programming). An assignment of a value other than zero to a null matrix is an example of this.

I have used this approach by deriving ZeroLiteral from IntegerLiteral to express the unique properties of zero when analysing code – Ed.

Another way to use inheritance is for modelling *roles*, which is similar to using *has-property relationships*. Consider a female human being. When she is born she is a baby from the view of her parents and society. As she grows she takes on many roles like pupil, perhaps being mom herself, and so on. Some roles are durable when acquired, some end under certain circumstances, and some are mutually exclusive. The individual never loses its identity or changes its original class but depending on a role an object can respond to different messages or can perform role specific behaviour for the same message. Multiple inheritance can be applied very well for modelling roles. An employee and a father can be joined to form an employed father for instance. In database theory, especially object-oriented databases, the concept of migrating between different roles or the acquisition of roles is much better understood than in programming.

There is another variant of using inheritance, the *is-like-a relationship*. Normally it is used to rank individuals along a specific dimension according to their similarity. It can be also be used to express similarities between classes. In similarity based hierarchies it is no problem to derive *Whale* from *Fish*.

And, as we all know, the whale is an insect – Ed.

Is-like-a relationships often tend to reflect opportunistic or naive classifications. Therefore the is-like-a relationship is similar to the next relationship.

The last variant of using inheritance I can think of is building *taxonomies*. At least in German, "taxonomy" has two meanings. The first is a system for classifying organisms into categories (think about the relation of comparing and classifying classes and individuals yourself) and the second is that of a discipline of linguistics, which is dedicated to the segmentation and classification of language elements describing the structure of language systems. Particularly for the second meaning, inheritance can be used to model hierarchies of terms and concepts forming a specific language. In the broad field of object-orientation, artificial intelligence and linguistics, colleagues and I already use this principle for structuring and modularising word and phrase groups, so called "vocabularies", of domain specific languages using inheritance hierarchies. Aside from such rather esoteric applications of inheritance for building taxonomies many inheritance hierarchies are at least partially taxonomies. Taxonomies are built from a specific point of view and do not necessarily reflect type relationships or relations which are of cosmic truth. Instead, they are very well formed is-like-a relations which needed many reflection and a long time to grow. By the way, some experienced Smalltalk programmers like to talk of sophisticated inheritance hierarchies as taxonomies.

A first figure...

In my opinion it is a nice way to depict the relations between the different semantics of using inheritance in form of a taxonomy graph. A first draft of such a graph is given in Figure 1: Taxonomy of inheritance relationships. I would appreciate any comments or criticisms for improving its current structure.

Mixing it up...

I do not believe that I have succeeded in enumerating all possible semantically different uses of the syntactic means of inheritance. I hope that I have made you question your "common sense" understanding of inheritance. In my opinion it is very important to follow a consistent principle in using multiple inheritance, make it obvious to those using a class hierarchy, and to document the exceptions from applied rules due to practical circumstances. I am not sure whether it would be wise to deprecate all uses of inheritance except type-relationships, but I agree fully with Kevlin Henney who states the importance of clearly defining the purpose of a class in analysis. It is always misleading or irritating if a class designer does not know his own intentions when using inheritance. It also becomes very difficult and erratic if many styles of using inheritance and multiple inheritance are applied unsystematically and without documentation in a class hierarchy.

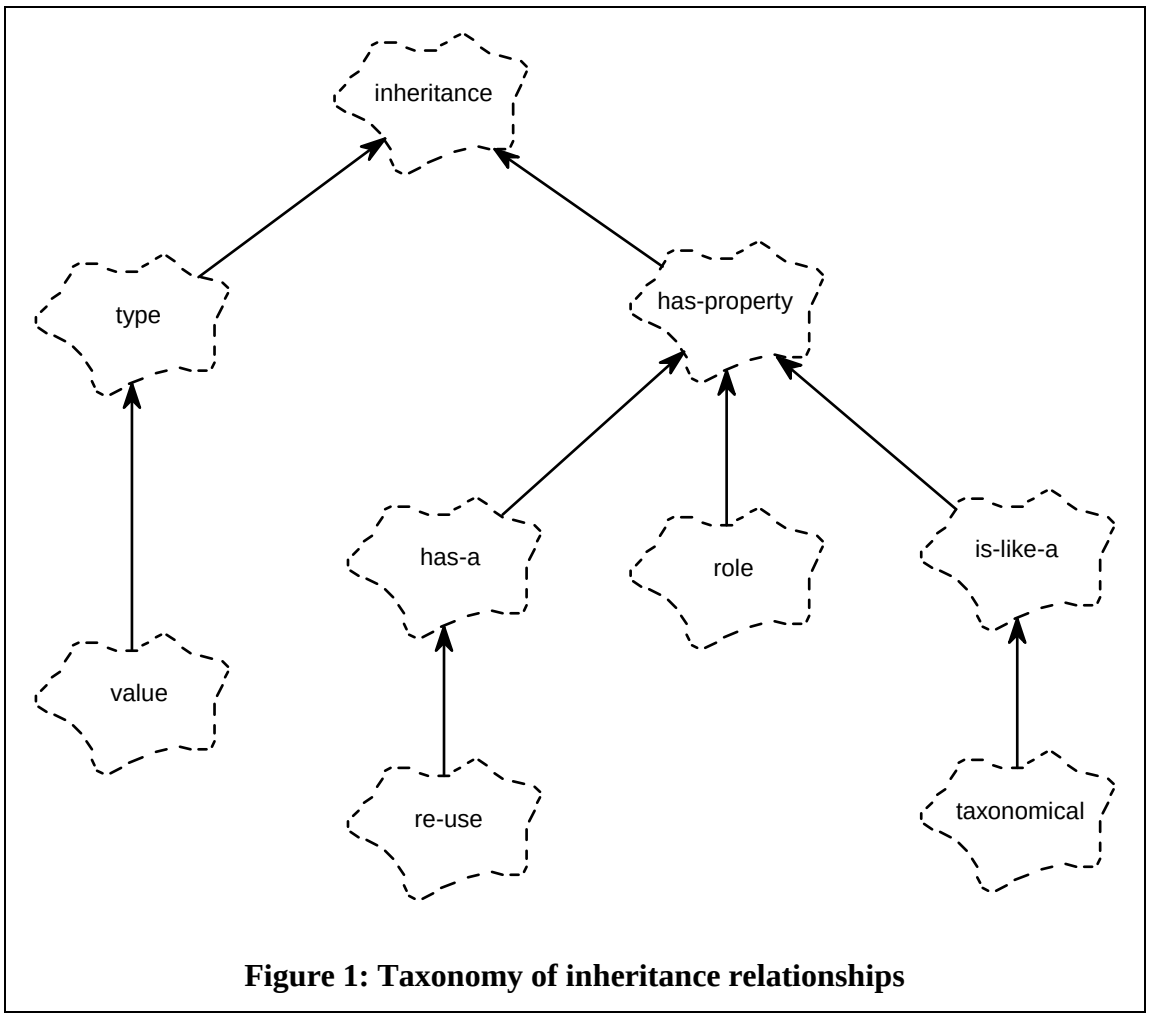

Figure 1: Taxonomy of inheritance relationships

Outlook

Well, inspired by the previous contributions in *Overload* I wrote about something completely different to what I formerly intended and you saw no C++ code. I hope you can forgive me and I will be very pleased if this article provokes some discussion and follow-up articles. After discussions with the editor I will decide the next topic in the context of multiple inheritance to write about.

Ulrich W. Eisenecker
eisenecker@mbgate.augusta.de

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

by Sean A. Corfield

At the end of part II, I provided some skeleton classes and asked you to consider what the interfaces should be. I'm going to start by fleshing out one of those interfaces and then look in a little more detail at some aspects of preprocessing. I may come back to the other classes in a future issue but I am no longer in a position to divulge some of the details that I had planned!

At source

The class I want to look at is the base class *Source*. In a purely abstract sense, all we can say for sure about it is that we can repeatedly 'get' items from a *Source* until it is 'empty':

```
template<class T>
class Source
{
public:
  virtual T  get() = 0;
  virtual bool      empty() const = 0;
  virtual ~Source() { }
};
```

The member functions are both "pure virtual" because there can be no sensible generic implementation for them – they must be provided by more derived classes. The class has no data members – no state – so a default constructor (implicitly generated by the compiler) is appropriate here. However, we must provide an explicit **virtual** destructor because the default would be non-**virtual** and could lead to problems later on. In many ways, assignment and copy

construction are irrelevant because the class has no state information. What kind of *Source*s will we be dealing with in reality? We've already said each *Phase* is a *Source* but we also need the lowest-level *Source*: a file. Or is it? An *istream* is a more general *Source* than an *ifstream* so perhaps we should consider that first:

```
template<class T>
class StreamSource
: public Source<T>
{
public:
  StreamSource(basic_istream<T>& i)
  : i_(i) { }
  virtual T  get()
              { T c; i_.get(c); return
c; }
  virtual bool      empty() const
                       { return
i_.eof(); }
  virtual ~StreamSource() { }
private:
  basic_istream<T>& i_;
};
StreamSource<char>  charSource;
StreamSource<wchar_t>      wideSource;
```

By templatizing *StreamSource* and using *basic_istream*, we take our first steps towards 'global' programs – a useful point to remember.

Whither STL?

Kevlin Henney noted in *Overload 9* (*Applying the STL mindset*) that the phases could probably be rewritten in terms of iterators and, given the above class interfaces, we are going to end up with a lot of code that does something like this:

```
Source<T>& st = ...;
while (!st.empty())
{
  T t = st.get();
  // do stuff with t
}
```

STL-style iterators would indeed allow us to rewrite this as:

```
Source<T>& st = ...;
for (Source<T>::iterator i = st.begin();
       i != st.end();
       ++i)
{
       T t = *i;
       // do stuff with t
}
```

By definition, however, a *Source<T>*::*iterator* would be an "input iterator" and these are one-pass iterators which come complete with a lot of semantic restrictions. For the initial input phases of preprocessing, this is not a great problem but as the mapping involved in each phase becomes more complex the restrictions associated with input iterators make them unworkable. The approach that I took was to implement all the early phases with the *get*/*empty* interface and then collect all the tokens that came out of preprocessing into a list that could be iterated over by the parser:

```
Preprocessor* p = new

Preprocessor(filename);
list<Token>  source;
while (!p->empty())
{
       source.push_back(p->get());
}
parseProgram(source.begin(),
source.end());
```

This is somewhat simplified because the actual input to the parser is the output of phase six whereas the preprocessor is only really phases 1 to 4 – see Table 1: Phases of translation.

Assuming that we really wanted to implement the input iterator for a *Source*, let's examine how we'd go about it:

```
template<class T>
class Source
{
public:
  class iterator
  : public input_iterator<T>
  {
  public:
    iterator(Source<T>* sp)
    : sp_(sp) { }
    T operator*()
                      { return sp_-
>get(); }
    iterator& operator++()
                           { return
*this; }
    iterator operator++(int)
                           { return
*this; }
    friend bool
                operator==(const
iterator&,
                           const
iterator&);
  private:
    Source<T>* sp_;
  };
  iterator begin()
                 { return
iterator(this); }
  iterator end() { return iterator(0); }
  // ... as before ...
};
```

The implementation of the equality operator is left as an exercise for the reader. Note two things:

1. this assumes the **!=** operator provided by STL – a template operator that guarantees "x != y" means "!(x == y)",
2. every time you use **operator*** on the iterator, the *Source* is advanced.

That last point may surprise you – in fact, the semantics of input iterators are under review by the C++ committee at present and it may turn out that the above implementation violates the intended requirements of an input iterator (I don't believe it does at present). If the requirements change, then the 'current item' would need to be cached within the iterator and a few other adjustments made – I may revisit this in a future article.

Parsing a program

In C++, a program is a sequence of declarations, so it shouldn't surprise you that I implemented that as follows:

```
void parseProgram(
       list<Token>::iterator cur,
       list<Token>::iterator eof
)
{

  while (cur != eof)
  {
    cur = parseDeclaration(cur, eof);
  }
}
```

parseDeclaration takes a pair of iterators specifying a range, [*cur*, *eof*), and returns an iterator that refers to the next, unparsed *Token* in the list. I will return to this later in the series.

Of symbols...

In *Overload 8*, I commented that the symbol table is an obvious abstraction in a compiler. The main symbol table required in C++ is relatively complicated because it needs to be "scope-aware" but the preprocessor also needs symbol tables – for macros and preprocessor directives. First of all, let's look at what information we need for a preprocessing token: its name (or spelling), its "key" (e.g., *IDENTIFIER*, *HASH*, *WHITESPACE*) and its position in the source code. We need the latter to be able to accurately report the location of warnings that we detect later on in the analysis. For the purposes of preprocessing, only a few different types of token are important, in particular there are no keywords.

There are two types of macros: object-like and function-like. In an ideal world we could implement these along the following lines:

```
class Macro
{
public:
  Macro(const string& n,
        const list<Token>& b)
  : name_(n), body_(b) { }
  virtual ~Macro() { }
  list<Token> expand() const = 0;
protected:
  const string& name() const
  { return name_; }
  const list<Token>& body() const
  { return body_; }
private:
  const string name_;
  const list<Token> body_;
};


class ObjectMacro
: public Macro
{
public:
  ObjectMacro(const string& n,
              list<Token> b)
  : Macro(n, b) { }
  virtual ~ObjectMacro() { }
  list<Token> expand() const;
};
```

```
class FunctionMacro
: public Macro
{
public:
  FunctionMacro(const string& n,
                list<Token> b,
                list<Token> p)
  : Macro(n, b), params_(p) { }
  virtual ~FunctionMacro() { }
  bool bind(const list<Token>& args);
  list<Token> expand() const;
private:
  list<Token> params_;
// ...
};
```

Then preprocessing would proceed like this:

```
  if (token.key() == IDENTIFIER)
  {
    MacroTableIterator m =

macroTable.find(token.name());
    if (m != macroTable.end())
    {
      if (FunctionMacro* fm =

dynamic_cast<FunctionMacro*>(&*m))
      {
        // collect arguments from token
        // stream
        if (!fm->bind(args))
        {
           warning(token.where(),
                   BAD_MACRO_CALL,
                   token.name());
        }
      }
      list<Token> expansion = m-
>expand();
    }
  }
```

Notice how we can bind the macro arguments after downcasting in the case of a function-like macro but always despatch the macro expansion from the base class. Unfortunately, RTTI is not portable enough at the moment to allow us the luxury of implementing macros like this. Instead,

For those of you without either *Overload 5* or the memory of Methuselah, here's a resumé of the phases of translation:

1. map character set and trigraphs
2. splice \ <newline> pairs
3. map characters to preprocessing tokens and replace comments by whitespace
4. expand macros and include directives
5. map escape sequences in character and string literals
6. concatenate adjacent string literals
7. convert preprocessing tokens to tokens and perform syntax and semantic analysis
8. combine translation units to form a program

Table 1: Phases of translation

I implemented it all in one class (*Macro*) and provided a method *isFunction***()** that indicated whether binding arguments was a sensible operation – an engineering compromise. *macroTable* is implemented as a hash table – something sadly missing from the draft standard library – that maps the name of a macro to the implementation of the macro (*hash_map<string,Macro>*).

The preprocessing operations that affect macros can be easily implemented:

```
// #define macroName body
// #define macroName(args) body
  MacroTableIterator m =

macroTable.find(macro.name());
  if (m == macroTable.end())
  {
    macroTable[macro.name()] = macro;
  }
  else
  {
    // if args & body are not identical
to
    // previous definition, warn about
it
  }

// #undef macroName
  MacroTableIterator m =

macroTable.find(macro.name());
  if (m != macroTable.end())
  {
    macroTable.erase(m);
  }
```

Service included

Since the **#***include* directive causes phases 1 to 4 to be recursively applied to the specified file, implementation should just be a matter of creating a new preprocessor on that file, '*get*'ing all the *Token*s from it and priming the owning preprocessor with that list of *Token*s – this implies that *Preprocessor* has a cache:

```
bool Preprocessor::empty() const
{
  return cache.empty() && lexer.empty();
}
Token Preprocessor::get()
{
  if (cache.empty())
  {
    // perform normal preprocessing
  }
  else
  {
    Token t = cache.front();
    cache.pop_front();
    return t;
  }
}
```

```
// #include filename
  Preprocessor* included = new

Preprocessor(filename);

  while (!included->empty())
  {
    cache.push_back(included->get());
  }
  delete included;
```

Coming soon

Whilst I have obviously glossed over many of the details of preprocessing, I hope that this gives you a feel for what is involved. In part IV, I shall move on to look at representing the C++ type system and some of the engineering considerations involved.

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

When is an "is a" not an "is a"?

by Alan Griffiths

Setting the scene

The software development process has always suffered from the difficulty of relating the initial description of a problem to the implementation of a system to solve it. Object-orientation appears to offer an easy method of mapping concepts between analysis of the problem domain, the system design, and the implementation. It appears simple to track the classes and relationships described in each of these processes to those described in the others. In practice it is not as simple as it seems because many of these apparent mappings are invalid.

To take a simple example consider a membership application form handled by the ACCU. This is initially sent to the treasurer who pays the cheque in (and waits for it clear) before forwarding the application to the membership secretary who then enters it into the membership database and files it.

- In the analysis model the *MembershipApplication* object would be represented as having states "being processed by treasurer", "being processed by membership secretary", and "completed processing".
- In the design (especially given the distributed nature of the ACCU administration), each of these states is represented as a collection of applications.
- When it comes to implementation it will be noticed that *AppsBeingProcessedByTreasurer* and *AppsBeingProcessedByMembershipSecretary* have common functionality and an implementation class (*SetOfMembershipApplications*) will be abstracted.

Given the above elaboration of the system during development it is no longer a trivial matter to trace the "state transitions" of the analysis model directly into the implementation. (They are still there of course but their representation has changed – they no longer belong to the *MembershipApplication*.) For those who doubt the validity of the above scenario, it is precisely what I was told to do on an OMT training course! (I was the only one on the course that appeared uncomfortable with the gaps in this progression.)

Now let's examine the relationships between some of these classes. One of the classic tests for valid inheritance is the "is a" test. That is "a class of objects *A* is a subclass of another class of objects *B* if each instance of an *A* is also an instance of a *B*".

When implementing such a design it is important to note that C++ supports a number of mechanisms of implementing an is-a relationship and that choices need to be made amongst them. The three main forms of is-a are:

- Inheritance from a public base class.
- Instantiation of a template class.
- Conformance to the restrictions on a template argument.

Dealing with these choices is a C++ "program design" issue which serves to further separate the implementation from the earlier development stages. These forms are discussed below, but note that there is no corresponding choice at the analysis or design levels of the development process. Indeed, as far as I am aware no other programming language requires (offers?) these choices.

Inheritance from a public base class

In the above example the classes *AppsBeingProcessedByTreasurer* and *AppsBeingProcessedByMembershipSecretary* are related as each of them is a *SetOfMembershipApplications* – they support the same methods, for example:

```
insert(const MembershipApplication& ma);
```

In this case a public base class *SetOfMembershipApplications* is appropriate:

```
class AppsBeingProcessedByTreasurer
: public SetOfMembershipApplications
{ ... };
```

Instantiation of a template class

Now consider the classes *SetOfMembershipApplications* and *SetOfMembers*. Each of these is a "set", but you cannot treat a *SetOfMembershipApplications* as a *SetOfMembers* as the functions have different signatures:

```
insert(const MembershipApplication& ma);
insert(const Member& m);
```

The approach here is to use a template *set* class as provided by the standard library:

```
typedef set<MembershipApplication>

SetOfMembershipApplications;
```

In real life, there is probably a need to add functionality to the functions supplied by *set* in *SetOfMembershipApplications* so delegation may be more appropriate:

```
class SetOfMembershipApplications
{
public:
  //...
  virtual void insert(
       const MembershipApplication& ma)
       { apps.insert(ma); }
private:
  set<MembershipApplication> apps;
};
```

Conformance to the restrictions on a template argument

One of the interesting features of templates is way that the operators and member functions

applied to template arguments are resolved. This means that a template can be written which requires an expression such as *t1* < *t2* to be valid and to return true if *t1* precedes *t2* regardless of how "precedes" is interpreted and whether the expression is evaluated by the built-in < operation, a member operator ***T*::operator<()** or by a binary operator **operator<(*T*&, *T*&)** that happens to be in scope when the template is instantiated.

In terms of our example there could be a template function *printAddressLabel***()** that requires its argument to provide member functions *getName***()** and *getAddress***()** with some appropriate behaviour. From the viewpoint of this function either *Member* or *MembershipApplication* is a source of address information without the need to introduce a *SourceOfAddressInformation* base class for each of them.

Artificial classes such as the hypothetical *SourceOfAddressInformation* deepen the inheritance hierarchy, often require multiple inheritance, and (when their purpose is to assist such utility functions) they tend to proliferate. They are best avoided.

In conclusion

C++ is such a rich language that there will be other ways of implementing "is a" that are not covered above. However, it has been demonstrated that blindly following the design to produce a hierarchy based on public base classes is not the only option.

The challenge is to ensure that implementation and the design are close enough to ensure that the system is correctly implemented without forcing the more flexible "is a" relationships shown in the design into a straitjacket based on a single implementation of "is a".

Alan Griffiths
alan@octopull.demon.co.uk

The Draft International C++ Standard

This section contains articles that relate specifically to the standardisation of C++. If you have a proposal or criticism that you would like to air publicly, this is where to send it!

After the various comments that have been raised about namespaces, I look at an alternative explanation for the way they work in an attempt to clarify what the draft standard says.

What's in a name?

by Sean A. Corfield

In *Overload 8* I promised that I would return to look at namespaces in more detail. When Bjarne Stroustrup presented the proposal to the committee in Munich in July 1993, he said that the feature was simple enough to explain to C++ programmers "in ten minutes" and could be "implemented in ten days". Prior to the Munich meeting, I had implemented most of the namespace mechanism in a day and it seemed very straightforward. However, I, like many other committee members, had not grasped a couple of subtleties of the namespace mechanism and had implemented it incorrectly. Metaware – the only commercial compiler implementation – had also implemented it incorrectly (Metaware's implementation largely agreed with my own!). Let's look at why the confusion arose and how namespaces *really* work!

First principles

The principle behind namespaces is simple enough: provide a way to partition the global scope to allow mix'n'match between components from different libraries – see my *short exposé* in *Overload 8*. A namespace may only be defined at file scope or nested directly in another namespace. Names declared inside a namespace can be accessed from outside the namespace by using the fully qualified name.

```
namespace ACME
{
  class Widget ...
}
ACME::Widget w;
```

A namespace can be "unlocked" for the purpose of name lookup with a *using-directive*, e.g., **using namespace** *ACME***;** or an individual name can be imported into the current scope with a *using-declaration*, e.g., **using** *ACME***::***Resource***;**

The *using-declaration* behaves exactly as if the unqualified name had been declared at the point

of the *using-declaration* and it is a synonym for the fully qualified name.

The *using-directive* says "if name lookup reaches file scope (or namespace scope) before the name has been found, search any and all unlocked namespaces as well as file scope (or the namespace scope we have reached)".

In addition, file scope is now deemed a namespace scope, but with a name you cannot utter. This makes the name lookup rules simpler by removing the 'special case' of file scope and we can rewrite the *using-directive* rule as: "if name lookup reaches namespace scope before the name has been found, search any and all unlocked namespaces as well as current namespace scope".

I'll explain this in more detail below, but first I want to round off the 'feature list'.

A shorter alias for a long namespace name can also be defined, e.g.,

```
namespace ACME =

a_company_that_makes_everything;
```

This also allows you to mix'n'match libraries more easily:

```
namespace lib = Modena;
// can easily change to:
// namespace lib = RogueWave;
// namespace lib = std;

// never need to change this:
lib::string banner;
```

It all looks so simple, doesn't it? How could we possibly have been confused?

Confusion will be my epitaph

Unfortunately, a *using-directive* and a *using-declaration* look similar enough that many people think the directive is some sort of declaration. Barry Dorrans' letter in *Overload 9* is typical of many programmers' first reaction to seeing namespaces. Even with hindsight, I don't know what syntax the committee could have picked to avoid *this* confusion. Part of the basic problem with namespaces is that they look a lot like classes or named scopes: they have a keyword, a "tag" and a brace-enclosed list of declarations:

```
namespace MyLib
{
  class ConfigurationFile
  {
  // ...
  };
} // ...
```

This naturally leads programmers to expect a namespace to behave in a similar way to a class and at least obey rules of scope. I think this is what causes many programmers to think that a *using-directive* will somehow be "found" during lookup prior to reaching an outer scope.

Second principles

There has recently appeared – within the committee, at least – an explanation of namespaces that does not so easily give rise to this confusion. Consider a namespace as simply a shorthand for exactly what we want to achieve – separation of names:

```
namespace MyLib
{
  class ConfigurationFile
  {
  };
}
namespace YourLib
{
  class ConfigurationFile
  {
  };
}
```

If we didn't have namespaces, we'd probably write this:

```
class MyLib_ConfigurationFile
{
};
class YourLib_ConfigurationFile
{
};
```

Here, the names are all in the global scope and "fully qualified" names would simply the the entire name with the prefix (*MyLib_* or *YourLib_*). This is, after all, something like the way that compilers will implement namespaces anyway. Let us now examine what a *using-declaration* does in this context:

```
using MyLib::ConfigurationFile;
```

becomes equivalent to:

```
typedef MyLib_ConfigurationFile

ConfigurationFile;
```

If the name referred to a variable, it would be like having a local reference to the original:

```
Type& name = MyLib_name;
```

or if it were a function, we could have a local delegation function:

```
inline Type func(Arg arg)
{
```

```
  return MyLib_func(arg);
}
```

(allowing for the subtleties of pass-by-reference and so on). What about the troublesome *using-directive*? In this imagined context, **using namespace** *ACME***;** says "if name lookup reaches global scope, look for *ACME_name* as well as *name*". Returning to the "confusing" example in *Overload 9*:

```
namespace A
{
  int j;
}
void f()
{
  int j = 0;
  if (j)
  {
    using namespace A;
    j = 0;
  }
}
```

Rewriting this to use a prefix instead of a namespace gives:

```
int A_j;
void f()
{
  int j = 0;
  if (j)
  {
    // using-directive means:
    // look for A_name as well as name
    j = 0;
  }
}
```

Clearly, the *j* in the assignment must refer to the local *j* because *A_j* is in an outer scope and name lookup never reaches it. Replace the *using-directive* with a rewritten *using-declaration*:

```
int A_j;
void f()
{
  int j = 0;
  if (j)
  {
    int& j = A_j;
    j = 0;
  }
}
```

and it should be clear that the *j* in the assignment now refers to the local reference and hence to the global *A_j*.

Like a namespace within a namespace

The simile used above can be extended to nested namespaces and *using-directive*s too:

```
namespace A
{
  int j;
}
namespace B
{
  namespace C
  {
    int i;
  }
  using namespace A;
}
```

B::*C*::*i* can be treated as a global scope identifier called *B_C_i* and then the lookup rules described above apply. The *using-directive* inside *B* says "look for *A_name* as well as *B_name*". Now if we unlock *B* with a *using-directive* ("look for *B_name* as well as *name*") we simply end up with a list of possible names to find *in the global scope*. Even if we use qualified name lookup to look inside *B*, we can still use this simile to explain the rules: *B*::*j* causes a lookup of *B_j* and since the *using-directive* in *B* adds *A_name* to *B_name* as possible candidates, we will correctly find *A_j* as expected. I should point out that qualified name lookup prior to Monterey (July '95) did not work in this manner but it was clearly the intent of the original proposal that it should and synthesis of namespaces relies on that property.

Synthetic libraries

As I explained in *Overload 9* (page 18), one of the goals of namespaces was the ability to synthesise 'new' namespaces from old ones, allowing a company to provide a single standard namespace that all its programs can rely on without worrying about exactly where names really come from. This should provide great benefits to companies that use many different vendors' libraries and have to deal with differences between versions of those libraries (in fact, this was one of the example Bjarne originally gave in '93 to support his proposal). All the differences can be dealt with in one place – the synthesised namespace – and no client code should need to be changed.

The nameless ones

Finally, there are unnamed namespaces. They are exactly like other namespaces with two slight differences:

1. their name is unique and cannot be uttered
2. each unnamed namespace is implicitly "unlocked"

That means that:

```
namespace
```

```
{
}
```

is absolutely equivalent to:

```
namespace UNIQUE
{
}
using namespace UNIQUE;
```

where *UNIQUE* is some compiler-generated name (probably with lots of digits and underscores in it!). Since unnamed namespaces effectively do have a name, all the previous explanation applies.

Alas poor static...

The committee have decided that file scope **static** should be deprecated – marked for possible future removal – because unnamed namespaces provide an alternative that is more consistent with the future direction, or style, of C++. Since file scope **static** can be implemented by generating a unique prefix for each translation unit and then treating the full names as externally linked, it should be easy to see how unnamed namespace can be used instead:

```
static int i;
static int j;
// can be treated as:
int UNIQUE_i;
int UNIQUE_j;
// which is equivalent to:
namespace UNIQUE
{
  int i;
  int j;
}
using namespace UNIQUE;
```

Still confused?

Ideally, you'd need to go away and try the code examples on your favourite compiler. Unfortunately, only Metaware supports any form of namespaces and that doesn't obey the rules given above. Several major vendors are working on namespaces now – hopefully implementing them the same way – so it shouldn't be too long before we can "play" with this useful, two-year old language feature!

If you have any questions or comments about the above, I'd like to hear them – perhaps a follow-up article will be necessary?

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

C++ Techniques

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

The Harpist provides a real world example of polymorphism to examine the issues involved in designing a 'proper' polymorphic type. Roger Lever begins a series on writing useful classes for debugging and Francis Glassborow provides a utility class for tracking order of destruction.

Addressing polymorphic types

by The Harpist

derived from an idea by Francis Glassborow

You will find both questions and tasks included in the body of this article. They try to identify areas that need either further development or extended exploration. I hope that all of you will look at these carefully and, if you can, provide appropriate articles about one or more. I would hope to see quite a number of items in future editions of *Overload* derived from this article.

Some time ago Francis and I had a discussion about examples of polymorphism. We both agreed that the ubiquitous *Shape* hierarchy was a poor example for reasons that have been covered in earlier issues of *Overload*. (Actually, I think that *Shape* is a good discussion topic for intermediate C++ programmers with leanings towards becoming class designers rather than writers of client programs and application code.) What Francis wanted was a simple concept that virtually demanded multiple implementations of a single interface. After throwing ideas around for a time, Francis suggested the idea of an *Address*. We refined that down to a postal address (excluding email addresses and speeches) but decided for simplicity to keep to the name *Address* for the base class.

I think that we can largely agree on a public interface for *Address* objects while recognising that the exact implementation will depend on the country in which the address is located.

My feeling is that the address concept is one that is ideal for introducing the ideas of polymorphism and reuse. Addresses are very common pieces of data and so justify the overhead for developing a good reusable implementation (class hierarchy). They also exhibit just about all the problems of pure polymorphic types. I use the term 'polymorphic type' to refer to the concept of types that share a single public interface without any extensions in sub-types.

Let me offer a possible ABC (abstract base class) for *Address*.

```
class Address {
  const char * const country;
public:
  enum type { UK, US, Germany, France };
  const char* get_country()
                        { return
country; }
  virtual void printon(ostream& =cout) =
0;
  virtual void getfrom(istream& = cin) =
0;
private:
  void operator=(const Address&);
public:
  Address(const char* );
  Address(const Address&);
  virtual ~Address() = 0;
};
```

No doubt there is other functionality that you might want to include but I think the above is a good starting point. For the less experienced (and so that the experts can pull it apart) here is my rationale for the various elements in my *Address* interface.

```
class Address {
```

I have elected to use the keyword **class** to emphasise that this is a specifically C++ structure. I reserve identifiers with single leading upper case characters as type names. This is purely an element of my personal style. The most important rule here is to be consistent.

```
  const char* const country;
```

This is an addition of mine. Francis' original design has no data in his ABC. I decided to add this single data member because it is a property of all postal addresses that they are located in a country even if that is not stated anywhere else. I chose to use a **char*** because I wanted to assign space dynamically. This choice does raise some issues about efficiency. It minimises stack use (OK, use of local storage if you want to be picky) but at a cost of speed (dynamic memory allocation and release is always more time consuming than any other form). At this stage I avoided using a *string* class because the Standard Library version is still not stable. On a second pass I would almost certainly consider replacing **char*** with either **wchar_t*** or some form of *string* object. As this is private data I can change my mind freely.

My choice of the two **const** qualifiers is rather more debatable. The first one effectively means that nothing may change the spelling of the *country* data. Optimisers can sometimes make use of such information. The second one prevents any attempt to reassign storage for *country*. Actually this has a valuable side effect (which is why I am using it), it requires value of *country* to be set in a ctor-init list (constructor-initialiser list). You will see that this works quite well with the concrete classes that follow. You will also see that the first **const** works rather badly.

```
public:
  enum type { UK, US, Germany, France };
```

This will be a list of all the countries supported by the hierarchy. As we pursue this we will find that somewhere we need such a list. This may not be the right place because it means that extending the hierarchy will require us to tinker with the abstraction. I am deliberately leaving this as an open question because I want to see what ideas our expert members will suggest.

Expert question: How should knowledge of concrete classes be encapsulated? Can you avoid providing it in the base class?

I have just placed four countries in this enum, in a full implementation there would be many more.

```
  const char* get_country()
                        { return
country; }
```

As this field is common to all *Address* types we can safely provide read access at this level. If we later change the type of storage the body will have to provide a conversion. As this might not be possible in some instances (e.g., providing full international spellings with **wchar_t**) I think providing a minimal class to handle names of countries needs to be added, urgently. Once such has been provided, you have far more room for future change.

Intermediate task: design and implement a class that provides support for using international character sets for country names.

```
  virtual void printon(ostream& =cout) =
0;
```

This function will allow us to provide an appropriate output for printing, screen display and persistent storage (though the latter has some ramifications). Note that it defaults to screen display, which seems reasonable to me. It is also a pure virtual function which forces the concrete classes to provide an implementation, though, as we shall see in a moment, it does not prohibit an implementation at ABC level.

```
  virtual void getfrom(istream& = cin) =
0;
```

And this is the symmetrical operation to allow data to be read into an instance.

```
private:
  void operator=(const Address&);
```

We know that we must do something about copy assignment because if we don't the compiler will. At first sight you may think that the assignment should be virtual. I don't think that will work – the derived versions would also have to take an **const** *Address***&** parameter. The client code is going to be using *Address****** and *Address***&** (i.e., pointers and references to the ABC). This leads to a problem, because we might have something like this:

```
Address& ger = *new German_address();
Address& fra = *new French_address();
ger = fra;
```

That won't work because we would be back to the polymorphic object problem that I tackled in the last issue (circles and ellipses). If we are going to provide copy assignment we are going to need something much smarter. For the time being I have declared the function private. That way we inhibit generation of default copy assignment both here and in derived classes (they will try to use the base class copy assignment and get an access violation). Strictly speaking, we need not provide a copy assignment declaration as long as country is of a **const** qualified type – the compiler cannot generate copy assignment (nor copy construction either) because it would violate the rules on assignment to (initialisation of) **const** qualified variables.

Expert task: provide a copy assignment for concrete address sub-types that will throw an exception (or otherwise fail under control) if the left and right sub-types are different.

```
  Address(const char*);
```

While we cannot have any instances of plain *Address* objects, we are going to need them as sub-objects so we need a constructor. Yes, we really do, as this has been written because *country* is both **const** qualified so must be initialised, and with private access can only be touched in the context of a plain *Address*.

```
  Address(const Address&);
```

If we are to copy concrete *Address* sub-types we must be able to copy the base sub-object.

```
  virtual ~Address() = 0;
```

In order for polymorphism to work we must provide a virtual destructor. As I believe it would always be an error for the destructor of a polymorphic base to be non-virtual I think that the language should specify that it will be. The question remains as to whether it should be a pure virtual (i.e., should I require implemented destructors in derived classes)? Even if I do so I will still need a base class implementation:

```
Address::~Address()
{
  delete[] country;  // note: delete[]
                     // not delete!
}
```

Which leads to all sorts of interesting debates: **const** qualified objects may not be changed: surely deleting them is changing them and so on. The fact of the matter is that you need to release the storage and the language allows it to be done this way. Yet, note that you cannot initialise it in the body of a constructor – that is too late. Its these inconsistencies that cause so many problems to novices. [We have even more fun with the logic of deleting member data within the body of a destructor when the object being destroyed is itself **const** qualified. I could make a strong case for requiring **const** destructors for **const** objects. That would require cv-qualification information to be stored in the RTTI (run time type information) record for a variable. As the language currently stands, cv-qualification creates a compile time fine structure for types that is not available at run time. Or in other words, the range of static types is not the same as that for dynamic ones.]

Some more implementation

Before we look at a concrete class derived from *Address*, we need to finish the ABC by providing implementations for its constructors.

```
Address::Address(const char* c)
: country(new char[strlen(c)+1])
{
  strcpy(const_cast<char*
const>(country),
         c);
}
```

I think I have this right. First attach sufficient dynamic memory to country to contain the string passed in as an argument, then temporarily suspend the **const** qualification on the elements of the **char[]** so that you can write to them. This is where having some form of *string* type would avoid the problem. One reason why I elected to use dynamic arrays of **const char** was to focus your attention on this problem.

Note that this is not a default constructor, but we do not need one because you cannot have a plain *Address* object. More to the point, you cannot have an array of pure *Address* objects. When do you need default constructors? You usually need them to create arrays. Here we hit an oddity of polymorphic types, you can have single polymorphic instances because you can do something such as:

```
Address& ger = *new German_address();
```

You can have static arrays of a polymorphic subtype:

```
German_address adds[10];
```

You can have arrays of pointer to a polymorphic type:

```
Address* padds[10];
```

But there is no mechanism for creating an array of type *Address***&**. Whatever the rights of the matter are, arrays of references are not supported by C++. This does not matter as long as you, as a programmer, stick rigidly to the concept that an ABC provides the whole interface and that you get the same behaviour independent of how you access an object. However as soon as an object has different behaviour depending on its mode of access the lack of arrays of references might cause problems. This is not a criticism of C++, I think the position it adopts is perfectly reasonable, however it does mean that arrays of polymorphic objects have to use explicit indirection (rather than the implicit indirection that is used to implement references).

I think that the argument that the inconsistency is in allowing references exhibit polymorphic behaviour has some force (not a lot, but some).

```
Address::Address(const Address& a)
: country(new char strlen(a.country)+1])
{
  strcpy(const_cast<char*
const>(country),
         a.country);
}
```

Now that makes the copy constructor almost identical to the normal constructor. Perhaps we should extract the code that writes to *country*. It is also possible that a derived class might want the power to change *country*. Such a decision would be one taken by class hierarchy designer with more than a little care. If you decided that you wanted to provide write access to *country* for derived classes what you must **not** do is change *country* to protected access. That would forever tie you to **const char* const**. What you need would be something like:

```
void Address::set_country(
              const char* c)
{
  delete[] country;  // free up any
current
                     // dynamic memory
  const_cast<char *>(country) =
            new
char[strlen(a.country)+1];
  strcpy(const_cast<char*
const>(country),
         c);
}
```

With the prototype declared **protected** in the definition of *Address*.

Because we must protect against possible memory leaks with the **delete[]** in the above function, we will have to change our two constructors so that they will make efficient use of this function. Something like the following:

```
Address::Address(const char* c)
: country(0)
{
  set_country(c);
}
Address::Address(const Address& a)
: country(0)
{
  set_country(a.country);
}
```

Now let us look at implementing a concrete class based on *Address*. I am not going to provide more than skeleton code for this.

Novice/intermediate task: flesh out and fully implement *German_address*, *UK_address*, *US_address* and *French_address*.

```
class UK_address : public Address {
  // UK specific data elements
public:
  void printon(ostream& = cout);
  void getfrom(istream& = cin);
public:
  UK_address(istream&);
  UK_address(const UK_address&);
  ~UK_address();
};
```

I am leaving most of the implementation to you but a possible implementation of the constructors might be:

```
UK_address::UK_address(istream& in)
: Address("United Kingdom")
{
  getfrom(in);
}
UK_address::UK_address(
       const UK_address& old)
: Address("United Kingdom")
{
  // code to copy the rest of the data
}
```

When we come to implement *printon***()** we will probably realise that we have tried to do rather too much by allowing the same function to write to all kinds of output. If we do not realise it then, we will when we come to tackle persistence – that is writing data to storage so that it can be read back again.

The problem of persistence

Consider the following:

```
int main()
{
       Address* data[100]= {0};
       for (int i = 0; i < 100; i++)
       {
              get_address(data[i]);
       }
       ofstream outfile("mydata");
       for (int j = 0; j < 100; j++)
       {
              data[j].printon(mydata);
       }
       mydata.close();
       return 0;
}
```

Now how do I read that data back into an array? My read data function must identify the correct kind of address sub-type then dynamically create an object of that sub-type so that it can read the data into that object. The logical place for such a function is as part of the *Address* ABC. But this function needs to know about all the sub-types. Also each address must be stored with some way of identifying its sub-type. This was why I had an enum in my ABC. It probably needs tucking away somewhere else (I can think of quite a few solutions, but I would like to see some from our experts) because as currently written, the ABC must be changed each time a new sub-type is added.

Let me sketch a possibility. Suppose *Address* includes a static data member that was a linked-list of *Address****** (or *Address***&** if you prefer). Now suppose that each sub-type includes a registration function that, when called the first time, added an instance of itself to the linked-list (and then did nothing if called again). Add a virtual function *Address****** *make_me***() = *0*;** to the ABC. Some time ago a special relaxation to the rules for return types of virtual functions was introduced so that the return types of virtual functions do not need to be identical but may be types derived from the return type in the base (original) virtual function declaration. This can be used here so that each implementation of *make_me***()** returns a pointer to the specific sub-type. The actual function must create a dynamic instance of the relevant sub-type.

We could then include the following in *Address*:

```
void Address::Make_address(
              Address*& handle,
              int       cntry)
{
  Address* model;
  // code to find the address of the
  // correct model sub-type in the
  // linked-list
  handle = model->make_me();
}
```

Expert task: implement a persistent storage mechanism for the *Address* hierarchy.

I think that is enough from me. By all means tear what I have written to shreds, but do so in writing and send it to Sean for publication. Better still, try one or all of the suggested tasks, write them up and send them in. I think you will benefit from the exercise, I know that writing this has already helped to crystallise my thoughts. For example I think that enum type is a hang over from an earlier view of solving the persistence problem.

I am sure that developing on this theme, which is based around an easily grasped abstraction of a frequently used object type, will do much to develop our general understanding of polymorphic types, just as writing a class such as *Complex* helps grasp the fundamentals of writing value based classes.

The one thing that would seriously depress me is if none of you had anything to add to the subject I have tried to open up in this article.

The Harpist

Simple classes for debugging in C++ – part I

by Roger Lever

One of the curiosities of starting a tiny project and getting side tracked for a while, is the change of direction that sometimes results. The Editor saw an earlier and very different version of this article and suggested some changes. Originally, the presentation style was based on discussing, in detail, a small but complete working program of Dr Conway's Game of Life. This also included some debug classes – to help track those elusive errors that seem to populate all code over one line long! It was supposed to be aimed at beginners, but on reflection, the level of detail given was probably too much too quickly to take in easily, or a case of "information overload". By concentrating on the destination, or finished product, the route of how to get there had been lost and, sometimes, the journey is more important than arriving. So here is a revised version showing the route to a simple debugging class.

Common problems

There are many common C++ problems which developers of all levels suffer from and probably the worst are pointer and memory errors such as:

1. Memory leaks (such as a **new** without a corresponding **delete**)
2. Deleting the same pointer again (probably corrupting the heap)
3. Wild pointers (the pointed to object no longer exists)

There are many good tools to help with these "challenges" which mainly fall into the category of post-implementation static analysers. This debug class is not intended to replace any of these tools, it is a mechanism to help with the learning curve of C++. As has been often repeated by experienced trainers: "Write your own class to understand it and then use someone elses"!

The debug class will be built up from scratch and as progress is made, more features will be included until the destination is reached. So what is the first major junction on our route? The *approach* to providing debugging facilities.

General approaches to these problems

At a source code level, there are probably two key ways of dealing with these issues:

- Use a base class *DebugObject* and derive all other classes from it
- Use a combination of smart pointers and templates

Or three, if you count:

- "Contract" programming, as evangelised by Betrand Meyer

The choice to use the *DebugObject* and an inheritance approach in this article(s) was because it appears to be easier to understand. Also inheritance has been around in C++ for longer and so is probably more widely understood. Leaving aside the Editor's remark about my last article on inheritance! [1] "Roger Lever follows up his campaign for real inheritance" :-)

Specific approach

Stop, and take a while to think in general terms about the design and then plan the tasks, where each task builds incrementally towards the objective(s).

Now that some vague ideas are floating around it is time to look at making it more concrete. So, in terms of overall objectives, in no particular order, the debug class should provide a way to:

- Catch the basic memory and pointer problems
- Be able to see which objects are in memory
- Exit from an application with debug information – not crash out
- Be able to remove debug from production code easily (like *assert*)

If an item is not on this list – the source code will be given to develop it further!

Next, precise specification of the behaviour expected is possible but it would make sense to keep it reasonably high level. So to plan the route a little more the next junctions will be:

1. Very basic debug class which will output state messages
2. Differentiating between memory allocated statically and with *new*
3. Provide some heap walking capability to "see" what's in memory
4. Output debugging information to a file
5. Provide some macro magic to automatically enable or disable debug

We have objectives, we have a basic plan, we can start!

Starting with simple classes

To place the debug class into context let's start with a simple class which outputs a few messages.

```
class Base
{
public:
  Base()
  { cout << "Base constructor\n"; }
  virtual ~Base()
  { cout << "Base destructor\n"; }
  virtual void print()
  { cout << "Base print\n"; }
};
```

The only point worth mentioning about this *Base* class is that the destructor and *print* member functions have been declared **virtual**. This allows the destructor to work correctly with derived classes and allows *print***()** to be overridden in a derived class. This use of **virtual** should be explained in some detail in any good introductory text. Now, that the *Base* class is available, further classes can be derived from it:

```
class Derived : public Base
{
public:
  Derived() : Base()
  { cout << "Derived constructor\n"; }
  virtual ~Derived()
  { cout << "Derived destructor\n"; }
  virtual void print()
  { cout << "Derived print\n"; }
};
```

Notice that the *Derived* constructor calls the *Base* one first with **:** *Base***()**.

Since *Base* only has a default constructor (one that takes no arguments) this call could be left out. However, it is useful as a visual reminder that the default *Base* constructor *is* being called if no other *Base* constructor is explicitly used. Clearly it would be necessary if a different *Base* constructor were used or required. In fact the theme of a visual reminder is also the reason that *print***()** is redeclared with **virtual** in *Derived*. Strictly speaking this is redundant and a polymorphic call to the *Derived* class *print***()** would work fine. But, even if you know that *print* is polymorphic would your successor? Of course, referring to the *Base* class *print* it would be immediately obvious to all and sundry that *Derived*'s *print* must be polymorphic too but, isn't it kinder, simpler and clearer to redeclare *print* with **virtual**?

Anyway, moving into *main***()**...

Main program and output

```
int main()
{
  cout << "Create B & D – "
       << "allocated on stack" << endl;
  Base B;
  Derived D;
  B.print();
  D.print();
  cout << "Scope rules implicitly
delete"
       << " stack item(s)" << endl;
  return 0;
}
```

Nothing to add to this except to remember to include <*iostream.h*>!

The output is not startling:

```
Create B & D – allocated on stack
Base constructor
Base constructor
Derived constructor
Base print
Derived print
Scope rules implicitly delete stack
item(s)
Derived destructor
Base destructor
Base destructor
```

As yet *print* has not been used polymorphically – it will be soon. Also, unlike heap allocated objects, *B* and *D* did not have to be explicitly deleted. The state output shows that the necessary destructors were called.

Is there an obvious way of trying to break this program? Yes. Placing *B.~Base***()** before *B.print***()** will explicitly destroy *B* before it is used and *B*'s destructor will be called again at the end. Clearly the explicit destruction using *B.~Base***()** is dangerous but this can easily happen anywhere in a slightly different form such as a function returning a reference to a local object:

```
// concat() should be returning a copy
// here not a reference
string& concat(string a, string b)
{
  string c = a + b;
```

```
  // assumes + deals with the tricky
bits
  return c;
  // returns reference to local object
}
```

The pointer or reference that tries to use *c* is in for a surprise!

What about using *print***()** polymorphically?

To use *print***()** polymorphically the following could be added to *main*:

```
  Base* ptrB = new Derived;
  // note: a Base pointer using a
Derived
  ptrB->print();
  // Derived's print() is called, not
Base
  delete ptrB;
  // Need to explicitly delete the
object
```

Life's getting more complex. The *Base* pointer can be used to invoke the *Derived print***()** but since the object has been explicitly allocated via **new** it must be explicitly deallocated with **delete**.

The additional output of this code is:

```
Base constructor
Derived constructor
Derived print
Derived destructor
Base destructor
```

Is there an obvious way of trying to break this program? Yes again. Forgetting to match the **new** with a **delete**, probably not disastrous but certainly not good practice. Moving the pointer (*ptrB*) to point to a different object and then deleting it twice. Never happen? Here is an example of just how easy it is to make such a mistake:

```
  Base* ptr1 = new Derived;
  Base* ptr2 = new Derived;
  cout << ptr1 << ' ' << ptr2 << endl;
  ptr1 = ptr2; // should be *ptr1 =
*ptr2;
  cout << ptr1 << ' ' << ptr2 << endl;
  delete ptr1;
  delete ptr2;
```

The output of this fragment is:

```
Base constructor
Derived constructor
Base constructor
Derived constructor
0x182e 0x1836
0x1836 0x1836
Derived destructor
Base destructor
Derived destructor
Base destructor
Null pointer assignment
```

It is clear from this fragment that the pointer is changed from 0x182e to 0x1836. The first object has not been deleted at all whereas the second is deleted twice, hence the *Null pointer assignment*. This gives away the fact that this code was compiled using the small memory model. (The actual addresses on your machine, for this code, are unlikely to be the same as given here.)

There are plenty of variations on this theme of problems with memory and pointers such as trying to use a deleted object...

Take arms against a sea of troubles

Control those ambitions to prevent *all* types of memory and pointer problems! The debugging class will start in a similar vein to *Base* and *Derived* – outputting state messages. A modest and entirely achievable task. A little less modest is the choice of name for this *DebugObject* class – *RNLI*. There are two reasons for this name:

1. There is a nice association with a lifeboat (Royal Naval Lifeboat Institution)
2. RNL are my initials! (nobody said it was a good reason!)

```
// start sentinel to prevent
// multiple inclusion
#ifndef RNLI_H
#define RNLI_H
// provide basic screen output i.e.,
// cout, endl
#include <iostream.h>
class RNLI
{
public:
  RNLI()
  { cout << "RNLI constructor\n"; }
  virtual ~RNLI()
  { cout << "RNLI destructor\n"; }
};
// end sentinel to prevent
// multiple inclusion
#endif
```

Using the class is as simple as changing the *Base* class to:

```
#include "rnli.h"
class Base : public RNLI
{
// as before
};
```

The output from *main* now including the heap allocated object:

```
Create B & D – allocated on stack
RNLI contructor
Base constructor
RNLI contructor
Base constructor
Derived constructor
Base print
Derived print
RNLI contructor
Base constructor
Derived constructor
```

```
Derived print
Derived destructor
Base destructor
RNLI destructor
```

```
Scope rules implicitly delete stack
item(s)
```

```
Derived destructor
Base destructor
RNLI destructor
Base destructor
RNLI destructor
```

This establishes that *RNLI* can be added into the current hierarchy (*Base⇒Derived*) very easily. What needs to be done next is to define or design the interface for *RNLI* to provide a level of useful information.

Design choices

Many small and not so small design choices need to be made for any particular class, for example:

a) Will this be a base class? (Almost mandates a virtual destructor)

b) Will polymorphic behaviour be supported or required?

c) Declare the data private, protected or public? (Public???)

d) What should the public interface include?

e) What private implementation data structures and algorithms?

f) How to communicate between classes?

And so on...

Bjarne Stroustrup's maxim [2] for class design "*a class should be minimal but complete*" is good but you need to make up your own mind as to what *exactly* that means. Designing for inheritance and polymorphism needs to be a measured response to a problem and applied with some understanding. Equally a balance needs to be maintained between providing the required functionality now and future-proofing.

So, given that, the following design choices were made:

- *RNLI* is not intended to be a base class for derived *RNLI*s
- Polymorphism does not need to be supported, except for the destructor
- The interface will only support querying of state
- The implementation will remain visible, in terms of data members
- *RNLI* is not designed to remain in "production" code

No indication is made here of implementation details such as choice of data structures, that is deliberate. The first step in the design process is to convert the "why" of requirement to the "what" of design and then shuttle between the "what" of design and the "how" of implementation. Or to quote Murray [3]:

- Designing the abstraction and designing the implementation should be two separate, but related activities
- What is *not* in the abstraction is as important as what *is* in the abstraction

In summary

We have established the "why" – avoiding obvious memory and pointer problems. We have also established the "what" in terms of the objectives and high level design choices. We now need to consider the "how". The simplest way is by literally building the class up a few functions at a time and since I like to keep things simple...

However, that will be picked up in the next issue of *Overload* as I have run out of space!

Roger Lever
rnl16616@ggr.co.uk

References

[1] *Overload 9, C++ Techniques*

[2] Addison-Wesley, "The C++ Programming Language Second Edition", Bjarne Stroustrup

[3] Addison-Wesley, "C++ Strategies and Tactics", Robert B. Murray

Pausing for thought

by Francis Glassborow

Earlier on today one of my delegates on a C++ introductory course had a problem with seeing what happened after his program returned from main. He had instrumented his constructors and destructors so that he could check that all the constructed objects had been destroyed. Unfortunately he was working in a Windowing environment that closed down the program window on completion of the program. This meant that

you had to be an exceptional speed reader to check all the destructors that were called for global and automatic objects.

The problem was how to pause a program during its clean-up at the end of its execution. Let me share a couple of our solutions and invite you to experiment with them. The first option was to create a function to be called by exit:

```
void fn () {
       cout << "Press a key.";
       char c;
       cin >> c;
       return;
}
int main() {
       atexit(fn);
       // rest of code
}
```

I'll leave you to clean up the detail, such as making sure that the prompt and input are compatible – think about it. There is also a matter of the linkage of *fn***()**. If you have any doubts, go and look it up. I know the answer, but too many programmers either trust authors (despite the name, they are not always authorities.) and you need to develop a habit of checking details. (Apologies to the real experts)

This worked for the program that we were interested in, because the relevant destructors were local objects to *main***()**. But it does not supply a general solution, because global objects are destroyed after the functions registered by *atexit* have been run. So our next 'solution' was to write a special class:

```
class Pause {
public:
  ~Pause() {
       cout << "Press a key.";
       char c;
       cin >> c;
       return;
  }
} p;
```

By placing this at the head of the client code, the last function run will be the destructor for *p*. It met our needs. There are a number of interesting points for less experienced C++ programmers. This is a simple example of a dataless object, one where it is the behaviour that interests us. I think it is close to minimalist (another common example of this usage is a *Lock* class used to lock records in a database – the constructor locks a record and the destructor unlocks it, but that usually needs some data to track which record is locked).

Another feature of class *Pause* is that it can be used in many other places. Just write:

```
{ Pause p; }
```

wherever you want you program to pause (yes I know you can set a break point with your debugger but don't spoil a simple idea). Make *Pause* a little more complicated with storage for a string, and the destructor can print out a message identifying where the program is pausing. Why is this interesting? Well, consider the order of initialisation problem for executables built from several files, each with global variables. By declaring a suitable global *Pause* object as the first line of each source code file, you can investigate the order in which the globals are being destroyed. Another aspect is that you can use a global *Pause* object to check the difference between explicit call of *exit***()** from *main***()** and using **return** *0*. Yes, that is right – there is no stack unwind in the former case – you knew that, didn't you?

Well that is all I have time for. Anyone else got any simple utility/instrumentation classes that they would like to share with us?

Francis Glassborow
francis@robinton.demon.co.uk

editor << letters;

Sean,

after reading your review I downloaded the S-CASE demo and played with it for a few hours. I had the following email exchange with MultiQuest about the product and some problems I had with it:

Keith Derrick: What price is a full version of this product for Windows 3.1 (UK Pounds Sterling please), and where in the UK can I obtain it?

MultiQuest: Single user node locked license is US $495. I don't know the current Sterling rate. When we accept international payments, for example, using MasterCard or Visa, they charge the customer at the current rate. Currently, we service all international markets directly. International delivery takes only 3 business days.

KD: When are the next 2 releases of the product due out? What are the predicted upgrade costs, and what extra functionality will they provide?

MQ: Release 3.0 is expected in the Oct/Nov time frame. We have a single unified technical support & product update program. New purchases are covered for 30 days under this program. Further support (which includes upgrades) can be purchased at approximately 25% of product cost per year ($125 for Windows node locked license). Release 3 will provide:

- Support for all Booch diagrams
- Better connection between diagrams (as you noticed, class & object diagrams are not connected right now)
- Enhanced code generation (better support for templates and latest C++ features)
- Enhanced user interface (we are switching over from XVT to MFC)

KD: Parameterised classes: how do you instantiate a template?

MQ: Template support isn't great right now. This has been given top priority for release 3.

KD: I assume that the "Abstract" check box in the cardinality section of a class specification is there to indicate that the class can not be instantiated?

MQ: That is correct.

KD: Why is this box not set (and checked) by other sections of the class specification? Surely, if I define a pure virtual operator, then the class is abstract, and hence this box should be checked automatically? Conversely, if the class has public constructors, and no pure virtuals, then it makes no sense to set this check box. Also, why can I associate an "A" annotation with a non-abstract class?

MQ: These are all good suggestions, and I have put them in our request database.

KD: If I annotate an inheritance as "virtual", is that reflected in the class specification?

MQ: Currently, none of the graphical properties (A, F, S & V triangles) are tied to the model. They are simply dumb shapes. This will be fixed in release 3. However, when you check the appropriate boxes in the class or relation specifications, they will be reflected in the code. Specifically, if you double click on an inheritance relation, you can specify that it is virtual in the specification box and this fact will be reflected in the code.

KD: There appears to be little or no integration between the class and object diagrams. It would be nice if you offered a drop down list of "known" classes in the object specification box. You could also then provide a selection of *valid* messages when annotating a relation between two objects. This is important as I often use object diagrams to "test drive" a set of interacting classes, and limiting my choice of messages to only those which is exist would make it a more realistic exercise.

MQ: As mentioned earlier this is coming...

KD: How can you declare exception specifications for class members?

MQ: The only way to do it right now, is to override the member implementation. This is done by pressing the "More..." button in the operation specification and specifying your preferred implementation. (BTW, this can be done for data members also). Again S-CASE 3.0 should understand exception handling.

KD: Your Operator specification box allows me to say it is a constructor, or destructor. It would also be nice to be able to select "Copy Constructor" which has a standard interface. If I rename the class, then I need not edit the copy constructor definition as it would change automatically a la the other constructors.

MQ: Good suggestion. I have entered it into our database.

Keith went on to say:

Sean, you asked about ROSE. I use v1 at present. I've had the demo for v2, but it didn't justify the 3-fold price increase. Rose v1 does have the integration between class and object diagrams, also STDs, but no interaction diagrams which I feel are extremely useful in clarifying the sequence of interactions.

S-CASE has the potential to quickly overtake ROSE if they provide 75% of what they promise. Also the move from XVT to MFC should provide an easier to use interface. The XVT one (also used in ROSE) seems a little primitive.

Still ROSE is better than nothing. I find it hard to visualise a set of classes without something like Booch's notation, and it's a godsend to have a CASE package to help you out.

Keith Derrick

Thankyou for that information, Keith. I too look forward to v3.0 of S-CASE after reading MultiQuest's responses to your questions. I wonder how their move from XVT to MFC will affect support for their UNIX and Mac versions?

Dear Sean,

Just a short letter this time to dot some i's and cross some t's. I think you slightly missed the point of my code that relied on **mutable**. You have sometimes suggested that I could have found out whether code worked by testing it on a compiler. In the current state of C++ this is completely useless. It may compile because the compiler is wrong, it may fail to compile because the compiler is wrong. Mental models of languages must be built on the language as written, not deduced by experimenting with compilers. We already have serious problems with ill-informed C programmers who have seriously flawed models of C. To make matters worse, they write books or publish articles in popular computing magazines. For example the regular writer on programming in Computer Shopper (UK version not the identically titled US magazine about which I know nothing other than that it exists) should be taken out and shot.

If C++ programmers/writers start doing the same thing (well they are, but don't encourage them) we are completely lost. We need a clear understanding of what C++ is supposed to do and how code is supposed to behave. That way we can shout very loudly at the many seriously broken implementations.

Now my purpose (well one purpose) in producing perfectly readable genuine C++ code that used **mutable** was precisely that after more than two years there is still no readily available compiler that supports it. When such a simple thing has not been included, what hope have we of learning to use C++, and check support for the alternative spellings of many of the operators if you want some more examples.

OK, I do not need **mutable**, **bitand** etc. in order to write C++, but the point is that implementors are not even providing such minor detail so suggesting that we use experiments on compilers to discover what the language does is a bit over the top.

Actually, I didn't suggest any such thing – I simply pointed out that it was unfair to ask people to compile and try out a piece of code that we both (all?) know will not compile!

I have said it before, C++ is a great language, I enjoy using it. My employer will not let it anywhere near any development site, not because there isn't a standard but because some of the experts seem reluctant to address criticism and mend the broken aspects of the language.

If you have any doubt that there are serious problems that need addressing just look at **namespace**, STL and name injection. Any one of those is bound to drive your average programmer screaming mad. The first doesn't seem to have been implemented correctly (indeed I gather they are still tinkering with the essential detail), the second is a brilliant idea but needs an awful lot more work as the current version is riddled with manifest errors (and I'm not talking about the typos). I am far from convinced that we should let name injection any where near the language. It seems a recipe for surprising programmers with bizarre behaviour.

Yours,

George Wendle

Perhaps you could write an article on the "manifest" errors you think are present in STL – I've been using parts of it heavily for about six months with no problems (other than continued poor support for templates from every compiler). On namespace, see "What's in a name?" elsewhere in this issue. As for name injection, you'll have to wait to see what happens in Tokyo – it's a hot topic on the agenda!

Hi Sean,

I've just enjoyed reading *Overload 9* and was interested in Kevlin's piece on the STL. I agree; it won't compile!

Keen the harness the power of STL, I tried two approaches – building the Stepanov & Lee source downloaded from the net and buying a commercial version, Modena's STL++. Eagerly compiling the Stepanov and Lee source on my Microsoft compiler gave numerous compile errors. Luckily, Kevlin came to the rescue, fixed

the 'errors' only to break the compiler which lay on its back, feet in the air, crying 'Internal compiler error, contact Bill Gates' etc. Undaunted, I turned to the STL++. More luck here, at least it built to yield a library. However, trouble with STL++ started when I tried multiple source file builds (not so abnormal?) which gave link errors. I contacted Modena who promised a patch disk in the post...

Is there an industrial-strength version of STL on an NT PC and Microsoft C++? I'd like to use STL 'at the coalface' but is it too soon?

Chris Simons

> *Well, I ended up implementing it myself for Cfront but then I've already done two other implementations of parts of it.*
>
> *I've been using STL heavily for about six months but (lack of proper) template support in various compilers is very frustrating...admittedly, the committee have put so much stuff into templates that vendors have a really hard job keeping up!*
>
> *As far as MSC++ is concerned, I think you're on a loser: Borland (and Watcom?) can handle STL. Symantec is fine on the Mac—it even ships with STL as a precompiled header (but not, unfortunately, on the PC!).*

Sean,

I came across an unfamiliar term in *Overload* the other month – could you tell me what a 'mixin' is?

Also, has *Overload* ever done a beginners' guide to exception handling? It is one of those new C++ concepts (along with templates) that I haven't managed to catch up with and I could really use a basic explanation.

Dave Midgley
100117.2522@compuserve.com

> *Mixins are hard to explain but typically you have an interface class that is implemented in two separately derived classes (some functionality in one, the rest in the other) and then the two classes are 'mixed-in' to a further derived class giving a diamond shaped inheritance hierarchy. I'll probably do something on it in a forthcoming Overload issue.*
>
> *Exception handling is something we need articles on in future issues (hint, hint, dear readers)! I can highly recommend Silicon River's "New and Emerging C++" video for a good explanation of exception handling (and STL). Contact Silicon River on 0171 317 7777.*

Dear Sean Corfield,

I subscribed to *Overload* hoping to learn something about C++, but in issue 9 I read '*A Better C*' which just states C++ is not a better C++ *[sic]* because it is not a better C++ *[sic]*. Q.E.D.

Then I read a lengthy account of how Francis Glassborow lost his dollars and all about his hotel and restaurant, this must have been a rivetting experience for him, but for his readers it must have been a bore. No doubt Richford think they are getting value for money I certainly am not.

yours sincerely,

Dr Brennig James

> *Ah, well, you can't please everyone!*

Books and Journals

Bruce Eckel and Daniel Duffy have their latest books reviewed in this issue. Barton & Nackman's much talked about "Scientific and Engineering C++" will be reviewed in *Overload 11*.

I'm looking for a reviewer for "Foundations of Visual C++ programming for Windows 95" – if you have Windows 95, VC++2.0 (or later) and a CD-ROM drive, please drop me a line.

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

Thinking in C++

reviewed by Peter Booth

Title:	Thinking in C++
Authors:	Bruce Eckel
Publisher:	Prentice-Hall
ISBN:	0-13-917709-4
Price:	£25.50
Format:	softback, 813 pages

"You can't just look at C++ as a collection of features; some of the features make no sense in isolation. You can only use the sum of the parts if you are thinking about design, not simply coding. And to understand C++ in this way, you must understand the problems with C and with programming in general" (Eckel, 1995).

"Thinking in C++" is an alternative to the crop of "How-To-Learn-C++-Without-Thinking" books that have sprouted like weeds in the Computing section of bookstores. Eckel teaches C++ for a living, is a member of the C++ Standards Committee, and writes for a range of programming journals, so is well qualified to write this text. He directs this book at those who understand C and intend to learn C++ on their own. This summed up my own situation, so I had high hopes when I first purchased "Thinking in C++". To a large extent, I have realised these hopes.

The book has a distinctive structure. It explains those features of C++ that are not a part of the C language. Eckel introduces new concepts strictly one at a time, beginning with data abstraction, classes, initialisation, and ending with multiple inheritance, exception handling, and RTTI. This avoids overwhelming the reader, but at a cost: the example code mixes C and C++ styles, i.e., using *malloc*/*calloc* for the first ten chapters. At first I found it hard to get into "Thinking in C++". Eckel's talent for clearly explaining complex features can make the subject material appear deceptively light. I had under-estimated the difficulty of learning C++ concepts, so I put the book on hold while I began a ten week course in C++ programming. When I finished the course I returned to "Thinking in C++" and found I could readily engage with it. By working slowly through the book I have gained confidence in C++.

Eckel has a clear and accessible writing style, with a real gift for explaining from the programmer's perspective. The book is visually appealing, which makes a difference, and is well indexed. It is very much a book for readers who want to know "what happens under the hood" with C++. I enjoyed the wide focus, which ranges from the strategic implications of moving to OOP, to how a compiler might implement late binding. Some readers might feel impatient with this. It was a relief to read a technical book and trust the writer's grasp of their subject. I found "Thinking in C++" more engaging than either Lippman's "C++ Primer" or Stroustrup's "C++ Programming Language". It doesn't cover all the ground that Barton & Nackman's "Scientific and Engineering C++" manages, but it fills in more of the gaps.

One criticism – finding the book's code examples on the Internet was extremely difficult. I hope that Prentice-Hall or Bruce Eckel take better care of this in future reprints. This is an excellent book, within its niche. If you are a confident C programmer who wants to work through a structured tutorial that explains how to write and, more importantly, think in C++, then you could find this book extremely useful. For me, it was money well spent.

Peter Booth
p.booth@ic.ac.uk

From Chaos to Classes

reviewed by Sean A. Corfield

Title:	From Chaos to Classes – Object-oriented Software Development in C++
Authors:	Duffy
Publisher:	McGraw-Hill

ISBN:	0-07-709118-3
Price:	£29.95
Format:	softback, 360 pages

Despite the subtitle, this is not a book about C++ nor even software development in C++. It is a strongly pragmatic book about Rumbaugh's Object Modelling Technique (OMT) methodology that uses C++ as the implementation language for a few of its examples.

Front loading

The emphasis of this book is on requirements through to design, with the claim that the iterative nature of the OO software lifecycle means that what most of us think of as "development" is really just an ongoing "evolution" of a prototype. Of the 18 short chapters, only three really cover C++ or development and the rest of the book is clearly focused on the front-end of the lifecycle. Consequently, the early chapters provide an in-depth discussion of identifying and classifying objects, analysing their relationships and designing their interactions. The discussion of object relationships in chapter 2 should raise everyone's awareness of the many different types of relationship that we need to model – see Ulrich Eisenecker's article on inheritance elsewhere in *Overload 10* for a similarly provocative discussion. Because of the focus on object relationships and effective requirements analysis, the OMT methodology itself is not even introduced until chapter 8.

OMT++

The most startling aspect of this book is the generally critical tone of exposition. OMT is presented in the context of a full software lifecycle and Duffy points out many shortcomings and suggests realistic solutions – clearly the result of practical experience with OMT. These solutions generally involve adopting a pick'n'mix approach to different methodologies, for example Jacobson's "use cases" are recommended for subdividing the problem domain to produce subsystems that can be manageably handled by OMT. Similarly, "concept maps" are introduced to reinforce the requirements analysis and filter out 'invalid concepts' that would otherwise not be discovered until later in the lifecycle using classical OMT, "event-response lists" are used as an aid to dynamic modelling and Object Flow Diagrams are used alongside the Data Flow Diagrams of classical OMT.

Throughout the book a handful of example problems are used to examine the phases of the software lifecycle, iteratively being fleshed out as requirements become clearer and analysis proceeds towards design. Partial implementations are given but after fairly thorough treatment, completing the implementations would be pretty much a mechanical process.

C--

If the strength of this book is in the focus on the front-end of the lifecycle, its greatest weakness is certainly in its presentation of C++. Although templates and exception handling are both introduced early on as being important features, their description is incomplete and both features are described as "not widely supported at the time of writing" – this in a book published in June 1995! Exception handling is treated in a particularly cursory manner with no mention of how it affects a class interface, despite repeated comments about the importance of considering exception strategies during analysis. This is particularly unfortunate given Duffy's recommendation that exception handling is written into templates – a thorny problem at the best of times for which I would have liked to have seen his solution.

Duffy hands down occasional "rules" for writing maintainable C++ which are usually vague guidelines and often unsupported – multiple inheritance is maligned for being difficult and generally unnecessary. This could be forgiven if the quality of the C++ code in the book wasn't so poor: code fragments suffer from inconsistent layout and naming conventions, occasional syntax errors and some outright poor practice, e.g., *Boolean* expressions are explicitly tested against *FALSE* and *TRUE* (the latter being particular error-prone).

Amongst the recommendations are some interestingly draconian restrictions on method complexity. Duffy states that no method should contain more than five lines of code nor contain more than one decision. This leads to an explosion of methods so it is perhaps no surprise that Duffy says classes should have "a maximum of 40" methods.

Summary

Duffy's thorough and pragmatic approach to the OO lifecycle is let down by poor illustration with inadequate C++. Although I found his style somewhat slow and leaden at times, particularly the introductory chapter, I think the detailed treatment of the example problems provides many insights into the difficult early stages of the software lifecycle. Occasional vagueness ("experience has shown", "general agreement in the literature") is offset by an extremely varied and interesting set of references at the end of each chapter and at least all of the exercises presented have model solutions given in the appendix – something I wish more books would do.

As a book on requirements capture and OOA/D, I can recommend it with the above reservations, but I'd be a little more enthusiastic if it wasn't trying to "tag along" with the C++ wave of popularity.

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

News & Product Releases

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

Microsoft Announces Visual C++ Version 4.0

September 12, 1995 – Microsoft announced today the upcoming release of Visual C++ 4.0, the latest version of the 32-bit development system for Windows 95 and Windows NT operating systems.

Key reuse features:

- *Component Gallery*, an intelligent storage and management system for OLE Controls and C++ components.
- *Custom AppWizards*, providing the ability to create or use powerful application templates.
- *MFC extensions* are dynamic link libraries that extend the Microsoft Foundation Class library to provide capabilities by deriving new custom classes from existing MFC classes.

Other features:

- *ClassView* is a background no-compile class browser seamlessly integrated with the project workspace window. ClassView allows developers to move "beyond files" by viewing and editing their code as a collection of classes.
- *Incremental Compilation and Minimal Rebuild*, to reduce the amount of time it takes to create an executable after making changes in source files. With incremental compilation, changing a source file causes only the modified functions to be recompiled, instead of the whole file. With minimal rebuild, changing a header file causes only the affected source files to be recompiled, instead of every file that includes the header.
- *Developer Studio*, which integrates multiple development tools in one central environment.
- Enhanced ease-of-use features, including emulation of Brief and Epsilon editors, DataTips that display the value of a variable or expression during debugging when the mouse pointer pauses over it, enhanced project management including sub-projects and custom build rules, and enhanced multiplatform support.

Full support for Windows 95

- Visual C++ version 4.0 is the premier development system for Windows 95. Key support features for Windows 95 in Visual C++ 4.0 include encapsulation of new Windows common controls and common dialogs in MFC 4.0, support for TCP/IP debugging under Windows 95, and support for new Windows 95 user-interface.

MFC version 4.0 class library

- Visual C++ version 4.0 includes MFC version 4.0. New with version 4.0 are support for all Win32 common controls, including rich-text edit controls; enhanced support for multi-threaded applications, and support for Data Access Objects (DAO), providing de-

velopers the power and flexibility of Microsoft's renowned Jet database engine in their C++ applications.

Expanded C++ language support

- Microsoft Visual C++ 4.0 compiler supports significant new C++ language features from the current working papers of the ANSI/ISO X3J16 committee on C++, including namespaces and run-time type information (RTTI). Additionally, Visual C++ includes Hewlett-Packard's Standard Template Library (STL).

New support for multiple platform development

- Visual C++ Cross-Development Edition for the Macintosh now includes support for Power Macintosh computers with a native PowerPC compiler. Additionally, Visual C++ for the Macintosh includes support for building OLE- and ODBC-enabled applications via MFC 4.0, a new incremental linker, and support for Win32 common controls on the Macintosh.
- Visual C++ for PowerPC provides a native toolset for PowerPC machines running the Microsoft Windows NT operating system.

Pricing and availability

- Visual C++ 4.0 on the Intel platform will be available in October, 1995.

Existing subscribers to Visual C++ will be automatically sent the new version once it is available. New customers can subscribe to the Microsoft Visual C++ Subscription 4.0 for around £389 or less plus VAT. Visual C++ subscribers receive 3 additional releases, including major releases, as they become available during the subscription year. Users of all previous versions of Microsoft Visual C++ can upgrade to Visual C++ 4.0 for around £189 or less plus VAT, or to Visual C++ Subscription 4.0 for £289 or less plus VAT. All prices are estimated retail prices, and Visual C++ 4.0 will be available from any software reseller worldwide.

- Special editions of Visual C++ 4.0 will be available in November for MIPS and Alpha-based systems, and for cross-development for the Apple Macintosh (both Motorola and PowerPC-based). Visual C++ for Windows NT PowerPC will be available later in 1995.

For more information on the Microsft Visual Tools family, visit Microsoft for Developers Only at
`http://www.microsoft.com/devonly`

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, Carthorse Lane
Woking, Surrey, GU21 4XS
john@wash.demon.co.uk

Subscriptions

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

Distribution

Mark Radford
mark@twonine.demon.co.uk

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 11* (December) must be submitted to the editor by November 6th.