ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 8

June 1995

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

C++ Public Review

The big news is that the ANSI public review has officially begun. At present, it seems likely that many other ISO member countries will also conduct public reviews of some sort – try to get involved because this is a critical time for the draft standard. In the *Draft International C++ Standard* section, you will find details of how to access the draft and how to participate in the public review process.

Living in the real world?

How "real-world" is object technology? At present, good OO design doesn't come naturally to most of us – it seems very hard to identify the right objects, partly because the "right" objects are not always the real world objects in any tangible sense. In this issue, David Davies explains one of the many OO methodologies and this highlights the necessity of looking beyond the tangible objects.

One particular topic that seems to split you all on "real world" issues is multiple inheritance and *mixin*s. The classic *is-a* relationship doesn't hold between derived and base for this sort of design so which is "right"? I recently had to design and implement a cluster of classes whose sole purpose was to model relationships. The key abstraction was a *relationship* which is certainly not a tangible object but the desired view of the data made this the easiest solution to work with: I could start with a list of all "inherits from" relationships and list the parent and child in each. The traditional way would be to have a list of all children (or even all people) and check an "inherits from" data member within each. That member would have to be a list because each child has two parents.

Wait a minute! Let's that around: would you say you inherit characteristics from both your parents? I expect you would: you'd consider some of your characteristics inherited from your mother and some inherited from your father. A clear case of multiple inheritance – what could be more "real world" than that?

The Overload Disk

As indicated in *Overload 7*, the *Overload* disk has been discontinued. However, Francis had the bright idea that I supply material to him that would otherwise have gone on the disk and he will put it on the *CVu* disk, which in turn will be placed on Demon for ftp.

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++.

In this issue Roger Lever questions whether the mainstream is ready for C++, David Davies explains the Shlaer-Mellor object-oriented analysis methodology and I continue my series about compiler writing.

The case against learning C++ right now – right or wrong?

by Roger Lever

My position

I do not exclusively use C/C++ for programming – I like to remain flexible in terms of development tools. I consider a programming language to be a tool in the same way that a writer would consider pen and paper, a typewriter or a wordprocessor tools. I prefer to use mainstream tools with established good practice which enables me to produce code that passes my 'six month test'. Revisiting code six months later it meets the following criteria:

1. It is understandable (to me!), so maintainable by a third party
2. It is well designed, clearly and concisely written
3. It performs well, in terms of speed and designed intention

4. It is forgiving in terms of unexpected conditions or actions

5. I do not want to *drastically* re-write it

Naturally, metrics can assist in quantifying these criteria but I'm happy with an intelligent guess or intuitive feel. Project managers and QA specialists need not comment!

I'll try to resist then – Ed.

C++ is maturing fast with the standardisation process now within sight of completion and all of the major pieces in place in terms of the language feature set. (There are also many job opportunities!) C++ has seen some drastic changes since the late eighties and Bjarne Stroustrup has the (probably) definitive list of events and the "whys" in his book "The Design and Evolution of C++".

I've only become really interested in C++ recently as it seems to be coming of age, although I have kept tabs on it for a number of years. However, is it the time *now* to really get to grips with C++?

Against C++

C++ was positioned as a C compatible object oriented extension with PIE (Polymorhism, Inheritance and Encapsulation). It was used and abused, and the ideas slowly evolved to build a body of wisdom of what should be considered good practice. This resulted in some idioms which one could apply (with some understanding as to why, of course) from top level design to low level code such as:

1. Inheritance – use *is-a* or *has-a* to model the solution

2. Use templates for same behaviour but different types

3. Use inheritance for same type but different behaviour

4. Avoid the surprise of bitwise copying with copy constructors

5. Do not return references to private member variables

6. Use const

7. ...

"Effective C++" by Scott Meyers was and is a good read but it is in need of an update to include the latest language additions. At that time templates and multiple inheritance were sketchy in terms of accepted wisdom (if my memory serves). Now that exception handling, RTTI (Run Time Type Identification) and the STL (Standard Template Library) are incorporated into the language, that body of wisdom has a lot of catching up to do.

Scott Meyers has just published "Effective C++ Plus" which should answer this criticism. It will be reviewed in Overload in due course – Ed.

For a developer, the changes are much more fundamental. Previously, language features were simply being abused due to ignorance and that disappears with experience and/or training. The basic mechanism of expressing a solution was to use the PIE approach and away you go. Problems regarding things like error handling at the memory allocation point were the same as usual.

That's changed. To embrace ISO C++, we need to use exception handling, to write code using a **try**/**catch** sequence, to handle the *bad_alloc* condition that a failed memory allocation gives rather than the old null pointer, to use the 'resource is acquistion' style that Bjarne Stroustrup describes in "The C++ Programming Language 2e".

Developers wanted a standard library which was portable and not to rely on vendor implementations, like Borland's BIDS (Borland Intl Data Structures). Writing these libraries in the first place is no small task for a developer; much easier to use a ready-made one and so the introduction of the STL which is vendor independent. Using the STL has advantages but it too imposes a shift in programming, in a similar way that using streams via *iostream.h* rather than *stdio.h* caused a shift.

The end result is that code written prior to the use of these features will be substantially different to code written using them. Of course, one could avoid all of that but it would be like writing C in K&R style – it works but one should be using ISO C.

Moving on from the addition of more language features to the use of existing ones, there are significant shifts in what is considered good coding and design practice. Inheritance is a good example of that: it started as a simple mechanism for code reuse and later became a mechanism for modelling the solution where the classic *is-a* and *has-a* were considered the correct way to use it. Now, following some comments in *Overload 7*, I wonder if it is about to be refined again in terms of modelling types or objects? There is a multitude of different practices in low level coding which could be held up as an example. Browsing back through *Overload* and/or *CVu* would highlight some, such as the use (or not) of friends and operator overloading.

Summary

There have been many, many changes both major and minor: what are the options? Of course, one could side-step the issues and take the "use it later" approach. Or one could go forward with the *current* perceived wisdom and assume that a major rewrite will not be required later. Or, perhaps one should wait and see?

C++ has moved on a great deal from C and it is becoming mandatory that developers receive quality training, but whilst C++ is still evolving that may be a tall order. Consequently, I don't believe C++ is ready yet for mainstream usage, or more accurately, it still needs to mature. Inevitably, it will mature and having an ISO C++ Standard will do a great deal in that direction, but in the meantime the leading edge must bleed. There will always be shock troops for the bleeding edge but the more conservative will learn from *their* mistakes and not devote too much effort to writing C++ – at least, not just yet.

Roger Lever

rnl16616@ggr.co.uk

There are certainly a lot of companies using C++ as just a "better C". Roger suggests that mainstream use of full C++ in an OO style is some way off – how "mainstream" do you feel your use of C++ is? – Ed.

OOA – The Shlaer-Mellor Approach

by David Davies

Introduction

There are many competing OOA and OOD methodologies available today, but in most respects the conceptual similarity between them outweighs any differences in implementation. One methodology may use rounded rectangles to graphically represent objects whilst another may use square edged rectangles for the same representation. It is this drive to differentiate their offerings from the rest that has lead Object Orientation methodologists to advocate the use of symbols such as clouds. In essence, all methodologies offer a means of modelling different views in order to facilitate comprehension of the problem. OOA is the process of identifying objects and their attributes, identifying the operations performed on or by each object and establishing the interfaces between objects. The fundamental concept of Object Oriented design is shown in Figure 1.

Overview of the Shlaer-Mellor methodology

One of the popular methodologies has been developed by Sally Shlaer and Stephen Mellor. It is particularly well suited to the analysis of information systems or re-engineering applications where the initial requirements are fairly well defined.

The S-M approach analyses the problem from three view points using an information model, a state model and a process model.

The information model addresses the static aspects of objects. It identifies the objects (or entities) which form the domain. Objects have attributes and relationships with other objects.

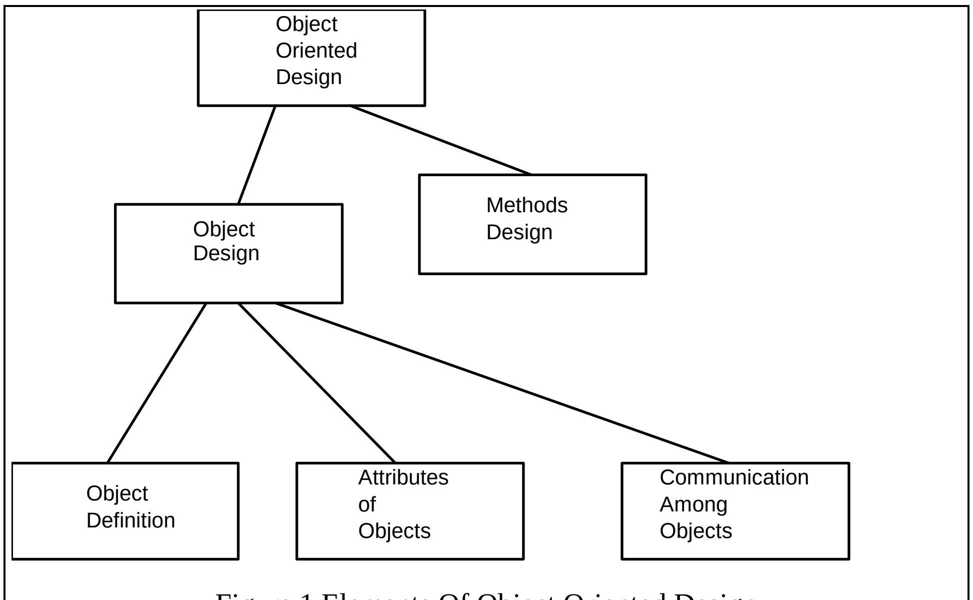


Figure 1 Elements Of Object Oriented Design

The information model comprises a table of objects and their attributes, an information structure diagram which shows the static relationships between the objects in the application domain and a description of the relationships.

The state model analyses the behaviour of objects over time. Each object and relationship may have a lifecycle – a series of events that follows a set pattern. New instances of the object in question may be created and deleted over time; the so-called ‘born and die’ lifecycle. For example, an instance of object *bank_account* is created when an account is opened and it is deleted when the account is closed. Other instances of an object continually cycle through their states e.g., in the example discussed later the car is continually going through a lifecycle of being available for hire and being hired. The outputs of the state modelling phase are state transition diagrams and/or state transition tables showing the cycle of events which affect an object, and an event list showing all the events that have been defined for all state models.

The object communication model shows how the various objects are co-ordinated. Event messages are used to synchronise the behaviour of the objects within a system. At a certain state in an object’s lifecycle an event is generated to initiate an action by another object. In a process control application the tank object may generate a *tank-full* event to its controlling valve to shut off the supply or to another object to initiate subsequent processing of the tank contents. The object communication model provides a graphical representation of the linkage between state models. The state model looked at an object on an individual basis, whereas the object communication model shows how they co-operate to implement the application.

Process models show the processing within the actions of the state model. Each action is expressed in terms of datastores and processes and the resulting diagrams are very similar to data flow diagrams used with Structured Analysis and Design methodologies. There is however one vital difference. In S-M OOA, the problem is first decomposed into objects, then into actions and finally into processes within an action, giving a single flat dataflow diagram for each action called an action dataflow diagram. This contrasts with Structured Analysis and Design methodologies where the problem is expressed as an hierarchical set of dataflow diagrams.

These views roughly map onto the structure shown in Figure 1. The information model addresses object definition and attributes of objects. The state model and object communication models cover the communication between objects, and the process model addresses the issues of methods design.

Hire car example

As an example of an application of the Shlaer Mellor methodology the requirements for a car hire reservation system will be analysed. The brief for the system is:

The system will accept reservations from customers for a car on a date for a specific number of days. At the end of their hire period customers will settle the account. The cars fall in to various hire groups and the hirer can hire the car on a per mile or unlimited mileage basis. The system shall determine the least cost hire basis at the end of the hire period taking into account period of hire and mileage driven.

The Information Model

Identifying the objects

A vital first step in OOA is to identify all the objects that are pertinent to the application domain being analysed. The scope of the proposed application must first be bounded so that objects relevant to the application are defined. Although in a large application identifying the objects is no mean task, correct identification is critical in ensuring a good quality analysis. There are several methods of identifying objects from a specification. Techniques such as using the nouns in the specification documents to indicate suitable objects or using the entities to be modelled as a basis for deciding on the relevant objects and classes. These techniques form a good starting point for the identification of objects. As an aid to identification, the Shlaer-Mellor method recommends that objects are classified into several major categories such as:

- Tangible objects
  A tangible object is an abstraction of a real world object, e.g., a computer.
- Role objects
  Role objects represent the purpose or task of an individual, a piece of equipment or or-

ganisation. For example, a person may be a cashier or piece of equipment may be controlling part of a process plant.

- Incident objects
  Incident objects represent occurrences or events in the problem domain that the application has to be aware of.
- Interaction objects
  Interaction objects generally have a transaction or contract aspect and are related to two or more other objects in the model.
- Specification objects
  Specification objects represent rules, standards or quality criteria that bound system behaviour.

The Shlaer-Mellor method provides a set of refinement criteria to assist in identifying valid objects and rejecting invalid ones. Every object should pass all the tests in order to be considered a valid object.

- Uniformity test
  Each object instance must have the same set of characteristics and be subject to the same rules.
- Attribute test
  An object must be more than a name, it must have associated attributes, e.g., a person (object) has a name, National Insurance number and domicile (attributes).
- Singularity test
  An object should refer to only one entity.
- More-than-a-list test
  The object description must not be merely a list of instances.

As a first pass (and a lot of OOA is iterative where the model is refined to reflect the analyst's increased understanding of the problem) a initial list of objects is produced. The two obvious ones are *Customer* and *Hire Car*. A customer can make many hirings and cars can be hired to many hirers. As explained in more detail in the section on relationships, a many-to-many relationship (between hirings and cars) can best be represented by using an associated object; in this case, reservation. For the car hire example the objects initially identified and validated are shown in Table 1.

The list may be updated as the analysis proceeds, but this list is sufficient for initial analysis of the car hire application.

Ascertain the attributes

These four objects shown in Table 1 will form the basis of the next step in the analysis which is to ascertain the attributes associated with each of these objects. Shlaer-Mellor define an attribute as 'an abstraction of a single characteristic possessed by all entities that are themselves abstracted as an object'.

According to Shlaer-Mellor attributes can be classified into three different types:

- Descriptive attributes
  Descriptive attributes provide facts intrinsic to each instance of the object.
- Naming attributes
  Naming attributes are used to name or label instances.
- Referential attributes
  Referential attributes are used to tie an instance of an object to an instance of another.

Each attribute should be briefly amplified. A few sentences is usually sufficient.

Shlaer-Mellor provide four criteria for the normalisation of attributes. Attributes should be:

- Atomic.
  That is have no internal structure. Ford Cortina would not pass the test as it is comprises make and model.
- One and only one value per attribute.
  An instance must have only one value for

Object types		**Object tests**			
Object	***Category***	***Uniform***	***Attribute***	***Singularity***	***List***
Customer	tangible	pass	pass	pass	pass
Car	tangible	pass	pass	pass	pass
Reservation	interaction	pass	pass	pass	pass
Account	incident	pass	pass	pass	pass

Table 1

Customer	***Car***	***Reservation***	***Account***
* customer ID	* registration #	* hire ref #	* account ref #
name	manufacturer	registration # (R)	hire ref # (R)
street	model #	customer ID (R)	amount
town	hire group	period of hire	CDW
post code	mileage at hire start		mileage charge
driver’s licence #	mileage at hire end		paid/not paid

Table 2 Car Hire Application: Objects and Attributes

each attribute. Null values or multiple values are not permitted.

- Characteristic of entire object
  When an object has a compound identifier every attribute that is not part of the identifier is a characteristic of the entire object.
- Represent characteristic of instance
  Each attribute that is not part of an identifier represents a characteristic of the instance named by the identifier. That is the value of the mileage attribute is only applicable to the car referred to in the identifier

For those of you that have done relational database design, the rules will be familiar. As in reducing database tables to their third normal form the aim of the exercise is to eliminate redundancy in the attributes.

One or more attributes are used to identify specific instances of an object. The particular attribute used to tag an object is called an identifier. One of the attributes for the hire car object is registration number. This will uniquely distinguish an instance of one particular car in the fleet of hire cars and so can be used as an identifier. Shlaer-Mellor recommend that the identifier is distinguished by a prefix such as ‘*’.

The referential attribute, which in relational database parlance is a foreign key, is annotated with ‘(R)’. In table 2 the account object has a referential attribute hire ref # to link it to the reservation object.

For the car hire example, attributes for the objects can be defined as shown in Table 2.

Note that if objects containing similar characteristics are identified, they can be grouped into super- and sub-classes. A super-class contains all the attributes that are common to a group. For example a super-class *road vehicle* would contain attributes that are common to, say, *car*, *lorry* and *bus*. In a similar way class *road vehicle* may be a member of super-class *land transport* along with class *rail transport*.

Relationships between objects

The next stage in the development of the information model is to identify the static relationships between objects. The Information Structure Diagram captures the various relationships between objects in the problem domain in an easy to understand diagrammatic representation.

Representation of relationships

In an Information Structure Diagram, objects are depicted by square boxes and a relationship between two objects is indicated by a line joining them, see Figure 2. The line is annotated by two verb phases describing the relationship between the objects from the viewpoint of each object. For example if the two objects where *Dog* and *Owner* the relationship would be *Owner* “owns” *Dog* from the *Owner* viewpoint of *Dog* “is owned by” *Owner* from object *Dog* viewpoint. Relationships can be one-to-one, one-to-many or many-to-many and can be conditional. In the *Dog-Owner* example the relationship is a one-to-many as an owner can have many dogs. However taking the population as a whole only some percentage would be dog owners so if a person to dog relationship was being considered, then a conditional one-to-many relationship would be required as not everybody owns a dog. See Figure 2.

A many-to-many relationship can be represented either by using two one-to-many relationships or by using an associative object that contains references to identifiers in each of the participating instances. For example a many-to-many relationship arises from a actor-role instance. An actor can star in many films and a film has many actors. The relationship can be formalised by using object *Appears* in a one-to-many relationship with both *Actor* and *Film* or an associative object *Appears* that contains references to both sides of the relationship. See Figure 3 below. Shlaer-Mellor favour the associative approach.

Information Structure Diagram

After all the objects and their relationships have been identified the Information Structure diagram can be drawn. In some respects the Information Structure Diagram is similar to the Entity Relationship Diagram which is generally used for developing relational databases.

Figure 4 shows the Information Structure Diagram for the car hire application. The many-to-many relationship between customer and car is shown as an associative relationship with reservation. A conditional one-to-one relationship arises between reservation and account. This caters for the situation when a reservation is made but subsequently cancelled.

The State Models

State models provide the second viewpoint that is created when following the Shlaer-Mellor methodology. State models formalise object life cycles and relationships by constructing a lifecycle diagram for each (nontrivial) object in the information model. The object lifecycle shows the behaviour of objects over time. Each object and relationship may have a lifecycle – a series of events that follows a set pattern. The object instance moves from state to state by means of events which trigger the move to the next state in the lifecycle.

State Transition Diagrams

Lifecycles are expressed as State Transition Diagrams. Each event in the lifecycle effects one or more actions at each state of the lifecycle. Objects do not change state until all the actions for that state are completed. Diagrammatically, states in the lifecycle are represented by boxes and the line linking adjacent boxes in the lifecycle is annotated with a description of the event which triggers the movement between the two

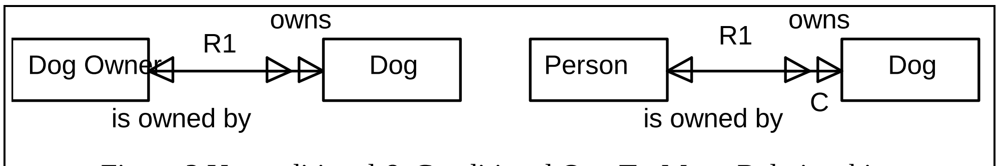


Figure 2 Unconditional & Conditional One-To-Many Relationships

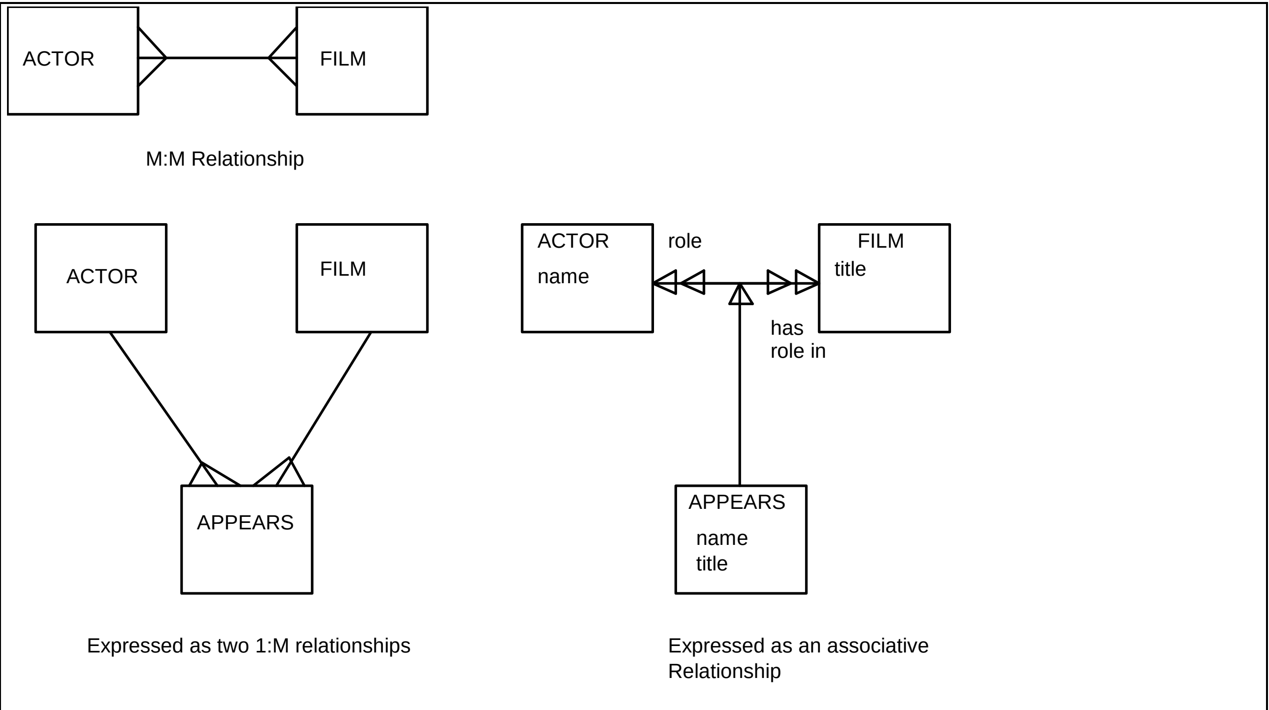


Figure 3 Many-To-Many Relationships

states. The STD should also contain a description of the actions performed at each state. However, practice has indicated that the diagrams can become very cluttered so generally the action description is held separately and just a cross reference is placed on the STD. Not having the action descriptions on the STD also means that there are no restrictions on the amount of text that can be associated with a state, although verbosity is not to be encouraged.

State Transition Diagrams need not be produced for all objects in the problem domain, only those that have complex behaviour and so require modelling in order to facilitate understanding of the issues involved.

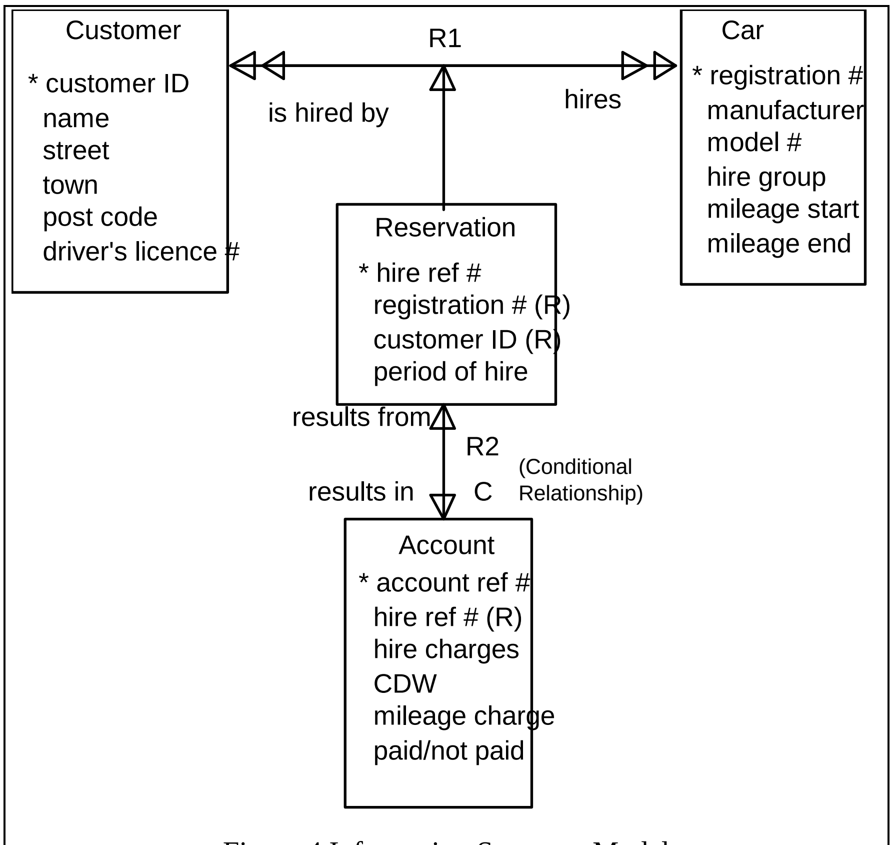


Figure 4 Information Structure Model

Alternatively, the object lifecycle can be represented by a State Transition Table. In a state transition table (STT) each row represents one of all possible states of the state model and the columns list all the events that cause a transition to the next state.

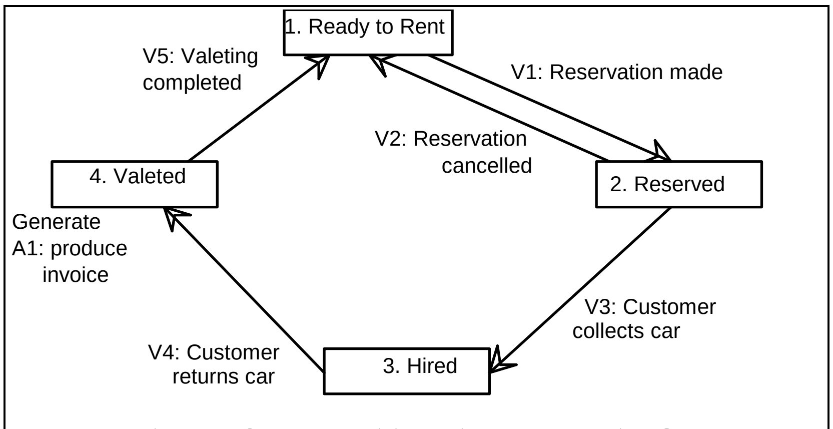


Figure 5 State Transition Diagram For Hire Car

The object *Car* has the lifecycle of “available for hire”, “reserved”, “hired”, and “valeted after use”. It is then “available for hire” again. The reservation can be cancelled at any time before the hire begins. The State Transition Table is shown in Table 3 and the State Transition Diagram for the car object is shown in Figure 5. The initial state is “available for hire” and the event of it being earmarked for a customer moves it to state of “reserved”. The event of the customer using the car triggers the move to the state “hired”. The event of the hirer returning the car triggers the transition to being ready for valeting and at the completion of this activity the car again becomes “available for hire”.

	V1: reservation made	V2: reservation cancelled	V3: hire car collected	V4: hire car returned	V5: valeting complete
1. Ready to rent	2	na	na	na	na
2. Reserved	na	1	3	na	na
3. Hired	na	na	na	4	na
4. Valeting required	na	na	na	na	1

Table 3 State Transition Table for Hire Car lifecycle

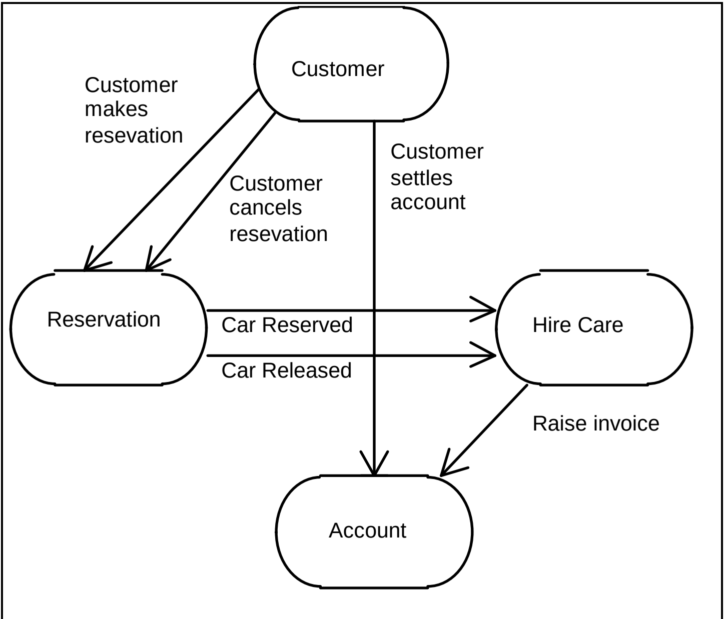


Figure 6 Object Communication Model

In a similar manner the customer object has a similar lifecycle. The customer makes reservation, incurs charges through using the hire car, and pays the account after returning the vehicle at the end of the hire period.

For all the objects identified in the application, an event list is produced showing all the events which trigger the transition from one state to another in the STD.

Object Communication Model

So far, the analysis has concentrated on individual objects and their attributes. The next stage is to consider how the objects interact to form the overall system. The Object Communication Model shows event communication between state models and external entities and objects in other systems.

By convention, the OCM is laid out with the most powerful and knowledgeable object at the top of the diagram with the more limited objects beneath. This provides a rough hierarchical layering of objects. In a typical process control application typical objects identified might be: *Batch*, *Pump*, *Tank* and *Valve*, where *Batch* represents a volume of the product being produced by the plant e.g., paint. *Pump*, *Tank* and *Valve* represent the physical objects making up the plant. In this application the *Batch* process would be the most knowledgeable about the process required to produce the paint and so it would be placed at the top of the OCM diagram. Underneath this would be the middle layer of objects comprising objects relating to sub-operations like transferring the contents of one tank to another or useful configurations of lower objects such as a number of valves, pumps and pipes to form a path. The lowest level would be the objects that directly control external entities, such as the valves and pumps and would be placed at the bottom of the diagram.

The most knowledgeable object has also been termed the “actor”. Middle layer objects whose function is to relay events to other (lower level) objects are termed “agents”, whilst the lowest level objects which generally communicate with physical objects are termed “servants”.

As stated above, objects communicate through events. An event can either be generated internally by a state model within the system, or externally from an entity outside the system being modelled. At a high level such entities may be other systems or operators which provide commands to the system. At the lower levels, typical external entities are pumps, valves etc. Events can be classified as either solicited or unsolicited. An unsolicited event is one fro which the system has no prior knowledge of its timing. It has not occurred in response to some previous action of the system. Conversely a solicited event is one that is expected by the system in response to an event generated by the system. An example of an unsolicited event would be a customer making a reservation hire a car. The system ‘knows’ what to do when such an event occurs but cannot ascertain when such an event will occur.

The recommended procedure for building an OCM is to extract the entries from the event list where the source and destination are different. An event list is usually generated as part of the object lifecycle analysis activity. The event list for the car hire application would contain entries for the car, customer, reservation and account objects. A simplified OCM is shown in Figure 6.

The Process Models

With reference to Figure 1, the last stage in the analysis is Methods Design. This covers the computational processes required to implement the required transforms within the application. At each state of a state model actions are performed. Actions are provided with event data by the event that caused the transition to the current state.

Action Data Flow Diagrams

Action Data Flow Diagrams are Shlaer-Mellor's way of describing the processing carried out to implement actions and are the main means of documenting the process model. Persistent data is held in the datastores. In the car hire example, details of the tariff structure would be held in a datastore.

An event that initiates action within a state model is depicted on the ADFD as a dataflow without a source. The event dataflow is labelled with the attributes that are required by the process. This can be seen in Figure 7 as the dataflow leading into process ACC1. The account details cannot be calculated until the customer returns the car and the mileage and period of hire determined.

To support the ADFD, process descriptions are also produced. These expand on the details of the processing required at each process. For the example in question, the process description of process ACC1 could be:

1. compare the cost of hire based on a daily rate and a per mile basis with the cost of unlimited mile rate for the hire period.
2. calculate the cost of hire based on the cheaper option.
3. produce invoice.

Code can be now be produced based on the three models generated in the analysis, but that, as they say, is another story.

Conclusion

This article has attempted to provide a high level overview of the Shlaer-Mellor methodology. For a more in-depth understanding of their approach I recommend studying their two books:

- OBJECT LIFECYCLES Modelling the World in States, Prentice-Hall 1992 ISBN 0-13-629940-7
- OBJECT-ORIENTED SYSTEMS ANALYSIS Modelling the World in Data, Prentice-Hall 1988 ISBN 0-13-629023-X

David Davies

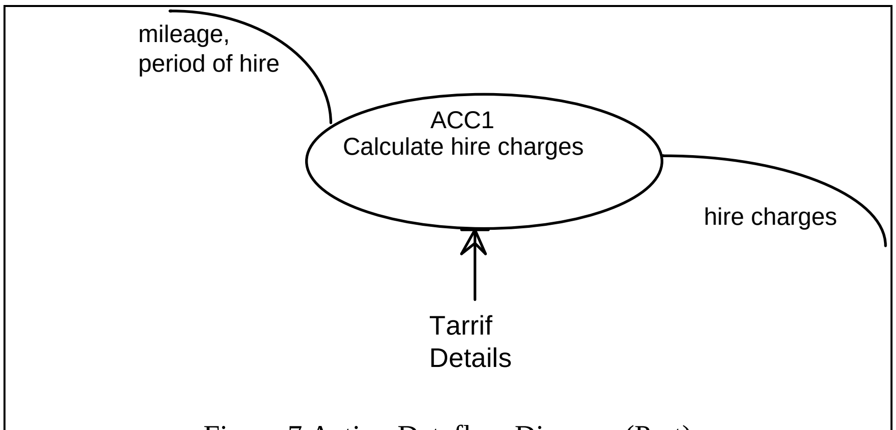


Figure 7 Action Dataflow Diagram (Part)

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

by Sean A. Corfield

The story so far...

When I started this series in *Overload 5,* I intended to show you what makes a compiler tick and how you design and build such a beast in C++. I was going to illustrate this with snippets of Programming Research code and discussion of some of the "bad practice" issues that our *QA C++* product detects. In future articles in this series, I may yet do that, but much has happened since I wrote that introductory article over a year ago and I want to digress somewhat.

A little diversion

I've been using C++ for about three and a half years and have been involved with the standardisation process all that time, initially attending BSI panel meetings then ISO meetings and for the last 18 months ANSI meetings as well. When my company first looked at C++, we had about 200K lines of C in our products – much of it K&R C but being migrated to ISO C by virtue of our in-house coding standards being automatically applied. Our decision to adopt C++ for future development meant that we would have to deal with mixed language development because we intended to reuse many of our generic library components – all written in C. My long-term plan was to migrate all the C code to C++ as I felt this would make maintenance easier, so we reviewed our coding standards for C to outlaw all incompatibilities with C++ and began to incrementally apply them. By the time I started writing this series, we had around 50K lines of C++, and about 60K lines of our C code was compilable as either C or C++. As I write this second instalment, one year on, the balance has completely changed: we have only about 100K lines of pure C code left and about 140K lines of C++.

Why am I telling you this? I mentioned our generic library components above. These are not container classes and so on but reusable code that deals with command-line argument processing, configuration and message files, data filters and error databases. We have been able to reuse most of our GUI code for three of our four products. These are all “obvious” components in products like ours and the result is that each of our language analysis products comprise a parsing engine – typically 40K lines – and support and GUI code totalling a further 50K lines. For three products, that makes 3 x 40K + 50K = 190K lines and it also means that the parsing engine is far and away the greatest contribution to any new product we plan. So we’ve been looking at parsing abstractions in order to make it easier to build new engines...

A typical compiler

At the heart of a typical compiler are three things: a lexer, a parser and a symbol table. In order to make compiler writing easier, tools were developed that can generate lexers and parsers from symbol and grammar descriptions. UNIX developers know these as **lex** and **yacc** supplied with every system. There are many variants (e.g., **flex**, **byacc**) and PC versions are available too – recently, C++ wrappers have also appeared. In essence, they all work in the same way: you write a description of a symbol, or sentence in the language, and **lex**, or **yacc**, produces code that “accepts” it.

```
/* some tokens using lex: */
IDENT: [A-Za-z_][A-Za-z0-9_]*
NUMBER: [0-9]+
%%
{IDENT}             return IDENT;
{NUMBER}     return NUMBER;
"+"          return PLUS;
"-"          return MINUS;
"*"          return TIMES;
"/"          return DIVIDE;

/* an expression using yacc: */
%token
IDENT NUMBER PLUS MINUS TIMES DIVIDE
%%
expr: factor addop factor ;
factor: term mulop term ;
term: IDENT | NUMBER ;
addop: PLUS | MINUS ;
mulop: TIMES | DIVIDE ;
```

The error-handling in both of these tools is legendarily poor and not all languages easily fit the mould of **yacc**’s restricted grammar description (e.g., **typedef** in C means that the lexer has to be smart enough to tell the parser whether an identifier symbol is a type name or not). The effort involved in making languages like C++ or FORTRAN “yacc-able” is huge. For instance, FORTRAN has no keywords:

```
      INTEGER IF(100)
      I=1
C     assign to element of array
C     called if:
      IF(I)=2
C     if statement:
      IF(I) GOTO 10
```

and it allows non-significant whitespace in identifiers:

```
C     start of DO-loop:
      DO 10 I = 1,3
C     assign 1.3 to variable
C     called do10i:
      DO 10 I = 1.3
```

In C++, deciding whether a sequence of tokens is a declaration or an expression can involve an arbitrary amount of lookahead.

In our parsing engines, we use a mix of yacc-like table-driven parsers and hand-crafted recursive descent or state-transition parsers (I’ll come back to these terms later in the series). For lexical analysis, we tend to hand-craft because we usually want to add a lot of checks into the lexer to support coding standards (again, more on this later in the series).

What about reusability?

Since the grammar is different for every language, you clearly cannot reuse the parser. Lexical structure is often similar, so we may make some progress there and basic symbol table operations have enough commonality that quite a lot of reuse should be possible. Again, these are “obvious” candidates for reuse because they are fairly concrete. The reason that I held this column over from *Overload 7* is because I want to look beyond the obvious candidates for some more abstract ones: the sort of candidates touched on by the columns on multiple inheritance and object relationships.

Back on track

In *Overload 5*, I outlined how a C++ compiler worked, in terms of the phases of translation and said that part II would take a slightly closer look at phases 1 to 4. Before diving into the mechanics of each phase, let's take a step back and look at what a phase does.

Each phase is a mapping – in particular, phase three is a mapping from a stream of (internal) characters to a stream of tokens. Later phases map streams of tokens to different streams of tokens, earlier phases map streams of characters to different streams of characters. Perhaps we could say:

```
class Lexer
: public Mapping<char, pptoken> ...
```

But a phase is also a source of characters or tokens or...

```
class Lexer
: public Source<pptoken> ...
```

What operations does a phase support? You can obviously "get" a character or token...

```
class Lexer ... {
public:
      pptoken get();
};
```

But since we saw that a phase is a source, we might expect *get***()** to be inherited from the base class *source<pptoken>*.

The client of each phase is the next phase in the sequence – such clients will either be hard-coded (statically bound) or instantiated at run-time (dynamically bound). The former approach might look like:

```
class Preprocessor .... {
private:
      Lexer  source;
};
```

whereas the latter approach might look like:

```
class Preprocessor ... {
public:
      Preprocessor(Lexer* source)
      : lexer(source) ...
      ...
private:
      Lexer* lexer;
};
```

or more realistically:

```
class Preprocessor ... {
public:
      Preprocessor(Source<pptoken>*
                          source)
      : lexer(source) ...
      ...
private:
      Source<pptoken>*    lexer;
};
```

Note that any source of *pptoken*s is an acceptable argument to *Preprocessor* – it doesn't have to be a phase, i.e., *Preprocessor* does not need access to any of the mapping machinery in its argument.

Finally, we would want to explicitly write out the abstraction of "phase":

```
template<class T>
class Source { ... };

template<class From, class To>
class Mapping { ... };

template<class From, class To>
class Phase
: public Source<To>,
  public Mapping<From, To> { ... };

class Lexer
: public Phase<char, pptoken> { ... };

class Preprocessor
: public Phase<pptoken, pptoken> { ...
};
```

I'm going to stop there for now. Elsewhere in this issue, various authors discuss multiple inheritance and **virtual**. Since both of those play a big part in what comes next, I'd like you all to think about the code above. As an exercise, you might like to try to write the interfaces for the base classes *Source* and *Mapping* – what services do they provide and what initial information do they need?

Sean A. Corfield

Development Group Manager

sean_corfield@prqa.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!

As noted in the Editorial column, the ANSI public review has begun – information about obtaining the draft and taking part in the reviews is given below. We also look at a proposal from Kevlin Henney.

C++ – the official UK site
maintained by Steve Rumsby

Steve Rumsby works for the Mathematics Institute at Warwick University. He is also an active member of the BSI C++ panel, regularly representing the UK at the joint ISO/ANSI standards meetings. He also maintains the primary UK site for information about C++ and the standardisation process in particular.

If you have a Web browser, go to

```
http://www.maths.warwick.ac.uk/c++/
```

which provides links to, amongst other things, the Standard Template Library and the Virtual C++ Library. It also provides a link to a “public review” page which will give information on how to participate in the reviews being conducted by various countries including UK and USA.

Members of the BSI C++ panel (IST/5/-/21) can access the ISO and BSI archives from here, including minutes of meetings, working papers and proposals. To find out about joining the BSI panel, send email to:

```
demorgan@parallax.demon.co.uk
```

Richard DeMorgan is the convenor of the BSI C++ panel.

The public review version of the draft C++ standard can be found in both PostScript and Adobe PDF formats at

```
ftp.maths.warwick.ac.uk:/pub/c++/std/WP/
```

but be warned that the draft is over 700 pages long when printed!

Steve Rumsby

steve@maths.warwick.ac.uk

cv-qualified constructors
by Kevlin Henney

Abstract

There is currently no mechanism for a constructor to be aware, and so take advantage, of an object’s cv-qualification. In a number of cases, such as the use of proxy objects that hold references to others and behave directly on their behalf, it is not possible to ensure statically with one class that a well defined system has been constructed.

This paper proposes that cv-qualification is defined for constructors.

Rationale, syntax and affected parts of the working paper are covered.

Introduction

The next three sections of this paper look at some particular problem areas that the introduction of cv-qualified constructors would address. The section following this considers alternative approaches and their shortcomings. The proposed syntax is used all the way through. It is explained in detail in the section following discussion of the alternatives.

The rationale for this proposed change is explained throughout. The proposal is summarised in terms of changes to the working paper at the end.

Optimising construction

Objects may be statically or dynamically created with cv-qualification, e.g.,

```
static volatile void* const port =
              (volatile void *)
0xfeedf00d;
auto const string message = argv[1];
const transform*  skewer =
               new const transform(dx,
dy);
```

If such an object wishes to take advantage of its qualification it has no means of doing so at construction time. In contrast, for the rest of its lifetime its cv-qualification, or the cv-qualification of the access path to that object, may be deduced from which of a set of member functions overloaded only on cv-qualification are called. This is somewhat after the fact of construction, where a decision based on cv-qualification could alter the particular use of runtime resources for the object.

Additional cv-qualification of constructors would allow separate strategies for construction to be adopted statically. For instance, not all member date is necessarily used in a **const** object: a less complex construction is possible:

```
class RecoverableResource
{
public:
      RecoverableResource()
      : change_log(log_name) {}
      const RecoverableResource()
      : change_log() {}
      // no changes possible on a const
```

```
        // object, so an open change log
is      // not required
        ...
};
```

To be effective, this proposal does not require the introduction of cv-qualified destructors. That would require that the language definition ensures each object remembers how it was constructed so that the correct destructor may be called. This is trivial for **auto** and **static** objects, but additional support is needed for deletion of dynamically allocated objects. Where special actions are required on destruction for actions taken in a cv-qualified constructor, a programming solution – such as using a flag set in the constructor – is simpler than placing additional burden on the language and the run time system. For most such optimisations the normal destructor action will be adequate and harmless, e.g., deleting a null pointer rather than an allocated pointer or closing an unopened file.

The latter may actually be catastrophic – Ed.

A class may use cv-qualified constructors to impose certain restrictions. For instance, non-**const** objects cannot be created of a class with only **const** constructors. More general uses of this technique are explored in the next two sections.

Object wrappers

Wrapper classes are often used to apply a high-level interface to a low level type. This may take the form of either a fully fledged abstraction that manages the type, or a simple convenience layer into which the objects of the low level type are passed. Consider the following code fragment for a class that provides a number of standard string operations on a given null terminated string:

```
class string_alias
{
public:
        string_alias(char *str)
        : wrapped(str) {}
        ...
        char operator[](size_t pos) const
        { return wrapped[pos]; }
        char& operator[](size_t pos)
        { return wrapped[pos]; }
        size_t length() const
        { return strlen(wrapped); }
        ...
private:
        char* wrapped;
};
```

Looking at the constructor provided, only non-**const** strings may be aliased. A programmer looking to use this class for a **const** string is forced to cast away its **const**-ness:

```
bool check(const char *filename)
{
        string_alias convenience(

        const_cast<char*>(filename));
        ...
}
```

The cast in this case not only casts away **const**-ness to allow construction: the appropriate type checking, and the guarantees that go with it, are also lost. The *string_alias* object created will now permit non-**const** operations on a **const** string, and thus introduce unwanted and potentially undefined run time behaviour. The designer of the *string_alias* class might chose to make the class more convenient to use by providing a weaker constructor which performs the cast itself:

```
class string_alias
{
public:
        string_alias(const char* str)
        : wrapped(const_cast<char*>(str))
{}
        ...
};
```

This is easier for users of the class, but it is now harder to track down strange behaviour. The cause of any problem is effectively hidden:

```
bool check(const char* filename)
{
        string_alias
convenience(filename);
        ...
        // modify filename via
convenience!
}
```

Introducing a constructor differentiated on **const** would provide a safe **const**-preserving route through the code:

```
class string_alias
{
public:
        const string_alias(const char*
str)
        : wrapped(const_cast<char*>(str))
{}
        string_alias(char* str)
        : wrapped(str) {}
        ...
};
```

The programmer can now guarantee that only **const** operations are performed on aliased **const** strings:

```
const char* const_str = ...;
string_alias illegal(const_str);
        // not legal because illegal is
non-
        // const and the only non-const
```

```
      // constructor requires char* arg
const string_alias legal(const_str);
      // legal: const constructor
accepts
      // const char*
```

Proxy classes

This problem is not restricted to wrapping up low-level types. The method of viewing one object through another will always beg the question of how the cv-qualification of the target can be reflected in the proxy (often, however, such issues are swept aside and ignored). Consider the following classes that represent a string and sliced views of it. The solution presented here uses the proposed cv-qualification for constructors:

```
class full_string;
class sub_string
{
public:
      const sub_string(const
full_string&,
                size_t from, size_t
size);
      sub_string(full_string&,
                size_t from, size_t
size);
      const sub_string(const
sub_string&);
      sub_string(sub_string&);
      ...
private:
      full_string& target;
      size_t start, count;
};

class full_string
{
public:
      ...
      const sub_string operator()(
          size_t from, size_t size)
const;
      sub_string operator()(
          size_t from, size_t size);
      ...
};
```

The reader is invited to consider where casts would have to be inserted in the class implementation or a class user's code if cv-qualified constructors were not present. Also under consideration is the reliability and maintainability of such code, in particular the scope for introducing undefined behaviour.

Alternative approaches

Top-level cv-qualification is ignored by *typeid*, and so this method of inspection is not available to a constructing object. That is, the first type of example cannot be implemented in a single class without additional dummy arguments for a constructor. If such a solution is adopted there is no way to enforce the correct construction of cv-qualified objects.

Similarly, the other examples illustrated that it is not possible to get the required behaviour in a secure way from a single class. Adding an extra class might at first sight appear to be the solution to some of these problems.

Patterning the creation of proxies after the STL container classes, where a container may return a normal iterator or a **const** iterator dependent on the **const** access path to the container, has some initial appeal. However, iterators represent a level of indirection not present in the examples chosen: the cv-qualification of the iterator describes the cv-qualification of the iterator itself and not the referenced container, the **const**-ness of which is described by the actual class of the iterator (plain or **const**). This is not the case with objects that are acting in some way like references rather than pointers.

To preserve expected substitutability the non-**const** class would also have to be derived from the **const** version. It is possible that in some cases an additional protected constructor would have to be added to bypass the normal construction of the base or to actually implement the construction of the non-**const** derived class. This adds significant complexity and, potentially, insecurity in the long term.

Templates represent an alternative method of generalisation. However, they have nothing to offer to this discussion: there is no way to select on the cv-qualification of a template parameter. As such, issues like substitutability cannot be tackled.

The reason for dividing one class into two is based solely on the **const**-ness of construction alone: in all other respects the roles are already partitioned within a single class by cv-qualification of member functions. C++ has a sound and regular method for specifying cv-qualification of objects at creation (the recent change to allow a cv-qualifier in a new expression serves to illustrate the need and desire for a regular approach) and for specifying member function access throughout the object's lifetime. An extension of cv-qualification to constructors would allow additional type safety and expressive power.

Syntax and rules

A constructor selected on cv-qualification is only effective if it assumes that the qualification is a minimum requirement of the object under construction. For example, an unqualified constructor may be used to construct any kind of object but a **const** qualified constructor may only be used to construct **const** and **const volatile** objects. This is fully compatible with the status quo, where unqualified constructors are used to construct all objects. Adoption of this proposal would break no existing code.

The syntax itself, where the cv-qualification precedes the constructor name, has been chosen to reflect the syntax used in the declaration of the object or the equivalent new expression:

```
class X
{
public:
       const X();
       X();
       ...
};

const X a;
const X* p = new const X;
```

It is also important that this syntax differs from the cv-qualification for ordinary non-static member functions. The semantics are quite different: a **const** constructor may only be called to construct a **const** object, but a non-**const** constructor may be called to construct any object; a **const** member function need not be called on a **const** object, but a non-**const** member may not be called on a **const** object. Using a single syntax to express two quite separate ideas would lead to confusion, hence the form chosen here.

The cv-qualification of the object under construction acts as the tie breaker for overloading, e.g.,

```
const X x; // const X() invoked
```

If there is any remaining ambiguity the construction is ill formed, e.g.,

```
class Y
{
public:
       const Y();
       volatile Y();
       ...
};

const volatile Y y; // error
```

The syntax for constructor definition simply prefixes the plain definition with the qualifier, e.g.,

```
const Y::Y()
{
...
}
```

Kevlin Henney

Westinghouse Systems Ltd

kevlin@wslint.demon.co.uk

Since the committee have tackled cv-qualification of objects many times with several proposals already being rejected, I'd like to think that Kevlin's proposal will be given serious consideration. It will be interesting to see whether this issue comes up in the various public reviews. – Ed.

namespace – a short exposé

by Sean A. Corfield

So far, very few compilers support namespaces. Metaware's is, I believe, the only commercially available compiler at present although several vendors are working on them. This means that we have no experience of working with them at all. I shall present a few short code examples showing how the draft standard says they will work and ask you for your comments.

```
namespace A {
       int j;
}
void f()
{
       j = 1;  // error: no j in scope
}
void g()
{
       using namespace A;
       j = 1;  // fine: finds A::j
}
```

Even without explaining the rules to you, this is probably intuitive. Let's look at a more complex example:

```
namespace A {
       int j;
}
int j;
void f()
{
       j = 1;  // fine: finds global ::j
}
void g()
{
       using namespace A;
       j = 1;  // ambiguous: ::j or
A::j?
}
```

Does that surprise you? Let me explain the rule for lookup: scopes are searched for a name and if the global scope is reached, any namespaces specified with a using directive are "unlocked" and also searched. A more complicated example will see whether we understand that:

```
namespace A {
       int j;
}
int j;
void f()
{
       int j = 0;
       if (j)  // fine: local j
       {
              using namespace A;
              j = 0;  // finds local j
       }
}
```

The scopes are searched outwards and the local variable *j* is found. Since we didn't reach the global scope, no namespaces were unlocked.

What about when one namespace uses another?

```
namespace A {
       int j;
}
namespace B {
       using namespace A;
       int k;
}
void f()
{
       using namespace B;
       j = 0;  // fine: A::j
}
```

This is because the namespace lookup is transitive – once one namespace is unlocked for lookup, any other namespaces mentioned are also unlocked. In particular if *B* had defined another variable *j* in the above example, the use of *j* in *f* would have been ambiguous: *B*::*j* or *A*::*j*. Note, however, that in the example as written, *j* is not a member of *B* so the following will not work:

```
namespace A {
       int j;
}
namespace B {
       using namespace A;
       int k;
}
void f()
{
       B::k = 0;  // fine: k is member
of B
       B::j = 0;  // error: no j in B
       using namespace B;
       j = 0;     // fine: B is unlocked
                  // and so is A
}
```

This can get particularly confusing, in my opinion, when functions are involved instead of variables, because they *overload across namespaces* once any namespace in the chain is unlocked!

```
namespace A {
       void h(int);
}
namespace B {
       using namespace A;
       void h(char);
}
void f()
{
       A::h('a');  // calls A::h(int)
       B::h(123);  // calls B::h(char)
       using namespace B;
       h('a');     // chooses B::h(char)
       h(123);     // chooses A::h(int)
}
```

I'll leave it as an exercise to construct more complicated examples but I'd like to hear your comments on this – send me email about it and I'll summarise in *Overload 9*.

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 "Circle & Ellipse" problem posed by Francis Glassborow in *Overload 7* generated quite a lot of responses – Kevlin Henney and Francis both give their sides to the solution. Multiple inheritance also features heavily as The Harpist continues his discussion of object relationships and Ian Horwill gives a beginner's-eye view of **virtual**. Kenneth Jackson provides useful data validation techniques for MFC controls, Bryan Scattergood shows how you might start to write "smart pointers" and Peter Wippell revisits the "Record I/O" theme from the last issue. Part II of Ulrich Eisenecker's series on multiple inheritance has been held over to *Overload 9* for reasons of space.

Wait for me! – virtual

by Ian Horwill

This is the second article aimed at people who, like me, are still valiantly trying to develop basic C++ programming skills while the rest of the world gallops on with the latest language developments.

In this article, I'd like to cover the two uses of the keyword **virtual**. This was an area of some unnecessary difficulty for me when learning C++, not because of any particular complexity of the ideas involved, but because of the word itself!

The problem started in 'O' level physics. For whatever reason, I couldn't really understand what was meant by a "virtual image" when studying lenses etc. Then, when I saw this word in C++, I thought – "Uh oh, this must be difficult!"

However, I finally sorted it all out. "Virtual" means "something that isn't really, but for our purposes we can pretend it is", e.g., virtual memory.

We'll see how I think that relates to virtual functions next. As for virtual base classes, there are probably sufficient similarities to virtual functions to justify re-using the keyword.

Virtual functions

Consider the following:

```
class NetwareServer
{
public:
    void connect();
    void disconnect();
    ...
};

void DoSomething(NetwareServer& host)
{
    host.connect();
    ... // do something!
    host.disconnect();
}
```

It's pretty obvious that *connect***()** and *disconnect***()** are member functions of the class *NetwareServer* and those are the functions that are being called.

Now, to extend the program you might decide to add different types of server, e.g., a Unix host on a dial-up link. As you can imagine, the equivalent *connect***()** function will be somewhat different from the Netware one.

Having read our book (or chapter – we're in a hurry) on object-oriented programming, we decide we need an abstract type that will represent the common features of all our servers without bothering about the details of any of them.

(Note – an abstract class is one that is not intended to represent anything directly, but which specifies an interface – set of functions etc. – from which we derive classes that do represent something, i.e., an abstract class represents a kind of "virtual" object. Hmm!)

We then want to be able to do this:

```
void DoSomething(Server& host)
{
    host.connect();
    ... // do something!
    host.disconnect();
}
```

without worrying about what type of server (Netware or Unix) we are dealing with. Obviously we need to write specific versions of the *connect***()** and *disconnect***()** functions somewhere, and we do that by deriving classes from our new abstract class:

```
class Server
{
public:
    void connect();
    void disconnect();
    ...
};
class NetwareServer : public Server
{
public:
    void connect();
    void disconnect();
    ... // Netware-specific stuff
};
class DialUpServer : public Server
{
public:
    void connect();
    void disconnect();
    ... // dial-up specific stuff
};
void NetwareServer::connect()
{
    // Attach/login to Netware server
}

void NetwareServer::disconnect()
{
    // Logout from Netware server
}
void DialUpServer::connect()
{
    // Dial up/login to remote server
}

void DialUpServer::disconnect()
{
     // Logout/disconnect from the
    // dial-up server
}
```

Because *NetwareServer* and *DialUpServer* are derived from *Server*, we can call *DoSomething* with either of them because it is declared with a *Server***&** parameter:

```
NetwareServer file_server;
DialUpServer  internet_host;

DoSomething(file_server);
DoSomething(internet_host);
```

So, we now have an abstract base class which specifies that objects of this type have *connect***()** and *disconnect***()** functions, and two derived classes that provide specific implementations of those functions.

However, as written, the last *DoSomething***()** function above calls the ***connect*()** and *disconnect***()** functions of the base class (*Server*). These functions may not even exist, and we don't want them to be used anyway!

What we want to say in *DoSomething***()** is "call whichever versions of these functions apply to the actual object we are dealing with". And that is exactly what virtual functions do. If we change the definition of **Server** to:

```
class Server
{
public:
    virtual void connect() = 0;
    virtual void disconnect() = 0;
};
```

then *DoSomething***()** works just as we want it to (we could also add "virtual" to the function declarations in the derived classes, but it is redundant).

The "= 0" makes our virtual functions "pure" virtual functions, which means we aren't going to define versions of these functions for the class *Server* itself. This also means we can't create objects of type *Server* directly, which is what we want and is what makes it an abstract class.

So this is why they are called "virtual" functions; they look as though they exist for an abstract base class, but they don't really.

If we left off the "= 0" and defined these functions for *Server*, they would act as defaults for any derived classes that didn't provide their own versions. We would also be able to declare objects of type *Server* (assuming it didn't have any other pure virtual functions).

How do they do that?

To accomplish this piece of virtual magic, each object of a class with virtual functions has a hidden data member, which is a pointer to a table containing the addresses of its versions of the virtual functions. Let's look at some fake compiler-generated code to see what happens:

(Please note – this is for illustration only!)

```
class Server
{
public:
    virtual void connect() = 0;
    virtual void disconnect() = 0;
    ...
    // The table must be accessible
    // through the base class:
    function_address* vtbl;
};
class NetwareServer : public Server
{
public:
    virtual void connect();
    virtual void disconnect();
    ...
    // Each class has its own set of
    // function pointers:
    static function_address
                     netware_vtbl[2];
    // And we initialise the vtbl
    // pointer in the base class to
    // point to our function table
    NetwareServer()
    { vtbl = netware_vtbl; }
};
function_address
NetwareServer::netware_vtbl[2] =
{
    connect, // address of
        // NetwareServer::connect()
    disconnect // address of
        // NetwareServer::disconnect()
};
void f()
{
    NetwareServer file_server;
    DoSomething(file_server);
}
void DoSomething(Server& host)
{
// call NetwareServer::connect()
    (host.vtbl[0])();
...
// call NetwareServer::disconnect()
    (host.vtbl[1])();
}
```

So when the compiler doesn't know which derived class it's working with (such as in *DoSomething***()**), and therefore which version of a **virtual** function to call, it looks it up in the objects vtbl.

When the object's type is known, the compiler can call the correct virtual function directly, avoiding the overhead of the indirect virtual function call:

```
void f()
```

```
{   NetwareServer server;
    server.connect();
    // NetwareServer::connect() can be
    // called directly here
}
```

Virtual base classes

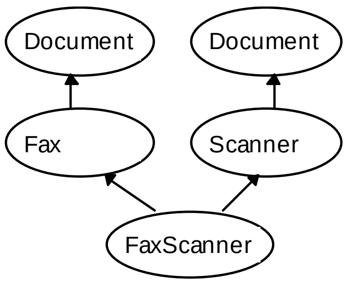
C++ allows multiple inheritance – i.e., a class can be derived from more than one base class:

```
class Fax
{
public:
    void receive();
    void send();
};
class Scanner
{
public:
    void scan();
};
class FaxScanner
: public Fax, public Scanner
{
// Other stuff in addition to
// receive,send and scan
};
```

So the class *FaxScanner* has all the capabilities of a *Fax* and those of a *Scanner*. Now presumably, the *Fax* class receives and sends a document, and the *Scanner* class scans in a document. So let's assume we have a *Document* class to represent this, which we will add as a base class to both *Fax* and *Scanner*:

```
class Fax : public Document
{
public:
    void receive();
    void send();
};
class Scanner : public Document
{
public:
    void scan();
};
```

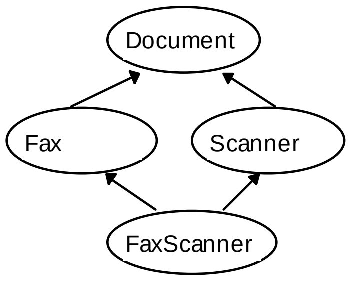
The problem with this is that our *FaxScanner* has two documents, one with its *Fax* part and one with its *Scanner* part:



To get round this, *Fax* and *Scanner* both have to declare *Document* as a virtual base class:

```
class Fax
: public virtual Document {
// as before
};
class Scanner
: public virtual Document {
// as before
};
```

Now **virtual** here doesn't mean that the *Document* doesn't really exist; rather, it means that a separate copy will probably not exist because it will be shared with all other classes in the same tree (i.e., that are bases – direct or indirect – of some derived class):



Note that any class in the tree that doesn't declare *Document* as virtual still gets its own copy.

Now that all seems simple enough, doesn't it? However, from this useful feature an number of complications arise. Here are two of them:

Initialisation

It is quite likely that *Document* will have one or more constructors, and a class derived from *Document* will supply arguments to one of these constructors to initialise its *Document* part.

You've beaten me to it – what if *Fax* and *Scanner* both supply different arguments to *Document*'s constructor? Or call different constructors?

The answer is simple and specific. A virtual base class can only be intialised by the "most derived" class, which in our case is *FaxScanner*. Intermediate classes (*Fax* and *Scanner*) can provide intial values for *Document* (and should, for when they are used on their own) but they will be ignored.

```
class FaxScanner
: public Fax, public Scanner
{
public:
    FaxScanner()
    : // probably initialise Fax and
      // Scanner here too
      Document(x) // x is some initial
                  // Document
};
```

Note that *FaxScanner* can initialise *Document* even though it is not a direct base class.

If the most derived class does not specify initial values for the virtual base class, then its default constructor will be used. If it doesn't have one, and the compiler can't generate one (because other constructors exist), the program is in error.

Casting

C++ allows a pointer or reference to a derived class to be implicitly converted (i.e., without you having to say so) into a pointer/reference to any of its base classes, as long as the base class is accessible at the point of the conversion (public base classes are accessible everywhere).

E.g., if the memory layout of a *FaxScanner* is:

Fax part
Scanner part
FaxScanner part

then a pointer to a *FaxScanner* can be inplicitly converted to a pointer to a *Scanner*:

```
void f()
{
    FaxScanner fs;
    Scanner* p = &fs;
}
```

and that the compiler will adjust the value of the pointer to do the conversion, by adding the size of the *Fax* part (because the pointer to *FaxScanner* points to the start of the whole object). This is a constant value known at compile time.

An explicit conversion is also allowed from the base class pointer back to the derived class pointer. The same constant value is subtracted from the base class pointer. The conversion has to be explicit because you are telling the compiler "this instance of the class X is part of a Y, honest". If it weren't, the resulting adjusted pointer could be pointing to anything (or nothing).

Now let's look at the situation with virtual bases. The layout of a stand-alone *Scanner* object might look like this:

Document part
Scanner part

giving an offset of 0 to convert a *Scanner** to a *Document**, whereas if the *Scanner* object is part of a *FaxScanner* the fact that *Document* is a virtual base means its position relative to *Scanner* or *Fax* (or both) must be different from its stand-alone position, e.g.,

Document part
Fax part
Scanner part
FaxScanner part

Therefore the offset of a virtual base relative to a derived class is not constant and so must be stored as an extra (hidden) field in the derived class. This mechanism is invisible to the programmer, and the derived-to-base conversion can be performed as normal.

(Note: I've used the pointer conversion example, but the offset is also needed when accessing, for example, *Document* fields from within *Scanner* member functions.)

The opposite conversion, from a virtual base pointer to a derived class pointer, is not allowed (remember it *is* allowed when the base is not virtual). My copy of the C++ Annotated Reference Manual says that this is "to avoid requiring an implementation to maintain pointers to enclosing objects".

And finally...

Needless to say, virtual base classes can contain virtual functions and they work pretty much as you might expect, although they do add some complications for compiler writers.

Well, that's it from me. May all your programs be virtually error-free.

Ian Horwill

ian@horwill.demon.co.uk

Circle & Ellipse – Vicious Circles

by Kevlin Henney

Modelling the relationship and similarity between circles and ellipses with inheritance is a recurring question in articles and newsgroups. It is often seen as a paradoxical problem beyond the reach of OO. The Harpist went over the problem in *Overload* 7 [1], and Francis followed this with a request for solutions [2].

The problem

The reason a solution is so hard to come by is because the problem is poorly stated: mathematics tells us that a circle *is* an ellipse, so I can substitute a circle wherever an ellipse is required, suggesting that a circle is a subtype of an ellipse:

```
class ellipse
{
      ...
};
class circle : public ellipse
{
      ...
};
```

So far so good, but the troubles start when we introduce any state modifying functions, such as assignment or the ability to change the major and minor axes independently. The invariant for a circle states that its axes are the same, and can be known alternatively as the radius. The following code illustrates that we can easily break the invariant:

```
circle c(1);
ellipse& e = c;
e = ellipse(2,4);   // oops rather
                    // than oop!
```

This seems less pathological when you consider that the reference binding might be of an actual to a formal function parameter. There seems to be a problem, and one that the Harpist and many others believe cannot be expressed using OO or within the C++ type system. I believe this to be a non-problem, albeit a subtle one. Not only do I think that OO and C++ are up to the job, but you will find that there is more than one solution.

The real problem

We've looked at what the problem appears to be, but did anyone spot what I did wrong? As I said, this is subtle: *what are the requirements and where is the analysis*?

I am in the middle of a design that doesn't work, so is it the paradigm or the design that is at fault? We are so confident that we understand the mathematical concepts behind circles and ellipses that we have not bothered to ask any more questions of that domain. The tip-off is in the phrase "the troubles start when we introduce any state modifying functions".

We have also not said what we wish to use our circles and ellipses for. Not only is the previous design flawed by internal contradiction, it is not fit for a purpose for the simple reason that we have failed to identify one.

Principia Mathematica

Let us restrict ourselves to just modelling circles and ellipses as we see them in maths, rather than for any specific application such as graphics:

```
class ellipse
{
public:
    ellipse(double a, double b);
    ellipse(const ellipse &);
    ~ellipse();
    // queries:
    double semi_major_axis() const;
    double semi_minor_axis() const;
    double eccentricity() const;
    double area() const;
    ...
    bool operator==(const ellipse&)
                              const;
    bool operator!=(const ellipse&)
                              const;
private:
    double semi_major, semi_minor;
    // no implementation
    ellipse& operator=(const
                          ellipse&);
};
class circle : public ellipse
{
public:
    circle(double r);
    circle(const circle &);
    ~circle();
    // queries
    double radius() const;
private:
    // no implementation
    circle& operator=(const circle&);
};
```

The first observation is that there is no way to change circles and ellipses once you have created them. Even if you do not declare them **const** they are effectively **const** objects. This is the correct mathematical model: there are no side effects in maths, conic sections do not undergo state changes, and there are no variables in the programming sense of the word. Two conic sections with the same parameters are indistinguishable, and so their internal state is effectively their identity: change the state and you change the identity. In some senses a copy constructor can be considered superfluous.

We are dealing with value based rather than reference based objects. Readers who are comfortable and familiar with functional programming and data flow models will recognise the approach. In the case of circles and ellipses, the circle is simply an ellipse with specialised invariants. There is no additional state and none of the members of an ellipse need overriding as they apply equally well to a circle. For this reason I have kept the classes completely nonpolymorphic.

The case for *const* inheritance

The solution above is a disarmingly simple solution, but from the mathematical perspective it is the correct one. In a procedural language, however, you will doubtless find the inability to assign these objects quite constraining: state is bound to objects once at the point of creation. Once you have assignment and a few transformation functions, such as a function that returns the ellipse after stretching along a particular axis, you need no other modifiers. But even the introduction of this one mutator function will break the subtyping relationship that circles may be used wherever ellipses are expected.

Another way to look at this is that we want to change ellipses in a way that is incompatible with circles, and yet preserve the value subtyping between them. The simplest solution to this is unfortunately not currently possible in C++:

```
class ellipse
{
    ...
public: // ch-ch-changes
    ellipse& operator=(const
                          ellipse&);
};
class circle
: public const ellipse // not real code
{
    ...
public: // as good as a rest
    circle& operator=(const circle&);
};
```

The circle class now partially inherits from an ellipse, but in a way that preserves the subtype relationship: a circle is substitutable for a **const** ellipse. In other words, where you expect an ellipse that remains unmodified, a circle is substitutable:

```
circle c;
ellipse& e = c;       // illegal
const ellipse& e = c; // legal
```

Because there is no polymorphism, no state added, and a closed inheritance hierarchy, the DESTROY-CREATE pattern for assignment can be used in the circle without any problems:

```
circle& operator=(const circle& rhs)
{
    if(this != &rhs)
    {
        this->~circle(); // destroy
        // and recreate in same place
        new(this) circle(rhs);
    }
    return *this;
}
```

This obviates the need for protected access to the parent class. However, this is one of the very few places that this particular pattern can be used safely, otherwise you would be wise to keep it out of your code [3].

In principle once you have assignment, all legal state changes to ellipses and circles are possible simply by assigning from a newly constructed temporary. However, this could become tedious for applications where certain operations are common, such as changing one axis independently of the other. In this case a convenience member might be appropriate.

More classes

All well and good, but **const** inheritance is not currently legal. The alternative is to model this using separate classes: the **const** versions form the trunk of the inheritance tree, with the mutable versions as extensions off them:

```
class ellipse // const class
{
... // as side effect free version
};
class mutable_ellipse
: public ellipse
{
... // as const inheritable version
};
class circle
: public ellipse // const class
{
... // as side effect free version
};
class mutable_circle : public circle
{
... // as const inheriting version
};
```

Although this approach uses more classes to express the same ideas, it is open to some extension that the briefer **const** inheritance version could not manage. For instance, both circles and ellipses may be symmetrically rescaled. Such a feature can be incorporated into the ellipse class without violating the invariant of the circle:

```
class ellipse
{
    ...
    ellipse& operator*=(double);
    ...
};
```

You may notice stronger coupling between the two classes now. This closed anticipated relationship may not be appropriate for many designs, so the purely **const** version may be preferred. As ever, such a design decision should be taken in context.

The property market

In the limit, a circle is simply an uneccentric ellipse, and its valid set of states is a strict subset of the ellipse's. It has to be said that other than its complete symmetry, there is nothing of particular interest in a circle that is not already modelled by an ellipse. For some applications this is enough reason to model circularity as a predicate property of an ellipse and completely abandon the modelling of this special case as a separate class:

```
class ellipse
{
    ...
    bool circular() const;
    ...
};
```

This is analogous to the way that empty strings do not need a separate class to model them: an ordinary string class will suffice. Again, whether or not this is an appropriate approach depends completely on your expected use.

All sorts to make a world

For graphics it is likely that we need a more complex model of ellipses and circles than the one presented above. For instance, we might wish to position our shapes on a set of world coordinates. Another design decision is whether to model location and orientation within the object or not. If we model a graphical shape as a primitive shape plus translational, rotational and scaling transforms, there is no need to change the mathematical model presented above. However, should we decide to incorporate positional features in the shapes and preserve the integrity of the design at runtime, we can:

- abandon the hierarchy, as Coplien [4] suggests, treating circles and ellipses as quite separate shapes — in addition we can include methods to create one from the other, or a null shape on failing;
- abandon circles, as suggested in the previous section, regarding the equality of major and minor axes as a usage convention;
- treat shapes as pure value objects, so that all objects are transformed by returning a replacement shape — this is an extension of the side effect free approach and is similar to the idea of replacement behaviour in *actor theory* [5].

Other approaches

There are often attempts to model the relationship the wrong way up, i.e., have a circle as the base class for an ellipse. In terms of subtyping, this clearly makes no sense and is often the result of a failed and misguided attempt to reuse implementation. It has been suggested that a virtual assignment operator for the ellipse, overridden in the circle class, might be a solution if it did 'clever' things like:

- ignore a non-circular operand, leaving the state unchanged, or
- take the average of the axes to assign the new diameter from, or
- use the larger or smaller axis as the diameter, or
- use assertions or exceptions to introduce stronger behavioural preconditions.

A similar philosophy has also been applied to stretching functions and other transforms. However, these all suffer from the problem that they are quite obviously kludges rather than correct implementations of the specified operation: stretching a circle in one direction should give an ellipse and not a larger circle.

I first encountered a practical example of the circle-ellipse problem in a graphical system to which group editing was added. Prior to this all editing was on primitive or separately definable and nameable composite symbols. Group editing permitted translation and rescaling, but not rotation. None of the above workarounds for rescaling circles work because a group requires its members to maintain their relative positioning — very strange things happened when they do not! Symbols, composed of primitive shapes and other symbols, never ran into this problem because they were defined as draw — rather than actual — transforms on a prototypical symbol notionally defined at the origin. Had the bounding boxes of shapes been independent of the shapes there might have been less of a problem, but one of the great advantages of primitive shapes is that the bounding box need not be explicit as it can be deduced!

Until I added group editing with stretching, the design problem either not had existed or had lain dormant, depending on your point of view.

Conclusion

Although I have done it above, in many cases it is not especially good practice to derive one concrete class from another: the other issue lurking in this discussion is the difference between type and class, and its effect on system design, integrity and reuse. However, it is possible to discuss the circle-ellipse problem without taking such a detour – for brevity I did.

The solution to the circle-ellipse problem is one of *method* rather than of *the* method. Put like that, it is unsurprising. What is surprising is that because of our expectations and prior knowledge such an apparently simple modelling task can have such sensitivity to initial (and final) conditions. Then again, I guess circles and ellipses are non-linear...

References

[1] The Harpist, "Related objects", *Overload* 7, April 1995

[2] Francis Glassborow, "Related addendum", *Overload* 7, April 1995

[3] Andrew Koenig, "Using constructors for assignment", *C++ Report* 7(2), February 1995

[4] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992

[5] Gul Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986

Kevlin Henney

kevlin@wslint.demon.co.uk

Circle & Ellipse – Creating Polymorphic Objects

by Francis Glassborow

A recap

You remember that in the last issue of *Overload* I threw down the gauntlet with a challenge to all and sundry to implement a polymorphic object type as opposed to a polymorphic type. Well no-one has sent me a solution (perhaps you sent them to Sean instead, or asked him to return all my contributions unopened).

As if! :-) – Ed.

In order to implement such mathematical relationships as those that hold between a circle and an ellipse we need objects that can change their types at run time. We can change behaviour by replacing member functions with addresses that can be reset at runtime. Unfortunately this is not enough because we need to make sure that the dynamic type is changed so that the new **typeid** and **dynamic_cast** facilities will work correctly as well. These facilities depend on the virtual function table pointer (or so I believe) for their success. This means we need to change this hidden variable if our objects are to behave polymorphically. We also have to tackle the problem of varying amounts of data – ellipses require more than circles, and make sure that we cope with any other hidden information that implementors have elected to use.

I am not going to give you a fully worked solution but I hope that the following will be enough for you to flesh it out for yourselves. I would also be delighted if some of you come up with other (probably better) solutions.

Variable volume of data

The first thought that crossed my mind was that I might try some form of inverted inheritance tree for the data with ellipse data derived from circle data. The problem with this approach is that, as a general solution, it inhibits further development of the principal hierarchy (*Shape* in this case). Careful thought led to the following:

```
class Shapedata
{
protected:
       // make Shapedata an abstract
       // base class
       virtual void setdata(const
                          Shapedata&)=0;
       virtual ~Shapedata() {};
public:
       // default constructors are OK
};

class Shape
{
       Shapedata* data;
public:
       Shape() : data(0) {}
       Shape(const Shape&);
       // shape interface of pure
       // virtuals
       virtual ~Shape()
       { delete data; }
};
```

Notes:

The *Shapedata* destructor is protected which means that only derived classes will be able to use it. The same is true of *setdata***()**. *Shapedata* is only intended for use by *Shape* derivatives. I have tried to ensure that derived classes from *Shapedata* are only available to the *Shape* hierarchy.

As all our concrete *Shape*s will have dynamic instances of *Shapedata* handled by the *Shapedata* pointer, I have set that pointer to zero in the constructor for *Shape* so that I can safely delete it in the destructor for *Shape*.

A point I noticed while testing the ideas for this solution was the way that different compilers deal with the definition of **virtual** functions in the interface. Some compilers simply suppress the implicit **inline** attribute, others issued a diagnostic and some messed up. I had not consciously recognised this as a problem before. Reflection shows that there is a problem here because **inline** requires code to be available so that the compiler can use it (it is too late by the time the linker is involved). By contrast **virtual** means delay selection of code until runtime (link time is too early).

Another problem that arose was that I wanted to declare the *Shape* hierarchy as a **friend** of the *Shapedata* hierarchy. The language does not seem to allow this. Perhaps someone else has a better idea.

I considered making *Shapedata* a **protected** local class nested in *Shape*. I decided to skip this idea for the time being because it would have led to deriving a nested class from a base class that was itself nested in a base class. I suspect that this approach has potential for hiding the implementation of the *Shapedata* hierarchy.

Now using these `ABCs` we can define two pairs of concrete classes:

```
class Ellipse; // predeclare
class Ellipsedata : public Shapedata
{
friend class Ellipse;
// copy constructor OK
// copy assignment OK
       float diameter; // major axis
       float xfocus1, yfocus1,
             xfocus2, yfocus2;
       Ellipsedata(float xf1,
                    float yf1,
                    float xf2,
                    float yf2,
                    float d);
       void setdata(const
                    Shapedata&);
};

class Ellipse : public Shape
{
       void change();
protected:
       Ellipse(){}
public:
       Ellipse(float xf1,
                    float yf1,
                    float xf2,
                    float yf2,
                    float d)
       : data(new Ellipsedata(xf1,
                           yf1, xf2,
                           yf2, d))
       {};
       ~Ellipse() {};
};

class Circle; // predeclare
class Circledata : public Shapedata
{
friend class Circle;
// copy constructor OK
// copy assignment OK
       float diameter;
       float xcentre, ycentre;
       Circledata(float x = 0,
                    float y = 0,
                    float d = 2);
       void setdata(const
                    Shapedata&);
};

class Circle : public Ellipse
{
       void change();
protected:
       Circle();
public:
       Circle(float x = 0,
              float y = 0,
              float d = 1)
       : data(new Circledata(x,y,d))
       {};
       ~Circle() {};
};
```

Notes:

Usually the derived versions of *Shapedata* will have *Shapedata* as a direct base class. This is because the data required for each type of shape will depend on the shape rather than on the shape from which it is derived.

The *change***()** function is special as it must only be called from other polymorphic functions at the level of the current dynamic type of the object because this is the key to polymorphism at object rather than class level. If you called it from a non-polymorphic function you might call a static version that was not correct for the dynamic type of the object. Remember that objects of any polymorphic kind are often handled via pointers or references whose static type is that of a base class.

The default constructors for the concrete *Shape* classes are **protected** because the true dynamic type will need to initialise the *Shapedata* pointer to its own type of data, but the rules will require the base class to be constructed first. So an instance of a circle must first create a dataless ellipse. On the other hand it must not be possible for a naive client to create a dataless ellipse. Dataless shapes can only be bases.

Now let me move on to the implementation of the *change***()** function. The first thing it must do is to grab hold of responsibility for the current *Shapedata* so that it can use it to initialise the *Shapedata* for the new type.

Then it must explicitly call the destructor for the present *Shape*. We do not want to release the memory in use; just to remove the object. When we have done that we use a placement **new** to construct the new object in the same memory. Lastly we delete the old *Shapedata* instance.

If you are going to use a placement **new** you must remember to include **new.h**. I forgot and found myself wasting rather more time than I should have done. The fact that ordinary versions of **new** work without **new.h** while placement **new**'s require it is a pain.

For the less experienced C++ programmer who is reading this I should explain the concept of a placement **new**. The **new operator** in C++ is implemented in two parts, both of which may be user supplied. The first is a function that supplies memory. The second is a constructor for the relevant object. The first of these functions is, confusingly, called **operator new**. The user can provide both global and in class overrides of the default version. You can also provide overloaded versions in both global and class scopes. Any overload version of **operator new** is said to provide a placement **new**. The most common of these is one where the user provides a pointer to the required memory.

The prototype for a basic **operator new()** is:

```
void* operator new(size_t);
```

All user written versions must return a **void*** and take a *size_t* as the first parameter (this will be the value of the **sizeof** the object being created). We do not need to mess with the normal **operator new()** for the current problem. What we need is a simple placement **new** where the user provides a pointer to the memory to be used. The following will do the job nicely:

```
void* operator new(size_t, void* vp)
{ return vp; }
```

Note that I have made the first parameter anonymous because I do not need it though the rules of the language require it to be present. I could have provided an in class version of the operator as a **static** member function, but the standard one (actually provided by most compilers) is all I need. Also note that an eccentricity of all placement **new**'s is that the value for the first parameter (the amount of memory required) is supplied by the last argument (the object being created) in the call.

So let us look at the code for a change function:

```
void Circle::change()
{
      Shapedata* temp = data;
      data = 0; // disconnect
                // original
      this->~Circle();
                // destroy the
                // circle object,
                // keep the memory
      new (this)
            Ellipse(temp->xcentre,
                  temp->ycentre,
                  temp->xcentre,
                  temp->ycentre,
                  temp->diameter);
      delete temp;
            // now discard the
            // original data
}
```

Of course this is just one of a number of ways that you can achieve the same end.

Conclusion

As far as I know this method must work. I am not conscious of using any undefined behaviour. However you will need to be particularly careful about where you call *change***()**. It could be very dangerous to make it a polymorphic function because by the time you return the object has changed to a different type. I think this means that it must always be the last function called from a member function that has done something that requires the object to morph into something else. I might be being over cautious about this and would welcome your opinions.

I suspect that our esteemed editor will describe the whole mechanism as a 'horrible hack'. It certainly is not something to rush at if you are programming with serious intent. Note that the mechanism for handling data is vital because it guarantees that all *Shape*s have the same basic size, with the variable amount of data hidden away out of harms way.

Francis Glassborow

francis@robinton.demon.co.uk

Francis is right – I think the DESTROY-CREATE *pattern is horrible and with the recent committee decisions on object lifetimes, I have no idea whether it's safe or not – Ed.*

Having Multiple Personalities
by The Harpist

Before I tackle the main topic of this article I would like to present a somewhat different view of inheritance criteria from the one that is glibly thrown around by most authors.

Writers of books on object-orientation (should those words be entitled to capitals?) give a simple rule of thumb for determining when to inherit (the *is-a* relationship) and when to use aggregation or layering (the *has-a* relationship). Scott Meyers extends this a little by including another option, that of 'being implemented as a' to which he assigns private inheritance, or protected inheritance if this detail belongs to the hierarchy rather than to the base-class designer. The problem with such simplistic rules of thumb is that they are expected to be applied with understanding. Those inventing the rules have the understanding to use them. They include this essential ingredient when they pass it on. It is those who hear the rule without listening to the intent that then teach it to others without the all important explanation.

It is my contention that the right decision are more along the lines:

- X can be used as Y – all X behaviour is directly Y behaviour: derive X from Y (or provide a conversion from X to Y)
- Y is a part of X but Y's behaviour is not directly part of X's: layer Y inside X (do not provide any conversion)

The traditional vehicle-car-wheel trio will serve to exemplify this. The behaviour of a car subsumes the behaviour of a vehicle and so we would expect to be able to use a car as an instance of a vehicle. On the other hand, all cars have wheels (let us not get pedantic about this and start talking about hover-cars) and the behaviour of a wheel is essential to the working of a car but it is not part of the behaviour of a car.

Other important criteria when building inheritance trees are those of data and behaviours. A more derived object should have at least as much data and at least as much behaviour as one of its bases. This is the reason why most mathematical objects do not fit in traditional OO inheritance hierarchies. Mathematical objects are related by increasing constraints. A square is specified by less data than a rectangle because part of the information is encapsulated directly in its nature. A rectangle has a greater range of behaviour than a square because squares are constrained rectangles. If you do not understand this – at least at the implicit level – you have not yet made the paradigm shift to an OO world view.

This does not mean that C++ cannot handle such relationships; it can but not through conventional OO methods. Many programs and class hierarchies are broken at the design stage because the designer/programmer does not understand this subtle aspect of OO. C++ used for mathematics, science and engineering demands these insights from its users. Those doing much business and commercial programming may get away without this understanding.

Real-World versus Object-Oriented

Another problem arises from those who relate the objects of object-orientation to real world objects. Just because something is a real world object does not automatically qualify it to be an object-oriented one. On the other hand there are many things that are not objects in the real world sense but which would be object-oriented ones. Abstract behaviours and algorithms are two examples of such.

If we are to obtain the versatility that we need for writing powerful, reusable code we need to grasp such things and understand the implications.

Several years ago Francis came to me with a problem that an ACCU member had raised. Take a relatively simple single inheritance hierarchy and try to change the behaviour of a base class. For example change a sort from a heap sort to a quick sort. You cannot do it unless you have the source code for the relevant base class. It is easy to tune behaviour of the most derived class, you just derive again and override the behaviour in question but you cannot splice in a change anywhere else.

This is not an essential problem with C++, it is one created by class designers who over-specify the behaviour of base classes. It is not the job of a class designer to determine which sorting algorithm should be used unless it is for purely internal data where the designer knows the nature of the data set.

Such behaviour should be provided via some form of polymorphic behaviour. To do this we need carefully designed algorithmic classes. It is because full code reuse requires such developments that writing reusable code takes so long. Many software groups are used to the concept of analysis and design applied to applications but code reuse requires that similar care is taken with the analysis and design of tools (reusable code).

I am not going to tackle this area here but I think that we need a series of articles addressing this vital area (like an article about baggage classes). The majority of commercial libraries for C++ are not fully reusable because their design too often over-specifies base classes. In the rest of this article I would like you to keep the needs of design for reuse in the back of your mind because I believe that many apparently complex techniques become attractive in such circumstances.

In simple terms I think that such things as multiple inheritance are largely the domain of the library designer. Further, I think library designers that do not understand multiple inheritance are like architects who do not understand load bearing.

Being two-faced

Consider the problem of developing a TextWindow class. It is clearly two things at once, an instance of text and an instance of a window. Because it is both these things we could reasonably expect it to substitute for either as the context required. The question is how we should provide this mechanism. First let me clear out the obviously silly.

```
class Window
{ ... };
class TextWindow : public Window
{ ... };
```

It's true that a *TextWindow* object can substitute for a *Window* but how is it going to be pure *Text*? We can do a little better with:

```
class Window { ... };
class Text { ... };
class TextWindow : public Window
{
public:
    operator Text () { return data; }
    // rest of public interface
private:
    Text data;
    // other private items
};
```

This means that by inheritance we provide an automatic conversion from *TextWindow* to *Window* and the user provided conversion operator will deal with conversion from *TextWindow* to *Text*. If the lack of symmetry worries you then you could write:

```
class altTextWindow
{
public:
    operator Text () { return data; }
    operator Window () { return info }
    // rest of public interface
private:
    Text data;
    Window info;
    // other private items
};
```

I have made no attempt to smooth off the rough corners in the preceding code because I think that the approach is flawed. I do not want to be able to convert a *TextWindow* into something else, I want to be able to use a *TextWindow* as either a *Window* or as *Text*. To spend time tinkering with the above code so that I can do this is plain stupid. A *TextWindow* can be used as a *Window* so it must have *Window* as a base. Equally a *TextWindow* can be used as *Text* and must have *Text* as a base. Neither can a *Window* be used as a *Text*, nor can a *Text* be used as a *Window* so neither must have the other as a base. This logic dictates that:

```
class TextWindow
: public Text, public Window
{ ... };
```

is the way forward. Now we have the correct automatic conversions. A *TextWindow* object can be handled through a *Text******, a *Text***&**, a *Window****** and a *Window***&**. Of course, we must be careful about the potential for confusion. A *TextWindow*'s address obtained as a *Text****** will not be the same as its address as a *Window******. However, in a RTTI (runtime type information) environment, information will not have been lost – only hidden. I think that this preservation of information is extremely important but not everyone will agree with me.

Another point that should be remembered is that only one user provided conversion can be used in any implicit conversion sequence. It is my understanding that derived to base conversions do not count as user defined conversions. They better not, else the concept of a derived object being a base object is broken – a base object conversion requiring a user-defined conversion would not be implicitly available to a derived object.

Derived to base is a standard conversion, i.e., not user-defined – Ed.

This rule about user-defined conversions means that the choice between multiple inheritance and aggregation with operator conversions provides different behaviour. Such choices should be made by the class designer and not just decided by arbitrary coding rules that outlaw multiple inheritance.

It seems to me that the multiple inheritance route is much safer for the client programmer. If I have a *Text* class and a *Window* class provided by libraries and need a *TextWindow* then multiple inheritance is less likely to have hidden surprises for the inexperienced programmer. Getting user defined conversions to work is full of surprises to the extent that it is these that should be constrained by company coding standards. I do not mean that you should actively seek to use MI, just that I do not think you should be frightened to use it to combine disjoint behaviours. Using MI for overlapping behaviours is a very different story.

Building interfaces

Experienced class designers have a very different view of the world. They regularly separate interfaces from implementation. They have a whole toolkit for providing such things as ‘single interface – multiple implementation’ (pure ABC based polymorphism), ‘multiple interface – single implementation’ (Cheshire Cat based methods) and ‘type independent interfaces-implementations’ (template classes). These are not tools for children. Just as you wouldn’t give a chisel to a child, you shouldn’t be letting inexperienced C++ programmers near much of the technology available for stable reusable class hierarchy design. Sorry folks, but you must learn to crawl before you try to run; a good teacher will let you get from one stage to the next more quickly but you still need to develop skill. You also need to develop the wisdom to know which tool is appropriate in each circumstance.

When you come to design an interface you have three routes. First you can do everything from scratch and then implement it. This does not seem to be an appropriate route for those who want to increase reuse of code and design.

The second choice is to build your new interface from your collection of interface fragments. This has the advantage that you are reusing earlier interface designs but it is hard to build on existing implementations. For example:

```
class ABC1 { // specification of an
             // abstract base class
};
class ABC2 { // specification of an
             // abstract base class
};
class ABC3 { // specification of an
             // abstract base class
};
class Interface
: public ABC1,
  public ABC2,
  public ABC3 {
// added interface elements
};
```

Now you must provide one or more implementations via concrete classes derived from the *Interface*. Because of the multiple inheritance every one of the derived versions can be handled as any of the abstract base types. But you do not get reusable polymorphic behaviour of the *ABCn* classes when you use them for building different interface classes.

We could produce sets of implementations for each of the primitive abstract *ABCn* classes. These could be combined to produce our composite concrete class. The problem with this approach is that each concrete class will only be related to the others via a collection of unrelated abstract base classes. What I want is something like:

```
class Concrete1 : public Interface
{ ... };
class Concrete2 : public Interface
{ ... };
```

to derive a polymorphic hierarchy from *Interface*. The problem is that I do not get any code reuse (well I can always use cut and paste – but I do not consider that code reuse). Can I do any better?

On to the third approach which is similar to the second but you fragment the interface by virtual public inheritance so as to create your final concrete class by adding together concrete classes derived from the abstract derived fragments used as base classes.Well how about something like:

```
class ABC1x : virtual public Interface
{ // implement ABC1 };
class ABC1y : virtual public Interface
{ // implement ABC1 };
```

```
class ABC1z : virtual public Interface
{ // implement ABC1 };
class ABC2x : virtual public Interface
{ // implement ABC2 };
class ABC2y : virtual public Interface
{ // implement ABC2 };
class ABC2z : virtual public Interface
{ // implement ABC2 };
class ABC3x : virtual public Interface
{ // implement ABC3 };
class ABC3y : virtual public Interface
{ // implement ABC3 };
class ABC3z : virtual public Interface
{ // implement ABC3 };
```

Now you have a whole menu of concrete implementations that can be used to flesh out the *Interface* as appropriate. Typically you could have:

```
class Concrete1
: public ABC1x,
  public ABC2z,
  public ABC3y {
// minimum extras
};
```

Your variations are now all derived from *Interface* and so provide a polymorphic hierarchy based on *Interface*. The trouble is that the implementations of *ABCn* lexically reuse code in different interface classes but have to use a specific virtual base class so each interface hierarchy needs its own set of implementation fragments. What I want is:

```
template <class T> class ABC1y
: virtual public T {
// an implementation of ABC1
};
```

So I could write:

```
class Concrete1
: public ABC1x<Interface>,
  public ABC2z<Interface>,
  public ABC3y<Interface> {
// extras
};
```

Is this valid C++ code? I have just over-stepped my knowledge of C++. While the above use of a template seems perfectly reasonable, I do not know if templates work that way. I think this is probably the place to stop this excursion but I hope that this leaves you with some food for thought.

Oh, before I forget, my understanding is that composition of classes by multiple inheritance from several base classes is called using *addin* classes while the method based on fragmenting an abstract base class into several whose derived versions can be recombined is a variety of *mixin* technology. But maybe you know better. If so I am sure our esteemed editor would love to publish.

The Harpist

I'm sure The Harpist's article will generate a lot of responses – multiple inheritance and mixins certainly seem to fire some people up!

As for The Harpist's use of templates: yes, the code is valid but some compilers do not currently allow derivation from a template formal parameter (inheriting from T in the above example). Another, similarly powerful technique that some compilers don't currently support is this:

```
template<class T> class Base
{ ... };
class Derived : public Base<Derived>
{ ... };
```

I use this technique quite a lot to provide memory management optimisation for classes. I'll write it up one day! – Ed.

'Individual' Control Validation in MFC

by Kenneth Jackson

The motivation

Validation of controls within a dialog using Visual C++ and the MFC is relatively straightforward. This is achieved by use of the ClassWizard to add DDV_ (Dialog Data Validation) functions for the appropriate control to the *CDialog*::*DoDataExchange*() member function of the *CDialog* derived class. However, validation using this method is only performed when the user presses the OK button.

The question arises: how do you provide validation on a per control basis? That is, validation as the user moves away from a control. In many situations the user does not want to proceed on through a dialog if some initial data is invalid, for example, checking the form of an account number; serial number; reference number; or name. Taking this a step further it may be necessary to check an entry against a database field.

Stepping towards a Solution

I shall present successive refinements in order to highlight problems and show their solutions.

Validation

Providing validation within the *CDialog*::*DoDataExchange*() function is inadequate in cases where validation is required before the dialog is to be closed. A means of knowing that the user has moved from the control is required. This is satisfied by providing a handler for the *EN_KILLFOCUS* notification message within the dialog class. Appropriate Edit Notification handlers for controls can be added by use of the ClassWizard.

So far so good. Within the *CDialog*::*OnKillFocus***ControlName**() handler we can either provide the validation or call a function to provide the validation. In order to access the entry within the control we must now use a function such as *CWnd*::*GetDlgItemText*() giving the ID for the control being interrogated.

Cancelling out of the dialog

What happens if the user presses Cancel? Oh dear!! At present if the user presses Cancel after moving to a control for which validation is provided within the *OnKillFocus***ControlName**() handler the validation will still be performed. To avoid this it is necessary to know which control is to gain focus. However, notification handlers are passed no arguments.

The necessary information is passed to a handler for *WM_KILLFOCUS* (if there is one) within the control class. The one argument to *CWnd*::*OnKillFocus*(*Cwnd** *pNewWnd*), the *CWnd**, is the pointer to the control object which is to receive focus.

We only need to perform the validation if the *CWnd** passed to the *CWnd*::*OnKillFocus*() handler is not the address of the Cancel button. This can be done by calling *CWnd*::*GetDlgCtrlID*() to determine the control ID and compare it with *IDCANCEL*. In order to provide a *CWnd*::*OnKillFocus*() handler for each control we need to add control objects for each of these controls to the dialog. The ClassWizard can be used to add *CEdit* control objects to the dialog, for the edit controls. In the case of the edit controls it is necessary to derive a class from the standard *CEdit* control in order to add functionality to a message handler. The ClassWizard can be used to create a new control class, call it *CMyEdit*, by deriving a class from the generic *CWnd* and then changing the base class from *CWnd* to *CEdit*.

Having added the above mentioned control objects to the dialog class using the ClassWizard, the handler *CMyEdit*::*OnKillFocus*(*CWnd** *pNewWnd*) can be used to set a boolean value within the dialog (call it *m_bCancel*) to true if the *pNewWnd* is a pointer to the Cancel button.

```
if( pNewWnd->GetDlgCtrlID() == IDCANCEL)
{
      ((CMyDialog*)Parent)->
             m_bCancel = TRUE;
}
```

The *OnKillFocus***ControlName**() handler can now use the boolean *m_bCancel* to determine whether or not validation has to take place, i.e., not on cancel.

Chasing the focus

We are getting there slowly! Now we can provide the validation if the cancel button is not pressed. But what about using the system menu to close the dialog window? From what we have so far this would result in validation being performed, whereas from a user's point of view we would more typically expect this to have the same effect as pressing Cancel. Arguably a dialog of the type we are trying to develop should have limited exit points, namely only OK and Cancel. If we were to restrict the exit points then the problem does not arise, i.e., simply turn off the system menu in the AppStudio for the dialog resource. The alternative is to trap the *WM_SYSCOMMAND* in order to provide a handler to set the boolean *m_bCancel*. The former approach I feel is cleaner, but that is a personal opinion and there may be occasions requiring the system menu.

Having provided validation on an edit control what do we do if the validation fails? This is the point where the problems start! A message box could be used to convey to the user some suitable comment. However, where do we want focus to rest after the message box dialog has closed? Even worse is that we cannot initiate the message box from within the *EN_KILLFOCUS* handler where we currently have the validation. Shifting focus in the middle of shifting focus can have some interesting results – try it!!!

Have we put the validation in the correct place? Is there a better way of doing things? The answers to these questions are not easy but, put simply, we could attempt to place the validation somewhere else but the same sort of problem arises. It is possible to perform all sorts of cosmetic rearrangements but one will still be left with the same dilemma: how to inform the user of the validation failure and successfully transfer focus back to the offending control.

Register a new message

The solution is in providing one's own message, and thereby a message handler, such that the user initiated change of focus can be completed successfully before displaying error message.

```
const UINT NEAR WM_FAILEDVALIDATION =
      RegisterWindowMessage(
            "Failed Validation");
```

Within the *OnKillFocus**ControlName*()** handler the *WM_FAILEDVALIDATION* message can now be posted to the dialog itself. Simply:

```
PostMessage( WM_FAILEDVALIDATION);
```

An entry must be added manually to the message map, namely:

```
ON_REGISTERED_MESSAGE(
      WM_FAILEDVALIDATION,
            OnFailedValidation)
```

This can be added between the Wizard comment markers and the resulting message/handler association will be shown by the ClassWizard. For further details on the above see MFC Tech Note 6.

Handling the failure

The message handler *OnFailedValidation*(*WPARAM***,** *LPARAM***)** can now be provided. This message handler should first move the focus back to the offending control before a message box is displayed informing the user of the error. The *HWND* of the offending control can be passed within the *WPARAM* of the *WM_FAILEDVALIDATION* message. Thus we can now write something like:

```
LRESULT CPenDialog::OnFailedValidation(
WPARAM wp, LPARAM)
{
    if( wp)
    {
        ::SetFocus(HWND( wp));
        MessageBox("Number to big",
           "Failed Validation",
             MB_OK|MB_ICONINFORMATION);
    }
    return 1;
}
```

If it is necessary to distinguish between different validation failures the *LPARAM* can be used to pass some additional information to be selected upon.

Focus revisited

Unfortunately there are still two further problems regarding the moving of focus! One relates to the shifting of focus to another application and the second that displaying the message box causes a second call to the *OnKillFocus**ControlName*()** notification handler and a further attempt at validation!

The first can be resolved by testing to determine if focus is shifted to another application within the edit controls *OnKillFocus***()** handler, such as:

```
void CMyEdit::OnKillFocus(
      CWnd* pNewWnd)
{
    // TODO: Add your message handler
    // code here
    CWnd* pMainWnd = AfxGetMainWnd();
    CWnd* pWnd = pNewWnd;

    while(pWnd && (pWnd != pMainWnd))
        pWnd = pWnd->GetParent();
    if(pNewWnd->GetDlgCtrlID() ==
                     IDCANCEL || !pWnd)
    {
        ((CPenDialog*)Parent)->
                      m_bCancel = TRUE;
    }
    CEdit::OnKillFocus(pNewWnd);
}
```

The above code now tests that the window, to which the handler is passed a pointer, is a window within the current application. This is achieved by stepping back through its parents to see if it matches the current applications main window pointer. If not, then *pWnd* will be *NULL*.

The second problem can be resolved by adding a boolean flag to the dialog class, call it *m_bFailedValidation*, which is tested within the *OnKillFocus**ControlName*()** handler before doing any work. This flag is initially set to *FALSE* within the constructor. If the flag is set to *TRUE* then do nothing otherwise perform the validation. If the validation fails then set the flag to *TRUE*, then reset the flag to *FALSE* after displaying the *MessageBox* within the *OnFailedValidation***()** handler.

Conclusion

Whilst providing validation on a per control basis is possible it is certainly non-trivial. I have presented 'a' solution to the problem, however I am sure that it is capable of refinement.

The quest for a solution to this problem arose out of a client's question about this. At first I thought there must be a straight-forward way of achieving this, but was then told that they had spent a considerable amount of time trying to resolve it before giving up!

An interesting point to note is that OWL2.0 which comes with Borland C++ 4.x supports validation on an individual edit control basis. From the developers' point of view it is simply a matter of adding a validator object to the *TEdit* control. Validator objects being objects of a class derived from the *TValidator* class. There are various predefined classes of validator, but one can derive one's own. The advantage which OWL2.0 has is that the functionality for this form of validation is built into the class library, that is, there is functionality for validation built into the *TWindow* and *TEdit* classes. From the developers' point of view the whole mechanism is totally transparent.

Unfortunately with the MFC if you want individual control validation you will have to resort to adding the sort of code I have indicated.

Kenneth Jackson

kpjackson@cix.compulink.co.uk

From polymorphism to garbage collection

by Bryan Scattergood

The use of virtual methods in C++ allows us to exploit late binding, but only at a price. Dynamic binding is only possible through a pointer (or reference) to a base class. Arguments for which this polymorphism is to be exploited must be passed by reference or pointer to prevent slicing. However, while pass by reference works well for arguments, it does not work well for return values. The only way in which a function can return such a value by reference is to construct the value on the heap, and in this case it is more natural to return a pointer to indicate that the value must be deleted by the caller.

The end result of returning such values as simple pointers is a program with calls to **delete** spread through it and which almost certainly contains errors in its memory allocation. For example, if we are using *Shape* as an abstract base class it is all too easy to produce code like:

```
class Shape
{
public:
    virtual Number area() const = 0;
};

class Circle : public Shape
{
public:
    Circle(Number x)
    { r = x; }
    Number area() const
    { return pi * r * r; }
private:
    Number r;
};

Shape* f()
{ return new Circle(1); }
```

which can leak memory unless care is taken. For example, consider

```
Number g()
{
    Shape* s = f();
    Number a = s->area();
    // didn't call delete - disaster!
    return a * 2;
}
```

In a large program, such leaks are almost unavoidable when pointers are used in this way. The only way of avoiding the problem is to outlaw returning pointers into the heap (which makes *f* difficult to write) or to make sure that the storage is freed automatically.

Basic pointers

What is needed is a smarter pointer which can delete the object pointed to when it is no longer required. Templates and some of the newer language features provide the facilities needed to encapsulate a pointer. The minimal implementation is:

```
template<class T> BasicPointer
{
public:
    BasicPointer(T* x)
    : ptr(x) { }
    ~BasicPointer()
    { delete ptr; }
    T& operator*() const
    { return *ptr; }
    T* operator->() const
    { return ptr; }
private:
    T* ptr;
    // Suppress the default versions
    BasicPointer(const
                        BasicPointer<T>&);
    const BasicPointer<T>& operator=(
             const BasicPointer<T>&);
};
```

where we must suppress the copy constructor and assignment operator since the default memberwise versions will result in erroneous calls to **delete**. This class is useful in itself, since we can write

```
BasicPointer<Shape> p(f()));
```

and be sure that the destructor will be called when the pointer *p* goes out of scope. Note that it is not possible to write this as

```
BasicPointer<Shape> p = f();
```

since this is equivalent to

```
BasicPointer<Shape> p(
         BasicPointer<Shape>(f()) );
```

and the copy constructor call involved in the initialisation of *p* is not accessible, even though many compilers can optimise the call away.

Adding a copy constructor

We need to provide a working copy constructor if the class is to be of practical use; this would allow functions to return smart pointers. The basic problem is that the default implementation of the copy constructor results in multiple calls to **delete** for a single allocation. This can be prevented in two ways; the copy constructor can produce a new object to which the copy can point, or the existing object must be shared and deleted only when all active pointers have been destroyed. The first approach can be coded as

```
template<class T>
BasicPointer<T>::BasicPointer(
    const BasicPointer<T>& x)
: ptr(x->clone()) { }
```

where the class *T* must provide a clone operation. It is not sufficient to write

```
template<class T>
BasicPointer<T>::BasicPointer(
    const BasicPointer& x)
: ptr(new T(x)) { }
```

since *T* may well be an abstract class.

However, this deep copying is not compatible with the behaviour of a traditional pointer since those are copied shallowly. The other alternative is more promising; all we need is a simple count of how many times a given object is pointed to; when the count reaches zero, the object can be destroyed. If the count and its manipulation is delegated to the object pointed to, a suitable definition of *Pointer* is given by

```
template<class T>
class Pointer
{
public:
    Pointer(T* x)
    { set(x); }
    ~Pointer()
    { clr(); }
    Pointer(const Pointer& x)
    { set(x.ptr); }
    const Pointer& operator=(
                   const Pointer& x)
    {
      if(this != &x)
      {
        clr();
        set(x.ptr);
      }
      return *this;
    }
    T& operator*() const
    { return *ptr; }
    T* operator->() const
    { return ptr; }
    int null() const
    { return ptr == 0; }
private:
    T* ptr;
    void set(T* x)
    { ptr = x; if (ptr) ptr->inc(); }
    void clr()
    { if (ptr) ptr->dec(); ptr = 0; }
};
```

The *set* method connects the simple pointer to the underlying object while the *clr* method breaks the connection. Given these, the constructors, destructor and assignment operation are comparatively simple. (Exercise: Convince yourself that the test in the assignment operation is sufficient to avoid aliasing problems and premature deletion.) Not only does this give the expected pointer semantics, it also avoid the potentially expensive clone operations. We can then rewrite the problematic functions given earlier as

```
Pointer<Shape> f()
{ return new Circle(1); }
Number g()
{ return f()->area() * 2; }
```

and be sure that the storage allocated will be freed as soon as it is no longer required (in this case during the destruction of the temporary at the end of *g*.) No increment, decrement or arithmetic operations are provided for *Pointer*, nor is a direct conversion to a simple pointer available; the former are not sensible since the pointer is expected to refer to a single, dynamically allocated value and the latter is too dangerous.

Counter as a class

We now consider the *inc* and *dec* methods which *T* must provide if we are to form *Pointer<T>*. The object pointed to is expected to come into existence with no smart pointers connected to it and to maintain an internal count which is modified by these two methods. When the count reaches zero from above, the object is allowed to deallocate itself. These basic properties can all be captured in a single class

```
class Counter
{
public:
    Counter()
    { count = 0; }
    virtual ~Counter()
    { assert(count == 0); }
    void inc()
    { ++count; }
    void dec()
    {
      if(--count == 0)
        // need the virtual destructor
        delete this;
      }
private:
    int count;
};
```

This simple class can then be used as a *mixin*. For example, if we are exploiting polymorphism through pointers to the abstract base class Shape, then we might have

```
class Shape : public Counter
{
public:
    virtual Number area() const = 0;
};
```

and it is then reasonable to form *Pointer<Shape>*.

Efficiency

An obvious question is how expensive these smart pointers are in comparison to traditional pointers. In terms of speed, the *inc* and *dec* operations are called frequently in typical applications and often need to be inlined to be effective. In terms of space, all objects which can potentially be pointed to are increased in size by the storage needed for the counter and possibly by the space needed to store the vptr for the virtual destructor. Typically this last cost can be ignored since a base class used to exploit polymorphism should already have virtual methods (including a virtual destructor). In practice, the reduction in memory leaks in programs using these two classes more than covers the time and space overheads.

Counter and const

The implementation for *Counter* given above is not quite sufficient. If we attempt to form *Pointer*<**const** *Shape*> to replace **const** *Shape******, then we find that the *inc* and *dec* methods cannot be called for a constant object; it is necessary to modify them to:

```
void Counter::inc() const
{ ++(((Counter*) this)->count); }
void Counter::dec() const
{
    if(--(((Counter*) this)->count)
                                == 0)
      delete (Counter*) this;
}
```

or to the corresponding construct using the new **const_cast** notation. Conceptually, modifying the count maintained in the object does not modify an observable property of the object, so the methods which do so can be **const**.

Other applications

Although the *Pointer* and *Counter* classes were produced to help support the use of polymorphism, they can also be used to encapsulate memory management in many other circumstances. For example, consider the construction of a singly linked list, modelled on those found in Lisp. Such lists are built from the empty list (which is returned by the default constructor) by adding elements to the front of the list using the **cons** operation. The operation **null** indicates if a list is empty, and if not it can be split into the first element and the remainder of the list using **car** and **cdr** respectively.

```
template<class T> class List
{
    struct Node : public Counter
    {
      T             data;
      Pointer<Node> next;
      Node(const T& d,
           const Pointer<Node>& n)
      : data(d), next(n) { }
    };
    Pointer<Node> ptr;
    List(Node* x)
    : ptr(x) { }
    List(const Pointer<Node>& x)
    : ptr(x) { }
public:
    // Default assignment operation
    // and copy constructor are
    // acceptable.
    List()
    : ptr(0) { }
    friend List cons(const T& x,
                     const List& y)
    {
      return List(new Node(x, y.ptr));
    }
    int null() const
    { return ptr.null(); }
    T car() const
    { return ptr->data; }
    List cdr() const
    { return List(ptr->next); }
};
```

In less than forty lines of code, this is sufficient to provide the basic expressive power of Lisp lists in C++. Lists are automatically deallocated when they are no longer needed, even though there are no explicit calls to a destructor in the class. (Exercise: Is the default destructor for *Node* acceptable? Is it **virtual**?) The traditional definition of list length can be written directly in C++ as

```
template<class T>
int length(const List<T>& x)
```

```
{ return x.null() ? 0 : 1 +
                    length(x.cdr()); }
```

or, avoiding the recursion

```
template<class T> int length(const
List<T>& x)
{
    int n = 0;
    for (List<T> i = x;
         !i.null();
         i = i.cdr())
      ++n;
    return n;
}
```

Of course the implementation using *Pointer* and *Counter* is less efficient than a more direct encoding of singly-linked lists. For example, we pay the space and time overhead of a virtual destructor for *Node* which could be avoided. Nevertheless, the gain in readability (and confidence) over a more direct implementation may be worth the increased cost.

Unresolved problems

There is one minor problem with the pointers presented here; there is no automatic conversion from type *Pointer<T>* to type *Pointer<***const** *T>* which mimics the natural conversion from T* to **const** T*. The obvious conversion function:

```
Pointer<T>::operator
             Pointer<const T>() const
{ return Pointer<const T>(ptr); }
```

works well for non-constant types, but when defining *Pointer<***const** *T>* my main compiler (Watcom) complains that a user-defined conversion cannot return its own class. This seems reasonable, but if this is true it is unclear how I can allow the conversion. Solutions are invited.

Summary

The problems caused by the fact that polymorphism needs pointers or references have been circumvented by using two small classes exploiting templates, *mixin*s and a virtual destructor. These classes have other uses; they implement a simple (shallow) garbage-collection strategy which should be adequate for any acyclic data-structure.

Bryan Scattergood

bryan@fsel.com

The C++ standards committees have considered adding some sort of reference-counted smart pointer to the standard library but so far there has not been sufficient support for this (or rather insufficient agreement on the exact details).

I'd like everyone to study Bryan's code carefully and see where you feel you might change things. Two questions which I'll ask to start you off are related to const-correctness and intrusiveness:

1. *Is the equivalent to* **const** T* *a Pointer<***const** *T> or a* **const** *Pointer<T>? Why? What does the other form mean and how well does it work?*
2. *Can you have smart pointers to library classes? If not, why not? How would you deal with this issue?*

I look forward to your responses – Ed.

A "too-many-objects" lesson
by Peter Wippell

It was great to see articles on program structure in the last *Overload*. Help is short in this area, which is a pity because it is a major hurdle in learning OOP. My response here is just to suggest a more straightforward approach to Roger Lever's article, "On not mixing it...", where a hierarchy of output management classes is created, which is then implemented in terms of existing stream library classes. Surely the stream library is complicated enough without introducing another layer of classes! Anyway it should be capable of meeting the requirement directly.

An alternative

My proposed *main* function doesn't use any new i/o classes and goes like this:

```
int main()
{
// create a file and printer object
       ofstream file("junk.txt"),
                printer("PRN");
// make a general device reference
// which can point to any device
// including cout
       ostream& device = file;
// declare record objects and send
// them to the chosen device
       Record r;
       ExtendedRecord rr;
       device << r << rr << endl;
}
```

The *ExtendedRecord* class has a *Number* field in addition to the *Record*'s *string* field and inherits a virtual function *write***()**, which is called polymorphically from *ostream***&** **operator<<(***ostream***&,** *Record***&)**. I thought this idea was original until I read Francis's latest article in EXE yesterday!

If required, extra screen handling capability could be obtained by deriving a *constream* class from *ofstream* and Borland provide such a class to show you how. Similarly, I have added some extra printer control, by deriving a new *Printer* class:

```
class Printer : public ofstream
{
public:
    Printer() : ofstream("LPT1")
    { if (0x90 != biosprint(2,0,0))
        cerr <<
          "Cannot access printer.\n";
    }
};
```

And given it underline capability with a parameterless manipulator:

```
ostream& set_underline(ostream& os)
{
    if (dynamic_cast<Printer*> (&os))
    {
      os << UNDERLINE_ESCAPE_CODE
         << 1;
    }
    return os;
}
```

Surprisingly, the file descriptor seems not to be available to users of *ofstream*. It's protected. So the *Printer* class had to be invented to make sure that escape codes aren't sent to devices other than printers.

Two points of detail

I had to use the BIOS to check printer status. The condition, **if (!***printer***)**, which some books suggest, doesn't seem to work on my PC. I find that sorting out this sort of problem, can waste a lot of time, especially when you try to be too clever with the stream library!

The draft ISO C++ *string* class is more appropriate for a record field than a *strstream*. A pitfall of *strstream* is that every time you call **char*** *strstream***::***str***()**, as is done in the article, memory is allocated using **new**. So, if you don't want memory leaks, you must delete the resulting **char*** when you have finished with it.

I have supplied the complete code in case anyone wants to improve it.

Peter Wippell

The code will be on the next CVu disk and will be available on Demon for ftp shortly after – Ed.

Stop press! Just as Overload was going to press, Peter supplied a revised version of his article which tackles the problem of accessing the file descriptor in an interesting manner – I will feature that in Overload 9.

editor << letters;

Hi Sean,

I moved at the end of Dec '94. I informed Francis but obviously failed to get the new address into the *Overload* address database. I discovered issue 7 after going over to the old address and scrabbling through a pile of old mail for previous tenants that the new tenants hadn't forwarded!

Well done; after spending around two hours reading through it, I'm glad I did. Some thought provoking stuff. And thanks.

Warm regards,

Fazl Rahman

fazl@hadronic.demon.co.uk

I'm glad you enjoyed Overload 7 – I hope Overload 8 finds its way to you more directly!

Dear Sean,

My tuppennyworth on Francis's "polymorphic objects" – isn't a circle just an ellipse with the eccentricity attribute set to 1 (or whatever the proper value is)?

Regards,

Ian Horwill

ian@horwill.demon.co.uk

If you fix an attribute of the base class inside a derived class then any operations that change that attribute will not accept an object of the derived class in place of an object of the base class – one of the basic premises of the "is-a" relationship. See Kevlin Henney's article in this issue.

The following two letters follow on from Dave Midgley's letter in Overload 7 and are extracted from email conversations – hence my answers are interspersed – Ed.

Sean,

Dave Midgley's letter in *Overload 7*, and your reply, show why something along the lines of '**public readonly:**' would be a popular addition to the language. However, in the absence of this, something similar may be achieved using macros:

```
#define READONLY(Type, Name)
       \
    public:                         \
       const Type& Name() const    \
       { return m_ ## Name; }      \
    private:                        \
       Type m_ ## Name
#define READWRITE(Type, Name)      \
    public:                         \
       const Type& Name() const    \
       { return m_ ## Name ; }     \
             Type& Name()          \
       { return m_ ## Name; }      \
    private:                        \
       Type m_ ## Name

class TestClass
{
      READONLY  ( int, ReadOnly  ) ;
      READWRITE ( int, ReadWrite ) ;
};

void CMainDialog::OnClickedButton1()
{
      TestClass TC;
      int x = TC.ReadOnly();
      int y = TC.ReadWrite();

      TC.ReadOnly  () = x; // Error
      TC.ReadWrite () = y;

// Demonstrates member access on
// a const object
      const TestClass& TC2 = TC;
      y = TC2.ReadWrite();
      TC2.ReadWrite() = y; // Error
}
```

A couple of points should be noted about the macro itself: firstly, I have omitted the final semi-colon in the macro definition, on the assumption that the user will supply it. Secondly, the macro is designed to leave the class access state as **private**.

For simple types, return of a direct copy may be more efficient than the **const** reference, but I would expect a decent optimiser to be able to cope with this.

Also note the override on **const** type of the *READWRITE* access functions. This allows one to read class members of **const** objects, but only to write to members of non-**const** objects.

Unfortunately, if *TestClass* wishes to modify its *ReadOnly* member, it has to refer to it by its mangled name (i.e., *m_ReadOnly*).

> *Apart from my knee-jerk reaction ("don't use macros") this is quite a neat idea.*

Find me a template solution, and I'd be more than happy. Actually, find me any solution excluding macros. PLEASE!

> *[On always leaving the access in state **private**] Hmm, liveable I guess.*

Well, the alternative was leaving access **public**. My personal preference is to keep access restricted unless loosened. This matches Bjarne's '**private** by default' of **class**, as opposed to the '**public** by default' of **struct**.

Part of what I'd *like* (I don't demand) would be a meta-language, such that one could loop over all members calling their serialisation functions, for instance. I don't know what the language would be, but I don't think it could be called C++ any more.

> *[On using the mangled name in member functions] Perhaps a DECLARE_READONLY macro (to replace READONLY above) and a new READONLY macro that just glues m_ on the front?*

What I was trying to do was define a member that need *only* be accessed using accessor functions, with a sensible name thereof.

My first solution for *READONLY* had a private member function returning a non-const reference (i.e., the same one as available in the *READWRITE* macro, but **private**). Unfortunately, of course, it tends to hide the **public** one returning the **const** reference when dealing with non-**const** objects.

Once I'd been forced into using another name, I decided that the internal use of the member's real name was no worse than a *SetMember* **private** function. YMMV.

Alan Bellingham

alan@doughnut.demon.co.uk

"Be prepared. Always carry a rose bush."

I include Alan’s .sig because it appeals to my sense of humour.

Dear Sean,

I just received the April issue of *Overload*. On pages 34 to 35, Dave Midgley asks whether it is possible to define a member variable which is **private** for writing but **public** for reading. It can be done by means of a **public const** reference as shown in the example below, taken from Dave’s letter and modified:

```
class fred
{
public:
       fred(int x)
       : readOnlyAttribute(privNum),
       // initializing the reference
         privNum(x) { }
       void changeAttribute(int z)
              { privNum = z; }
       const int& readOnlyAttribute;
private:
       int    privNum;
};

fred aFred(100);
cout << aFred.readOnlyAttribute
     << endl; // reading works!
aFred.readOnlyAttribute = 0; // error!
              // writing doesn't!
              // (const!)
aFred.changeAttribute(88);   // change
              // with method works
```

I admit that this approach is not very elegant. An inline-function is normally better in all cases, where possibly a computation shall be added later. It is easy to change a function, but it is not easy to change all locations where the variable is used.

Yours is one of the more elegant solutions I received, although it still has the problem that the external name doesn’t match the internal name.

This is not a real problem. The external name has to be chosen carefully, and the internal name can’t be seen anyway by *users* of the class.

The resemblance of both names is important only for the class *developer*. For such cases I prefer the underscore-notation which is used by Myers, Gamma et al.

However, in the example I sent to you I wanted to express the attribute properties “read-only” and “private” in the names of the variables.

Best regards,

Uli Breymann

uli.breymann@m2tek.north.de

I agree with Uli’s point here: at the end of the day, the importance of the public names is user-centric, which should be the higher priority.

Dear Sean,

Thank you very much for some very stimulating reading in your *Overload*s.

Here is a contribution which tries to build on Roger Lever’s article in the last issue. If acceptable, I suppose it could be treated as a letter or a short article.

About your remarks on streams, “Borland C++ Object Oriented Programming” by Ted Faison gives a number of examples, deriving classes from parts of the stream library.

My feeling is, however, that use of the stream library is limited mainly to file i/o, because Dialogs, Windows, and customised printer classes take the place of streams in real modern applications. String streams, though, are very useful for formatting output. See Adrian Fagg’s article in the current CVu. I was concerned to read that they were being discarded.

All the best,

Peter Wippell

Just to clarify, strstream was deprecated in Valley Forge (November ‘94) which means that it remains part of draft standard C++ but might be removed in a future revision of the standard. The reason for deprecating strstream was that stringstream provides effectively the same functionality using the standard string class instead of the error prone raw ***char****.

Peter’s article appears elsewhere in this issue.

++puzzle;

In *Overload 7*, I asked “What is the longest sequence of distinct keywords and reserved words possible in a valid C++ program?”. Unfortunately, I was completely underwhelmed by responses so no-one wins the prize (and I’m not even going to tell you what you missed out on).

Anyway, here’s a program containing the longest sequence that I know of:

```
// keywords.cc      lastmod 12 Nov 94  SAC
//           created 11 Nov 94  SAC
//
// copyright: (c) 1994 Jonathan Caves, Sean Corfield, Fergus Henderson,
//           Mats Henricson, Steve Rumsby, Erwin Unruh
//
// purpose:  to write a valid C++ program containing the longest sequence
//           of unique keywords/reserved words
//
// history:
//     12 Nov 94  SAC  Added comments and tidied up the code
//     11 Nov 94  SAC  Initial 25 keyword version
//     11 Nov 94  ---
//     Started with:
//     explicit virtual inline operator const volatile unsigned long int*();
//     Realised you could use bitand to replace * (i.e., change pointer to a
//     reference type) and took it from there...

#include <iostream.h>
#include "keywords.h"     // overloaded operator definitions

struct X {
     int f()
     {
          if(0)
               return 0;

// --------------- all these *really* are keywords! ---------------
        else        do          return      throw       sizeof
        true        bitor       compl       not         new
        const       volatile    unsigned    short       int
        not_eq      false       and         bitand      operator
        and_eq      or_eq       this        or          static_cast

                    <B&>(b), 0;
               while (1);
     }
};

int main() {
     try {
          X     x;
          x.f();
     } catch (int) {
          cout << "Hello world!\n";
     }
     return 0;
}
```

I’ll leave the contents of keywords.h as an exercise for the reader!

Hopefully, *Overload 9* will see the return of the *Questions & Answers* section unless, of course, you no longer have any holes in your C++ knowledge...

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

Books and Journals

Forthcoming reviews will include “Taligent’s Guide To Designing Programs”, Barton & Nackman’s “Scientific and Engineering C++” and in *Overload 9*, “Design Patterns” by Gamma, Helm, Johnsson & Vlissides. If any C++ experts want to get involved with in-depth reviews of books old or new, please drop me a line.

Writing “Industrial Strength C++”

by Mats Henricson

I asked Mats Henricson and Erik Nyquist – authors of the forthcoming book “Industrial Strength C++” – to write about how the book came to be written and what is involved in writing a “public” coding standard. This article was their response – Ed.

Background

In early 1990, C++ was chosen as the implementation language for a huge telecom project at Ellemtel Telecom Systems Labs in Stockholm. Ada was rejected since it wasn’t object-oriented, and Eiffel fell through for commercial reasons. A small group of people was formed to discuss general C++ issues. The first task for the group was to review and improve a first version of a programming standard. The result was a completely new document that Erik and I maintained.

Then, in 1991, there was a discussion about programming standards in the newsgroup `comp.lang.c++`. I wrote a message describing the structure of our document. Suddenly I got an email from Bjarne Stroustrup asking if he could have a look at the document. The fact that it was written in Swedish was no problem to him, since Danish is close enough to Swedish. I got cold feet and had to ask around for advice within the company. After some lobbying by Erik and I, the document was put into the public domain. Shortly after, it was translated into English by a consultant, Joseph Supanich, and put up for anonymous ftp.

Why was it so successful?

This document spread like wildfire across the world, and I still get several emails a week from people asking for new versions or other questions. I have a list from August last year with names of companies or organizations that I *know* have the document. It lists 35 universities, 4 banks, 18 laboratories and 89 other companies.

Many people know how hard it can be to write a company wide programming standard for a language as complex as C++. Instead of endless internal debates they could just pick something for free from the net. The copyright notice gave people the option to edit the document as long as the original copyright notice was intact. This way they could, with only a small amount of editing, get something that was good enough.

The second reason was probably that we explicitly listed all the guidelines instead of having them somewhere in a block of text, entangled with discussions and code examples. Another reason was probably that we differentiated between rules and recommendations (R&R). Everyone should follow the rules, while the recommendations were more “good ideas” that should be followed unless there is a good reason not to.

The document eventually ended up at Prentice Hall and we got an email asking if we would like to rewrite it as a book. We accepted without really having any idea of the implications. Now, more than two years later, when it seems like we are actually pretty close to wrapping the whole thing up, it is time to try to find out why it has taken us such long time.

It is a constantly changing C++ world

C++ has changed in many ways during the last couple of years, which has been problematic for us. What was previously looked upon with suspicion is now widely accepted, like multiple inheritance. We have constantly changed our minds in quite a few areas, while others are so new that hardly any experience exists anywhere (e.g., RTTI and namespaces). We have often been worried because we haven’t had many R&R for templates but how do you find good R&R without considerable experience?

Areas like mixin-programming, OO design patterns and the STL library have given a new twist to our view of the world. We previously had a recommendation saying that virtual inheritance should be avoided but practical experience has shown that it is sometimes the easiest way to implement a derived class. For more information on this subject, see “Scientific and Engineering C++” by Barton & Nackman (Addison Wesley, ISBN 0-201-53393-6).

To be reviewed in a future Overload – Ed.

Formulating R&R is very hard!

It can sometimes be painfully hard to find the best possible angle for a rule or recommendation. For example, which of the following wordings are best?

1. Do not modify string literals.
2. Only use **const char**-pointers to access string literals.

They basically deal with the same thing, but from two different view-points. The first points out that modifying string literals gives you undefined behaviour:

```
char a[] = "abc";
a[1] = 'x';  // undefined behaviour
```

The second tells you how you should avoid such intended changes:

```
const char a[] = "abc";
a[1] = 'x';  // compile-time error
```

Unfortunately you can cast away const anyway:

```
((char*)a)[1] = 'x';       // not a word
              // from the compiler!!
```

Another problem has been whether or not we should just list the base R&R and avoid all corollaries. A rule saying that you should avoid all undefined, unspecified and implementation-defined parts of C++ makes sense for portability reasons. Unfortunately such a large rule makes all other R&R in this area completely unnecessary, which is not very wise since many of these issues need to be warned about explicitly (e.g., do not depend on the order of evaluation of arguments to a function).

Sometimes it is very difficult to define the words necessary for formulating a R&R. We have this recommendation:

> *Before throwing an exception from a member function, make certain that the class invariant holds and, if possible, leave the state of the object unchanged.*

The problem here is that it is painfully hard to find rock solid definitions of the words “state” and “invariant”. We can be pretty sure that our definitions will not be the same as other authors’ definitions.

What is the best structure for the book?

The problem we have wrestled with most is how to structure the book. In the beginning, we had grandiose ideas of a chronological structure that would begin with R&R on analysis and design and end up with stuff on testing. Unfortunately neither Erik nor I am experts on OO testing, nor OOA/OOD for that matter. I can tell you that it was with considerable unease I started to write about testing! So, we decided to settle for areas we knew well, i.e., the language C++ itself. It will, for example, not contain anything about testing, code reviews, OOA/OOD or metrics.

After reading the book “Safer C” by Dr Les Hatton (McGraw-Hill, ISBN 0-07-707640-0) I got carried away by finding out that the international standard ISO 9126 defined six aspects of software quality that seemed to fit pretty well with our current structure of the book:

1. Functionality
2. Reliability
3. Usability
4. Efficiency
5. Maintainability
6. Portability

The problem was that “pretty well” was not good enough. Another problem was that it seemed like most rules would go into chapters 2 and 3, while chapters 1 and 4 would be pretty empty, which makes a rather strange structure!

“Safer C” also made me browse through ISO 9000-3, ISO 9001, ISO 9126, “The Capability Maturity Model” (CMM) and other standards or pseudo-standards in a fruitless and disappointing search for a description of what a programming standard should contain. The only thing we found out was that it seems that the issues of programming style should not be a part of the main standard. Our approach will be to put stylistic issues in an appendix. However, “Safer C” did not give us a good definition of “style” – a fact that every now and then throws me and Erik into long discussions.

R&R must follow guidelines

R&R must not only warn against blatant bugs, but also stop people from doing dangerous stuff, even if it is sometimes valid, e.g.,

> *A concrete class should not inherit from another concrete class.*

Should it always be possible to check R&R with a tool like Programming Research's QA C++? Well, we would like to, but the world is a bit too complicated. Unfortunately it seems that rules that cannot be checked by a tool are followed much less often than checkable rules (see "Safer C"). We have chosen to use the same criteria as the public domain document for deciding between making something a rule or recommendation.

R&R should basically be valid both for rookies and experts at the same time. This is sometimes mind-bogglingly difficult to fulfill. Most programmers should not have to deal with virtual inheritance, virtual assignment operators or pragmas, but how can you ban such dangerous features when some people need them badly? Lengthy discussions and descriptions are needed in many cases to make sure the reader is aware of the problems with special features.

Another problem is that R&R should be valid and relevant for programming on all possible platforms. Banning all extensions to C++ for portability reasons would stop **__huge** for DOS/Windows programming, which would be fatal since Windows programmers will probably be the vast majority of the customers of the book. Banning signal handling makes sense for a completely portable UNIX application, but there are no signals on Windows!

R&R may be perfectly valid but still not make it into the book since they are just too obscure for most programmers. Like banning the use of bitfields, which makes good sense, but most programmers would never dream of using them. That is why we don't have such a recommendation. A 500 page long C++ standard would never be used.

Something that has given us a lot of headache is the problem of finding good examples for describing particular rules or recommendations. Do we really know what we are talking about if we cannot produce anything but a completely pathological example? Will readers just swallow and digest text without code examples?

Finally, the fact that there are two of us working on this project has delayed it a lot. It would have been published a long time ago had it been written by just Erik or just me, but the quality would not have been as good. By having two authors we stop each other from going astray into something that is either not particularly fruitful or important. We also believe that the set of R&R in the book are so carefully worded after endless iterations that they should be as bullet-proof as anything gets in this world.

Mats Henricson

mats.henricson@eua.ericsson.se

Erik Nyquist

erny@enea.se

You can be sure that when "Industrial Strength C++" is finally available, it will be reviewed here! – Ed.

Vendor Focus

In this issue, I turn the spotlight on a C++ compiler-writer. If you'd like to see a particular vendor under the spotlight – especially if you are willing to conduct a virtual interview – let me know.

Edison Design Group
a virtual interview by Sean A. Corfield

Edison Design Group is a small American company that is big "behind the scenes" in the C++ world. They write compiler front-ends for many well-known companies. This article is adapted from an email interview conducted with Steve Adamczyk.

Steve founded Edison Design Group in 1988 with a partner who left about a year later. They currently have three staff: Steve, Mike Anderson, and John Spicer.

Can you tell me a bit about yourselves?

The three of us met at a company called Axxess Information Systems back in 1982. John and Mike were working there, and Steve came there from a company called Advanced Computer Techniques (ACT). About a year later, Axxess foundered, and the three of us went (back) to ACT. ACT was, among other things, a compiler house, selling compilers to companies like computer manufacturers. Mike left ACT in 1986 and moved to New Hampshire, going to work for DEC. Steve left in 1988 and founded EDG, and John left shortly thereafter and went to work for AT&T. Mike joined EDG in 1990, and John in 1992.

We have combined programming experience of about 63 years, an average age of about 42, an average height of about 6’ 2”, three wives, and four children (three boys and one girl).

Is it true you all work from home?

Yes, it’s true. At the end of 1989, Steve was looking to hire someone to help him with development of a Fortran front end. Mike was the obvious choice: he and Steve had worked together previously, on Fortran among other things, and Mike was at that point working on Fortran at DEC. The only snag was that Mike was in New Hampshire and liked it there and therefore was not going to move back to New Jersey. So Steve suggested that we try it with Mike working from his home. What started as an experiment ended up as our preferred way of doing things. John and Steve work from their homes in New Jersey, and Mike from his home in New Hampshire.

That must be pretty different from the average office – does it cause any problems?

After a bit of practice, it works great. You have to learn to do your work and your socializing over the phone and by e-mail, but if you can adapt to that, this way of working is very convenient: there’s no commuting and your schedule can be quite flexible to deal with family obligations. Of course, the phone bills are large, but they’re less than the cost of renting an office.

It’s also true, though, that this wouldn’t work nearly so well if we weren’t good friends too. Knowing one another’s strengths and not-so-strengths (surely there are no actual weaknesses), having a lot of trust in each other, being able to communicate honestly and well together have been a key to making this work.

So why did you start ‘yet another’ compiler company?

We had been in the compiler business with ACT, so it’s the business we knew. But we didn’t just start yet another compiler company; we started a compiler front end company. We decided we wanted to stay small and technical, and that suggested that we should pick just one part of the compiler business and do that as well as we could. As it turns out, by doing only front ends, we have made it possible for us to sell to companies that would ordinarily be thought of as our competitors, i.e., compiler vendors.

Why front-ends?

If you can do it better than anyone else and make money at it, why not :-) ? The front end of a compiler is a nicely separable piece. Doing one requires a substantial investment in development time and maintenance, and also a substantial investment in learning about the language at the level of detail required to write a front end. Our customers can get a front end from us, in source form, for less than they can develop it themselves, and we take care of the ongoing updating of the front end to track the evolving language. That frees them to concentrate their efforts on code generators, optimizers, and libraries. A C++ front end, in particular, is a very large wheel to reinvent.

Can you name some of your users for the readers?

There are 30 licensees of our C++ front end. Of those, we can name Silicon Graphics, Cray Research, Novell/Unix Systems Group, Tartan, The Portland Group, Kuck & Associates, CenterLine Software, Siemens Nixdorf, Apogee Software, Tera Computer, and Visual Edge. Only a few of our licensees have products out in the field at this point, but quite a few more will be releasing products during 1995.

Hmm, plus nearly twenty others – an impressive list! What about the software? Is it all written in C++?

No, it’s written in ISO C.

Why's that? Surely you need development experience in C++ to write a good C++ compiler?

Well, it is a bit embarrassing, but... We're really just C programmers who know a lot about the C++ language rules.

Our C++ front end is based on a C front end begun in 1988, and adapted into a C++ front end beginning in 1991. The C front end was written in C, naturally enough, and it was gradually changed into the C++ front end, so we lost whatever opportunity we might have had to start over in C++.

We do find we could make use of things like constructors and destructors, but on the other hand having the front end in C makes it quite portable and avoids a bootstrapping step.

What've been the hardest C++ features to implement and why?

Templates have been a challenge, because it's hard to do much with a template until you do an actual instantiation, and yet you want to do certain things before that to improve error diagnosis.

Name lookup issues have also been tricky.

Beyond that, the hardest problems have been compatibility issues. We provide a mode that offers fairly complete compatibility with the AT&T/Unix System Laboratories/Novell cfront, and it's been quite an adventure to duplicate some of cfront's behaviors.

You picked on name lookup – could you elaborate on that, please?

We haven't done namespaces, or template name binding. There's plenty to keep one busy in the other name lookup issues, though. C++ is very rich in local contexts that change the lookup rules: ***A*::*x***, **struct *x***, **void *A*::*f*() { ... *x* ... }**, and so forth. Some of the variations in lookup can make the difference between a name being a type and being a nontype, which can have an effect on how the program is parsed. It gets even more interesting when you combine these cases with things like lookahead for disambiguation: you have to look ahead and recognize the contexts and modify the symbol table lookup appropriately, but make no permanent changes in the symbol table, since the the disambiguation scan is just exploratory.

Since you don't deal with code generation, has RTTI had any impact on what you do?

We don't do code generation, but we have to make sure that our intermediate language provides the right information so others can do it. We also have and use a C-generating back end for our testing (it allows one to compile C++ to C), and therefore we have to do some kind of implementation of every language feature. Initially, we just followed the cfront implementations of features. More recently, we've gotten into language features that were never implemented in cfront, and we've had to design the runtime representations for those. RTTI did require some work, but it wasn't as bad as, say, exception handling.

What would you change about C++?

Well, it would be nice if it weren't such a big language, but it's hard to decide what one would choose to remove. You get used to it. It would have been helpful, however, if all these features had been implemented somewhere before being written into the standard.

What about the library? It's very large – any comments on that?

That's not really much of an issue for us, since we don't provide a library. We have been getting pressure to deliver the language features needed to write a standard library, so that library developers can use our front end. We expect to have those features out in the next few months (i.e., mid-1995).

One fear we have is that when programmers start using complicated template libraries like STL they're going to be getting cryptic error messages when the templates fail to match or instantiate. Simple programmer mistakes are going to produce pages of error messages from deep in the bowels of the library headers. We're doing what we can to provide clear error messages, but there's only so much a front end can do.

My experience with STL bears this out – compilers need to get a lot more helpful! EDG's very involved with the standards process – how confident are you about the schedules?

We seem to have misplaced our crystal ball, so we can't help you on the schedules. There's lots of work that still must be done on the draft to make it a standard. It's hard to know if that can be done in time.

Do you have any other comments to make about the joint ISO / ANSI process?

Of the standards process in general, we'd have to say that it's the worst possible way of producing standards, except for all the others. It does produce results eventually, and the safeguards built in help avoid the worst problems. Unfortunately, they don't prevent language bloat.

What's EDG going to do next?

We'd love to be working on some other front end (we're coming up on five years on this one), but for the next year or two it's clear we're going to be working on C++ full time. When we've wrapped this one up to our satisfaction, we'll see what the market seems to want.

Presumably there's only a small number of people that you can sell a front-end to?

As for the potential number of sales, we keep being surprised; we would not have predicted that we could sell 30 licenses, and the C++ market shows no signs yet of slowing down. And we're just starting to be known on your side of the Atlantic!

Which for a three man company working from home is quite an achievement! Thankyou for your time.

Steve Adamczyk

jsa@edg.com

Mike Anderson

rma@edg.com

John Spicer

jhs@edg.com

News and 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 Ships Visual C++ Version 1.52

Introduces OLE and ODBC programming to entry level C++ developers!

Responding to the tremendous success of Microsoft Visual C++ Standard Edition, Microsoft is upgrading its 16-bit product to Visual C++ development system for Windows version 1.52. Visual C++ version 1.52 adds support for OLE and ODBC programming through MFC classes and wizard technology, and will be available for an estimated retail price of only £66.00 + vat.

"We are excited by the rapidly growing interest in C++ development for Windows among entry-level developers," said Andrew King, European marketing manager for the Developer Division at Microsoft. "While the professional community of developers for Windows is successfully developing powerful 32 bit Windows-based applications using Visual C++ 2.0, there is strong interest among entry-level C++ developers to learn OLE and ODBC programming using MFC."

Features and benefits:

The Visual C++ development system version 1.52 provides the following:

- Support for both Windows and MS-DOS programming
- Latest Microsoft optimizing 16 bit C and C++ compiler
- New 16-bit MFC (the industry-standard application framework for programming Windows) supporting OLE, ODBC, MAPI, Windows Sockets, property pages (tabbed dialogs) and floating toolbars
- More than 20,000 lines of MFC code specifically to enable easy OLE development
- ODBC database drivers for Microsoft Access, Microsoft SQL Server, the FoxPro database management system, Paradox, dBASE, Microsoft Excel and Btrieve.
- The MFC Migration Kit, helping developers migrate their C code to MFC
- An extensive help system

- More than 2,500 pages of printed documentation in the box, including a step by step C++ tutorial
- Eight of the most popular games from the Microsoft Windows Entertainment Packs

System requirements:

The system requirements for the Microsoft Visual C++ development system version 1.52 are:

- An IBM-compatible personal computer running MS-DOS 5.0 or higher and Microsoft Windows version 3.1 or higher
- An Intel 80386 or higher processor, with 8 MB of available RAM, a CD-ROM drive, and a VGA or higher-resolution adapter and monitor
- A hard disk with enough space to install the options needed: 40 MB of available storage space minimum using the CD-ROM installation option; 80 MB of available disk space for the full configuration.

Pricing and availability:

The Microsoft Visual C++ development system version 1.52 is available for an estimated retail price of £66.00 + vat. An Academic Edition is also available at a discounted price to students for only £32.00. To obtain Visual C++ version 1.52, customers should contact their usual software dealer or call Microsoft on 0345 00 2000.

“Entry level” developers will be forgiven for their disappointment at the continued lack of templates or exception handling in Microsoft’s 16-bit offerings – Ed.

Credits

Founding Editor

Mike Toms
miketoms@calladin.demon.co.uk

Managing Editor

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

Production Editor

Alan Lenton
yeti@feddev.demon.co.uk

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
mradford@devel.ds.ccngroup.com

Copyrights and Trademarks

Next Issue

In the August issue, *The Draft International C++ Standard* will report on the July meeting of the joint ISO/ANSI C++ committee and discuss some of the issues arising from the public reviews. *C++ Techniques* will, no doubt, continue the discussion of multiple inheritance. *Books and Journals* will look at the “Gang of Four” Design Patterns book. *Product Reviews* will cover S-CASE: a multi-platform OO case tool based on Booch’s notation. The rest is up to you!

Copy deadline

All articles intended for inclusion in *Overload 9* (August) must be submitted to the editor by July 3rd.