ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 15

August/September 1996

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

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

£3.50

Contents

Editorial ***3***

Software Development in C++ ***4***

Some questions about OOD *4*

Explorations around a linked list *6*

Go with the flow *11*

So you want to be a cOOmpiler writer? – part VI *15*

The Draft International C++ Standard ***19***

The Casting Vote *19*

Making string literals constant – a cautionary tale *20*

C++ Techniques ***24***

Circles and ellipses revisited *24*

The Standard Template Library – sorted associative containers part 1 set & multiset *27*

The return type of member functions *29*

/tmp/late/ Constraining template parameter values* *33*

editor << letters; ***35***

Reviews ***36***

Java in a Nutshell *37*

News & Product Releases ***39***

OMT User Group Seminar *39*

ACCU and the 'net ***40***

Editorial

Designed for abuse

I am inspired by something Bjarne Stroustrup told me in Stockholm at the recent ISO C++ meeting. Or rather I'm horrified. And surprised. But first, let me digress.

Look around you and consider how much of your world is open to abuse. Lots of things can be used as weapons that were clearly not designed as such. On a less dramatic scale, think of the things pressed into service as screwdrivers. And yet we generally have to be somewhat inventive to come up with these unusual uses and many times are driven by the necessity of what we have on hand.

Machinery is generally designed to be safe in use. It often has warnings that it should only be used for its "intended purpose". You wouldn't trim a hedge with a lawn mower for example, but I'm sure it has been done. This is because manfacturers like to protect themselves against lawsuits. Witness the (hopefully apochryphal) tale of the woman who microwaved her pedigree cat after giving it a bath, thinking that a microwave was simply a different form of oven.

But what of programming languages? What are they designed for? Problem solving... but there's more than one way to skin a cat (oops!) and programmers seem rather ingenious at figuring out ways to abuse the language.

It was such an abuse that Bjarne described to me. All the more horrifying because it would never have occurred to me to do it. Redeclaring library functions in block scope with default arguments. "I'm going to be calling *fgets* lots of times here with the same arguments, I think I'll just redeclare it locally with three default arguments to save me some typing." Where's my shotgun? I was incredulous. "Oh yes," said Bjarne, "they really do do this."

Perhaps you're not surprised. Perhaps, heaven forbid, you actually do this yourself? I would hope most of you consider this to be an abuse of language features.

When manufacturers of equipment get caught out by the ingenuity of their users, they either redesign the equipment to make it safer (e.g., making the abuse impossible) or clearly label the equipment "Not intended for XYZ" (as supposedly happened in the cat-drying incident... American microwave ovens are not for drying pets!). Language standards committees have a similar label: "deprecated". In my opinion, they just don't use it enough! The suggestion was made in Stockholm to deprecate default arguments on block scope declarations. It didn't get much support and that's when Bjarne told me how common the practice is.

At least we've banned implicit **int**...

An appeal!

This issue is late but at least it's packed full of interesting articles. Why is it late? Various changes in my personal life have taken me away from home almost every weekend since early June. Since weekends are mainly when I get time to work on *Overload*, that has meant, quite simply, that I haven't had enough time to edit this issue and stay on schedule. This is likely to continue for the foreseeable future.

In addition to the continued flow of articles, what I now need to ensure the continued success of *Overload* are some volunteers to help with the bi-monthly task of editing. If you are interested in helping, or simply want to find out more about what is involved, drop me an email.

The Editor
overload@corf.demon.co.uk

Software Development in C++

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

In this issue, two thought-provoking articles from newcomers to C++ and OO: Graham Jones asks whether Object Orientation is really the Emperor's new clothes and Peter Moffatt reminds us of just how difficult C++ can be when you first try generic programming. Richard Percy begins a series that will look at C++ from the point of view of a financial applications programmer and I continue looking inside a compiler, this time concentrating on static and dynamic polymorphism.

Some questions about OOD

by Graham Jones

This article is in response to calls for contributions from non-experts. I have very little experience of writing C++, not really enough to have sensible questions, but I have read and thought quite a bit about OOD/OOA. I have read Booch's book on OOD, Gamma et al's "Design Patterns", Russell Winder's "Developing C++ Software", and many back issues of *Overload*. I have programmed in C for about six years – my serious work in the past few has been developing an OCR program for Acorn machines. Most of my questions relate to attempts to apply what I have read to this program.

OOD for OCR?

After a lot of thinking and trial designs, I don't know how to usefully apply OOA/OOD to the OCR engine in my program (using OOA/OOD for the interface is fine, but a minor issue either way). The trouble seems to be as follows.

All the interesting candidates for objects in my OCR program are abstract entities of my own invention, and as the program develops, I keep changing my mind about the nature of them. As I understand it, one the main advantages of OOD is that the class hierarchy is likely to be the most stable part of the design. Thus your idea of a car may change, your idea of a vehicle may change, but at least a car goes on being a vehicle. This is certainly not the case with any objects I might have chosen in the past as objects.

Perhaps I am choosing the wrong classes. Certainly the 'obvious' ones that you might think of for OCR – letter, word, line, etc, are pretty useless. The representation of each of these objects and the operations possible on them changes often and radically during the OCR process. I could have half a dozen kinds of 'letter' object, etc, but even then most of the work would not be methods on these classes, but conversions from one representation to another. Even worse, these classes and relations between them tend to be very unstable when the OCR algorithms get modified. In fact, such a set of classes has the interesting property that their implementations are often more stable in the face of change than their interfaces.

As far as I can gather, a more usual set of classes for a problem like OCR is to have 'expert' classes: a *LineFinderExpert*, *LetterRecogniserExpert*, *WordExpert*, etc. This is better, and it is pretty much the way I have divided the program into files. (By the way, I find it much easier to think of objects as more flexible implementations of the compilation units in C than as of extensions of C structs. Is this sensible?) It would make my program look more fashionable to use 'expert' classes but this does not 'get at' the complexity of the program. It is a fairly crude division, leaving large lumps untouched, and even so it is by no means a clean division: there is much complexity in when and how the 'experts' talk to each other. I'd say it was akin to 'simplifying' the problem of wiring a house by dividing the house into rooms: useful, but not very.

The way I tend to think of dividing the OCR process up is in terms of functions. (Remember them?) I basically see the process as made of links like this:

```
data in -> function -> data out
```

Some of them are complex enough to be called subprograms. These functional units have proved fairly stable as the program has developed. The key to this seems to be the flexible way in which such functions can be combined in new ways to try out new ideas. And they do get at the essential complexity of the process. If I wanted to describe the way the program works, I could do it in terms of these things. Trying to describe it in terms of experts sending messages to one another would make me ill.

So what should I do? Give up on OOD for the really difficult stuff? Call my functions Functor Classes (or somesuch) so I can pretend to be as trendy as everyone else? Any other ideas?

Data Conversions

Converting from one representation to another is the main focus of OCR, but similar conversions occur in many areas of programming. If you want to convert between linked list and array representations of some data, where do the functions that do the conversions go in a OO program? Does a linked list know how to make itself out of an array, or how to create an array out of itself? Or both – or neither?

> *STL containers have constructors that allow them to be constructed from any other container, or rather from any pair of iterators that define a range of objects to be contained – Ed.*

If you want to convert between different ways of writing the date, you could go via a kernel class. (See Francis's articles in *Overload* 11 and 12.) Each kind of date knows how to convert itself to and from the kernel date class. At least, I think that is what Francis intends, and it seems fair enough. But suppose I replace 'date format' with 'image format'. What if I want to convert between the dozens of ways people have dreamt up of representing images as bitmaps? What would the kernel be? Do you want to convert every format into the kernel and back? Would there be a kernel? How would you organise the convertor code now?

What is OOD?

It might seem a bit late to ask this question, but it was when I was trying to see what the OOD gurus had to say about the problems above that I began to wonder if I had got the basics right. More worryingly, I began to wonder if they knew what they were talking about either.

Grady Booch describes the 4 main elements of OOD as modularity, abstraction, encapsulation, and inheritance.

What is modularity? Booch uses "modularity" only in relation to compilation units (as far as I can see). Others use it more generally, e.g., Russell Winder moves some code (for searching for an item in a binary tree) from the binary tree into the units of the binary tree, and claims a "significant increase in modularity". What does it mean to you?

What are abstraction and encapsulation? Booch says that abstraction and encapsulation are complementary: defining the outside view and hiding the inside view of an object. Again, he seems to restrict these words only to apply to one thing, in this case objects. Surely they apply to functions, compilation units, subprograms, assembler macros...?

I also find Winder's concept of "encapsulation" confusing. Winder develops a *Complex* class, with real and imaginary parts represented by doubles. He says "that we are using a Cartesian representation should never come to the notice of the user" [p165]. His constructor allows three arguments to be passed, the third, if present, indicating that the number should be constructed from polar coordinates. "Unfortunately, this does rather give the hint, to the user of the type, that Cartesian representation is being used internally" [p169]. Is Winder trying to hide things from a program, or a programmer? Why hide things from a programmer?

The meaning of "inheritance", perhaps surprisingly, seems clear enough. When to use it is another matter.

So, according to Booch we have abstraction and encapsulation applied to objects, modularity applied to compilation units, and inheritance. Am I the only person who thinks this is rather wierd?

I guess I have a very basic view of design, as consisting of two main principles: (1) decomposition into smaller, simpler pieces, and (2) finding and exploiting similarities among the pieces.

The first is the most important, and the hardest. The difficulty is that the number and complexity of the links between the pieces may increase as fast as the complexity of the pieces decreases. The concepts of abstraction, encapsulation and

modularity are part of the decomposition process – it is difficult to have one without the other two. Programming languages provide various mechanisms for implementing these concepts for various kinds of pieces, but the concepts are the important bit, not the mechanisms. Describing these concepts as "Object Oriented" may make commercial sense, but it makes no other kind of sense that I can see. Applying them inconsistently to different kinds of pieces seems even more absurd.

Programming languages also provide mechanisms for exploiting similarities among the pieces: templates, aggregation, inheritance and above all, functions, come to mind. I am baffled as to why Booch only mentions the latter in his formulation of OOD.

Grady Booch is a lot more experienced than me, and so are many of you, so I'm sure someone will enlighten me about the true meaning of abstraction, encapsulation and modularity.

What is OOD good for?

Sean said he was struck by the variety of application domains described in "Design Patterns". I was struck by how few were to do with scientific/engineering software, and how many to do with GUIs. I suspect that the current popularity of OOD has a lot to do with the current popularity of GUIs. Has it proved itself in any other area?

Graham Jones

I trust that many of you will have strong opinions on this! My comment about "Design Patterns" was intended to mean that not all the examples were GUI-related, which is sadly so often the case. My own OO experience is such that I've never used it for GUI projects but I've found it very useful in many other application areas – Ed.

Explorations around a linked list

by Peter Moffatt

Editor's note: I thought long and hard about including this article. At first I wanted to reject the article, then I found myself inserting editorial comments everywhere. Finally, I decided to publish it pretty much as supplied without comment. What convinced me to do this is that Peter's article highlights several very common misconceptions and mistakes made by those new to C++. The "gotchas" which Peter trips over are: conversions, template specialisations, arrays vs. pointers and the more general one that what compilers allow or disallow isn't necessarily right or wrong respectively. I hope you will all think very carefully about each of the decisions Peter made in arriving at his solutions and that you will examine each of his conclusions in depth. I also hope Peter will forgive me for holding his article up as a good example of a reminder to the "experts" of just how much they take for granted – Ed.

I have recently been exploring various aspects of C++ while working on a linked list program, using first Turbo C++ 2nd Edition, then Borland C++ 4.02, and I thought that some of the things I learned along the way might be of interest to others. I wanted to produce a base list class to handle lists of records or structures ordered on a member field, so that the base class handled as many as possible of the basic list functions of inserting, removing and finding items, and displaying the complete list, while details of the actual list items and the sort key field were provided in a derived class for each individual list. It was a fundamental objective that the base list class should remain independent of the content of any particular list, so that it could be used unchanged by different applications. I decided not to include the data items directly in the list, but to use a list of "links" – structures containing pointers to a data item, its key field, and the next link. The following is a skeleton of the classes and data members used:

```
class item;
class list
 {
 struct link
  {
  item* data;
  void* key;
  link* next;
  ....
  };
 link *headpointer, *current, *entry;
 ....
 };

class item
 {
 int number;
 char name[11];
 ....
 };

class derivedlist : public list
 {
 item *input, *dummy;
```

```
....
};
```

Traversing the list to add a link/item in order involves a series of comparisons between an "entry" link/item, and successive "current" link/items already in the list, where entry and current are pointers-to-link. Much of the development consisted of the evolution of base and derived class comparison functions and various ways of dealing with different key types. This was done with a series of partial versions of the program which accept new items and compare each with the previous one, without maintaining the list.

As *entry* and *current* are pointers, attempting to overload the comparison operators to use **if (***entry* **<** *current***)** to compare items will not work – (it compares pointers using the standard operators). Operators could be defined so that **if (****entry* **<** ******current***)** would do what was intended, but I decided it was clearer to use a comparison function *lessthan***(***link****,** *link****)**.

This function must call further functions to compare the key fields. How far is it possible for these functions to be members of the base class? If the key is of a type which can be compared by the standard operators, the base class only needs to "know" where the key is located in the data item. If it were made a requirement that the key value be held in a data field called "keyfield", then it could be accessed through the pointer-to-item member of a link object by

```
if (entry->data->keyfield <
            current->data->keyfield)
```

If the key is required to be accessed by the pointer-to-key field in a link object, or if the key cannot be compared by the standard operators (a character string, for example) then the comparison functions can only be base class members if they "know" the type of the key. A means of passing this type information is provided by templates, but before I had a compiler which implemented them I looked at other ways round the problem.

I assume that string keys in a defined key field could be handled by defining a *String* class with overloaded comparison operators, but I did not explore this. Otherwise, it became apparent that to handle all key types (including user-defined types) the comparison functions must be derived class members called from the base class. A base class function can only call a derived class function if the latter is a redefinition of a virtual function declared in the base class. One version of my list therefore relies on virtual functions to solve the key type problem.

The body of function *list***::***lessthan***(***link****,** *link****)** is

```
{
 return less(entry->key, current->key)
        ? 1 : 0;
}
```

and class *list* contains the pure virtual member function

```
    virtual int less(void*, void*) = 0;
```

Any derived class must provide a function **int** *less***(void*, void*)** the body of which will compare the item fields pointed to by *entry->key* and *current->key*, after first casting the **void** pointer to the correct type.

Thus for a key of type **int**:

```
    int less(void* entrykey,
             void* currentkey)
    { return *(int*)entrykey <
             *(int*)currentkey ? 1 : 0;
}
```

or for a key of type char[]:

```
    ....
    { return strcmp((char*)entrykey,
                    (char*)currentkey) <
0
             ? 1 : 0; }
```

Virtual functions can also be used to print or display list data, with a base class pure virtual function

```
    virtual void print_item (item*) = 0;
```

redefined as required in each derived class and called by

```
    print_item (current->data);
```

I would welcome comments on the soundness or otherwise of the virtual functions method, given the general intentions I have outlined. Listing COMPARE1.CPP is the test program for the comparison functions using this method, and I have also used the same method in a full linked list program LNKLIST1.

All the code from Peter's article will be supplied on a future CVu disk and made available on the FTP site – Ed.

The test programs simply accept new items and compare each with the previous one, without maintaining a list. A dummy current link/item is

set up before the first actual item entry. Each item has a number and a name field, the latter being the key in these examples. Items are created in a general function *derivedlist*::*run*(), and a pointer or reference to each item, and to its key field, are passed as the arguments to the *list*::*insert* function.

In the test programs this simply sets up an entry link/item, and *derivedlist*::*compare* then calls the functions I have been discussing to compare each item with the previous one and report the result. The current link/item is deleted, and *entry* becomes *current* ready for the next comparison.

Using the same data items ordered on number rather than name requires the redefinition of function *derivedlist*::*less* for type **int**, and calling *list*::*insert* with argument *input->name* replaced by &*input->number*. Using different data items obviously requires the redefinition of class item.

When I changed to a compiler which implemented templates, I went on to develop a template version, again using a limited-function program (COMPARE2.CPP) to test the comparison functions.

The principle seemed to be to make the base class a template class, declared

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

with its argument the type of the item key. The link data member pointer-to-key would be declared as *T** key, the key comparison function parameterised as **int** *less* **(***T****, *T****)** and the class instantiated with the required key type for a particular list. At first I confused the instantiation of a class from a class template with the instantiation of a class object from a class definition, and did not see how the former was achieved in an inheritance situation where no base class object is instantiated. In fact, instantiation of the base class type is achieved by using its name with the required actual type parameter in the class derivation list:

```
    class numberlist : public list <int>
```

or

```
   class addresslist: public list
<char[]>
```

I went through numerous minor versions with *link*::*data* and *link*::*key* declared as pointers or references, *item*::*name* declared as **char*** *name* or **char** *name***[]**, with corresponding different methods of setting up a new item in an input function, and the class *list* template argument for a string key declared as **<char>**, **<char*>** or **<char[]>**. The most logical and consistent combination seemed to be to use

```
    item* data;
    T& key;
    link* next;   // in struct link,

    char name[11] //  for a character
key
                  // in item, and
    <char[]>      // for the template
                  // argument for this
key.
```

The base class template function **int** *less* **(***T***&, *T***&)**, compares keys of all types with comparison operators, without the need for virtual or other functions to be provided by the derived class. The comparison function looks quite normal if defined inline within the class definition, but if defined outside it must take the form

```
template <class T>
int list<T>::less (T& entrykey,
                   T& currentkey)
{ return entrykey < currentkey ? 1 : 0;
}
```

Even member functions which make no reference to the template type must be defined in this way, presumably to indicate which class they belong to. I wonder if template syntax couldn't have been made less cumbersome?

Whilst you could argue that list<T> == list in this case, it would be an exception to the many more general cases and would make the language harder to learn and teach. Now that we have member templates, omitting <T> gives a completely different meaning:

```
template<class T>
int list::less(const T& entryKey,
               const T&
currentKey)
{ ... }
```

Here, less is a member template function of a ***non****-template class called list and would have to have been declared:*

```
class list
{
       template<class T>
       int less(const T&, const
T&);
};
```

I hope this convinces you – Ed.

Once again special handling is required for character string keys. The problem is addressed by providing a "specialised" function, also in the

base class. Because the *less* function defined above will not work for a string key, a specialised function with **char[]** arguments is provided, using *strcmp***()**.

When the instantiation of class *list* **<char[]>** generates the function:

```
int less (char(&entrykey)[],
          char(&currentkey)[])
```

from the template but a specialised function with the same signature is also defined, the specialised function is called. Because this is not a template function but must be defined as a member of the **<char[]>** instance of class *list*, its definition is:

```
int
list<char[]>::less(char(&entrykey)[],

char(&currentkey)[])
{ return strcmp (entrykey, currentkey)
== 0
          ? 1 : 0; }
```

(I took some time to work out the declaration of a reference-to-**char[]** used here. I suppose it is consistent with **char** *name***[***10***]**, but why isn't that **char[***10***]** *name* anyway?) I thought this specialised function would need its own declaration and tried to provide it, getting "multiple declaration" errors until I realised that a single declaration:

```
    int less (T&, T&);
```

in the template class definition declares both the template and the specialised functions.

Providing a specialised function in the base class only provides for a particular key type – in this case a character string. We also need to be able to handle other types (perhaps user-defined), without comparison operators, which might be used as keys by individual lists, without affecting the independence of the base class.

We can deal with such types by combining the template facility with virtual functions. If we declare the template comparison function:

```
    virtual int less (T&, T&)
```

and if the template argument is <*newtype*> and *derivedlist* contains a function:

```
int derivedlist::less(newtype&,
newtype&);
```

then the redefined virtual function will be called in preference to both the template function and any specialised function with the same signature defined in the base class.

To summarise: keys of any type for which comparison operators are defined will be compared by the template comparison function; keys of particular types without comparison operators but known when the base class is written can be compared by a specialised function which overrides the template function; and keys of any other type can be compared by a virtual function provided by a derived class, which overrides both template and specialised functions.

Thinking around the various methods I had used, including the use of references instead of pointers, I wondered if it shouldn't be possible to make more use of overloaded operators to simplify the comparisons. I had found they wouldn't work for *link* objects because they were necessarily accessed by pointers. However, the relationship between each link and its associated data item is fixed, and the data item could be accessed by reference.

If *link***::***data* is type *item***&** then *entry->data* and *current->data* are references to data items. If comparison operators could be defined for class *item* then the comparison **if (***entry->data* < *current->data***)** should be possible. If the operator was a class member function its definition could be written to use the standard operators or *strcmp* according to the key type for the item being defined. In object-oriented terms, an item object would "know" how to compare itself with another. It became clear that neither derived classes, virtual functions or templates were needed, and there was no storage overhead as only one copy of a member function is held in memory however many objects of the class are instantiated. The resulting partial program (COMPARE3.CPP) is certainly shorter and simpler than the other versions I have described. (Though this dramatic simplification is less apparent in the full list program LNKLIST3, where a derived class is used mainly for input/output and user control of list operations). A visibility problem arose with LNKLIST3 which concerns the independence of class list. Calling the class *item* overloaded comparison operators from inline functions in the class list definition gave "illegal structure operation" errors because class *item* had not yet been defined. In a single-file version it was simple enough to rearrange the class definitions, but in a multi-file or library situation this need for the operators to be visible, and also the need for *list***::***remove* to know the size of an item in order to delete it, seems to re-

quire that the complete class item definition be written as a header file and **#include**d.

I would welcome opinions on the relative merits of the virtual functions, template, and "object-oriented" approaches I have outlined.

I would also like to highlight a few specific problems in the hope that others may be able to offer explanations or solutions where I have not found them. Several difficulties seemed to relate to the use of a nested class within a template class, and others to function argument matching.

Two very similar problems occurred when

1. Defining a nested class member function outside its class definition.
2. Defining a template class member function with a pointer to a nested class as its return type.

I tried to define a destructor for *list***::***link* as

```
template <class T>
list<T>::link::~link()
{ delete data; }
```

but found that it would only compile as a member of a specific instance of the template class, as

```
list <char[]>::link::~link() {...}
```

Similarly, in the template full list program LNKLIST2, a list member function *lookup*, returning a pointer-to-*link*, would only compile as:

```
list<char[]>::link*
  list<char[]>::lookup(link* entry,
                       link* current)
{...}
```

It seemed that I was trying to do something that was not allowed, and I confirmed this by a reference in Lippmann, C++ Primer, 2nd edn., p.376 -

> *"Whenever a nested type...is referenced outside the scope of the enclosing template class, the template name must be qualified with the full parameter list."*

The real problem here is that having to specify the actual template parameters in these cases defeats the objective of making class *list* independent of the item and key details of individual lists. It may be that class *list* should be redesigned without a nested class, but I managed to find other solutions.

I think Peter encountered a compiler bug that prevented his code compiling and then was confused by Lippmann's comment into changing his code to something that happened to work with his compiler! – Ed.

The *link* destructor was defined inline within the struct definition, where it would have been in the first place, except that in the non-template COMPARE1 this would not compile because class *item* had not yet been defined and the size of data was not known. I believe that inline definition works in the template class, although the order of definitions remains the same, because the template functions are not actually set up until the class is instantiated – after the definition of class *item*.

The *lookup* function was changed to pass back the pointer-to-*link* as a reference-to-pointer argument (which is within the scope of class *list*) rather than as the function return value, which is not. The definition header thus became:

```
template <class T>
void list<T>::lookup(link* entry,
                     link* current,
                     link*& found)
{...}
```

A similar difficulty arose when defining a nested class outside the scope of a template class.

In COMPARE1 class item is declared within class list and subsequently defined at file scope as:

```
class list::item {....};.
```

I think I would change this now, but I have left it to illustrate this point. When I changed class *list* to a template class I could not find a corresponding template definition which would compile, and declared and defined class *item* independently.

Is the constructor of a class nested in a base class accessible in a derived class?

Struct *link* is declared and defined within class *list*. In COMPARE1, *derivedlist***::***run* sets up a dummy node for the first comparison with

```
dummy = new item (0, "");
current = new link (dummy, dummy->name);
```

which creates a *link* object and calls the *link* constructor *link***(***item****, void*)** to initialise the pointers *current->data* and *current->key*. In COMPARE2, with *list* a template class, the same use of **new** with the *link* constructor gives a compiler message "No match for *link***(***item****, char*)**". In this case the *link* constructor is *link***(***item****,** *T***&)**, where *T* is the key type supplied by the template argument, in this case **char[]**.

The cause of the error seems to be the reference key rather than the template class. I got round the problem by using a *list* protected member function with the same arguments as an intermediate stage:

```
void set (item* data, T& key)
{ current = new link (data, key); }
```

called by:

```
set (dummy, dummy->name);
```

I do not understand, and would be glad if someone could explain:

a. Why the actual arguments **(***dummy***,** *dummy*->*name***)** are interpreted as **(***item****, char*)**, when *name* is defined as **char[***11***]**;

b. Why these actual arguments in a call from a derived class function *match***(***item****,** *T***&)** as the formal arguments of a standard base class member function (*set*), but do not match the same arguments of the *link* constructor when operator new is used in the derived class function.

A related problem arose in the template version of the list program.

The base class *list* contained a protected member function **void** *find***(***T***&)** which was called from the derived class function *addresslist***::***run***()** by *find***(***input*->*name***)**, and in turn called a private base class function **void** *find***(***link****)**.

The problem occurred as a compiler message "No match for *find***(char*)**..." at the first call.

Now to my mind what ought to happen is that *input*->*name*, having been defined as **char** *name***[***11***]** should match the *T***&** argument to the first *find* function, instantiated as **char(&***key***)[]** – that is, the name of a character array should match a function argument of type reference-to-character-array. I also felt that the use of two *find* functions ought to be safe (a) because their different arguments distinguish them under overloading rules, and (b) because one of them is private to the base class.

However, I tried changing one of the names, and this was successful. I had just read somewhere that argument matching and overload resolution takes place before access control, which led me to conclude that what was probably happening was that *find***(***input*->*name***)**, taken as *find***(char*)** was matched with *find***(***link****)** in preference to *find***(char(&)[])**, and that *find***(***link****)** was then found to be private and thus inaccessible from a derived class function. I would welcome confirmation of this or an alternative explanation, and any general observations on handling the types **char[]** and **char***. I wonder if it has anything to do with *item* objects (with **char** *name***[***11***]** as a data member) being created on the free store rather than as local data?

Peter Moffatt

I hope that some of our more expert readers will provide reasoned commentaries on Peter's experiences with C++ – Ed.

Go with the flow

by Richard Percy

Preface

This series of articles tackles the subject of cashflow and related techniques and their implementation in C++. It is aimed squarely at programmers in the finance arena and is intended to show how object-oriented methods and C++ can be used to provide simple, effective, reusable system solutions to some of their day-to-day problems.

Although the subject matter of this series is finance-centred, I hope that the design and coding issues addressed will benefit programmers in other areas. Of course, I would appreciate greatly any feedback concerning these articles to *Overload* or to the mail address below.

Analysis and initial prototypes

Introduction

The requirement for cashflow analyses and projections is almost as old as some of the accountants I know. However, these days business needs seem to prescribe more and more of the kind of financial projection that used to take analysts hours of painstaking, error-prone work before the computer age. Indeed, cashflow projections have come to form a significant part of many financial

computer systems from management information systems to personal pension quotations.

It would seem, therefore, that an efficient, reusable system design for this type of work is in order. For ad-hoc cashflow analysis, the spreadsheet has the area well-covered. Moreover, its widespread use for this work gives a clue to the nature of the cashflow problem: its principles are simple and easy to apply, yet great flexibility is demanded by its consumers.

The sections that follow attempt to define the problem with enough precision to formulate and apply the solutions suggested. The discussion has a bias towards my particular area of expertise: life insurance and pension products.

Requirements

We require a generalised cashflow calculation to be implemented in C++ with the following services:

- Generation of a cashflow of any type from a given start position for a specified number of periods or until a certain condition occurs, if earlier. The position in each period may depend on any other previous position.
- Optional storage of all intermediate positions.
- The option to choose at run-time the function to generate each position.
- The option to convert the cashflow to a different type mid-term.
- Simulation runs from a given start position with/without variation of parameters.
- Model point ("portfolio") runs using the same parameters for each run but different start positions.

The following sections describe some practical applications of the requirements listed above.

Endowment policy quotation

A unit-linked endowment policy quotation for a known sum assured requires the premium to be calculated so that the projected unit value at maturity is equal to the sum assured. This is achieved by simulation runs on the company's pricing basis, each with a different premium.

The policy illustration is then required. This involves calculating the projected unit value at maturity using the basis prescribed by the Personal Investment Authority, which is normally different from the pricing basis.

Only the final projected value is required to be stored. The cashflow in each period depends only on the previous period, so there is no need to store intermediate values.

Stock volatility

The discounted mean term of an investment can be obtained by simple cashflow calculations. The cashflows in all periods are generated then discounted and summed.

An investment decision may rest on calculating the volatility of various stocks at different rates of interest.

Profit testing

A company selling investment products must formulate its pricing basis by calculating the projected profit on what it thinks is the going to be the total portfolio.

The company establishes representative products called "model points" and assumptions (e.g. customer lapse rate). It then simulates the effect on the total profit if these assumptions and/or the charging basis are varied.

A solution

The solution I will outline below will address most of the requirements above. I am leaving out the two requirements for the time being because I think that they will cause complications in the model that will make it difficult to understand at this early stage. These are:

- The position in each period may depend on any other previous position.
- The option to convert the cashflow to a different type mid-term.

These also have limited application (for example, moving averages and investment policy conversions) and the work required to implement them may be onerous. I will address them again as the model is developed.

The model I am developing consists of two main components:

- A *Cashflow* template class that offers services.
- Its clients.

The Cashflow class interface

The interface of the initial Cashflow template class consists briefly of the following:

- A constructor that initialises the cashflow storage and takes as an argument a cashflow "vector" object that it adds to its internal store.

```
Cashflow(Vec* pStartPos);
```

- A function to generate the cashflow for a specified number of periods. The function takes as an argument the address of a member function of the vector class that is required to have certain arguments.

```
// RollFunc type is a pointer to
a
// member function of class Vec
typedef bool (Vec::*RollFunc)
       (const unsigned long,
Vec&);

// generate the entire cashflow
// using duration-limited roll
// forward
void RollUpLim(RollFunc,
      const unsigned long
duration);
```

- Functions to print the cashflow vectors to a stream.

```
virtual ostream& PrintOn(ostream&
=
                         cout)
const;
template <class Vec>
ostream& operator << (ostream&,
                const
Cashflow<Vec>&);
```

- A destructor that clears all resources allocated by the cashflow object.

```
virtual ~Cashflow();
```

The client class interface

The *Cashflow* class requires its clients to have the following functions implemented:

- A default constructor and copy constructor.

```
TestVec(double U=0, double G=0,
        double Q=0, double S=0,
        double P=0, double M=0)
: uv(U), g(G), qx(Q),
  sa(S), p(P), md(M) {}
// default copy, assign and
// destructor are OK
```

- Overloaded comparison and stream operators.

```
bool operator == (const TestVec&,
                  const
TestVec&);
ostream& operator << (ostream&,
                    const
TestVec&);
```

- At least one cashflow projection function.

```
bool ProjectionRF(
    const unsigned long
newDuration,
    const TestVec& oldRow);
```

Implementation of the *Cashflow* class

The implementation shown here compiles with Borland C++ version 4.02.

I must apologise in advance for my heavy use of the Borland-specific classes to provide array containers, strings and exception-wrapping. I had intended to start off with the Standard Template Library and, indeed, I downloaded a tailored version for version 4.5 from a Web site to this end. However, this did not work with my version and so I have elected to use what I already have for this initial prototype.

Storage of the cashflow is managed by a Borland template-based array as a private data member:

```
typedef TIArrayAsVector<Vec> CfArray;
CfArray huge* pcf;
```

The array is initialised in the constructor (which also adds the start position)...

```
template <class Vec>
Cashflow<Vec>::Cashflow(Vec* p)
: pcf(new CfArray(cfInitUpper, 0,
                  cfGrowth))
{
  pcf-
>OwnsElements(TShouldDelete::Delete);
  // array "owns" elements
  pcf->Add(p);
}
```

...and trashed in the destructor:

```
template <class Vec>
Cashflow<Vec>::~Cashflow()
{
  pcf->Flush(); // delete all elements
of
                // underlying array and
                // free all memory
  delete pcf;
}
```

This model only contains the cashflow generation process and a routine to print the entire cashflow. The generation is very simple, in spite of the forbidding syntax. The duration-limited-

rollup function works with a simple loop, each iteration of which creates a new vector, adds it to the internal array and calls a member function of the client class to fill in the values.

```
template <class Vec>
void
Cashflow<Vec>::RollUpLim(RollFunc
pfRollUp,
                    const unsigned long
dur)
{
  if (0 == pfRollUp)
    throw xmsg(string(
                "Null pointer passed"));

  bool cont;
  unsigned long c = 0;
  Vec* pNewRow;

  do
  {
    pNewRow = new Vec();
    pcf->Add(pNewRow);
    cont = (pNewRow->*pfRollUp)
           (c,

*(*pcf)[static_cast<int>(c)]);
    c++;
  }
  while (cont && c < dur);
}
```

Printing is achieved by creating an iterator and using it to call the client class' overloaded << operator.

```
typedef TIArrayAsVectorIterator<Vec>

CfIterator;
template <class Vec>
ostream& Cashflow<Vec>::PrintOn(
                        ostream& o)
const
{
  CfIterator iter(*pcf);
  while (iter)
  {
    o << *iter++;
  }
  return o;
}
```

The prototype seems to work well and is reasonably efficient and simple to use. I have hidden the underlying implementation of the storage as far as possible so that I can substitute an STL container at a later date. The only external evidence of the array implementation is the fact that the client class must provide an overloaded == operator, which is required by the Borland container.

The model now requires further interface functions to fulfil the requirements listed at the start. I will look at providing these in the next article. We will also require some "helper" classes to provide generalised formatting for the numbers and targeting functionality. I will also address these in this series.

Implementation of the client class

The implementation of the required functions of the client class needs little explanation. I have provided an example that shows a projection calculation for a unit-linked endowment policy. This is an accurate representation of the way such calculations are done in practice but the details are greatly simplified. Most functions are trivially implemented but the rollup routine requires a few comments.

The unit value in each month is calculated by taking the unit value in the previous month, adding the premium and subtracting charges, then inflating the result at an assumed interest rate to simulate the effect of unit price growth.

The only charge is for mortality, which is obtained by applying a crude probability of death during the month to the difference between the sum assured and the unit value (the "sum at risk").

```
bool TestVec::ProjectionRF(
           const unsigned long
newDuration,
           const TestVec& oldRow)
{
  g = oldRow.g;
  qx = exp( newDuration/100.0 ) /
50000.0;
  sa = oldRow.sa;
  p = oldRow.p;
  md = max(qx * (sa - p -
            max(oldRow.uv, 0.0)),0.0);
  double um = oldRow.uv + p - md;
  uv = um * (1 + g);
  return uv > 0 ? true : false;
}
```

Performing the calculations

The entire calculation is performed by creating a start position, creating a *Cashflow* object then running the projection. The start vector must be allocated dynamically and the *Cashflow* class takes care of deallocation. Note that if an exception is thrown back to *main***()** then all dynamically allocated objects are cleared up by the destructor of the *Cashflow* class.

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

```
                25*12);
    cout << "...finished roll forward!"
         << endl;
    cout << t << endl;
    retCode = 0;
  }
  catch (xmsg x)
  {
    cout << "\nException!\n\n"
         << x.why() << endl;
    retCode = 32767;
  }
  catch (...)
  {
    cout << "\nException!\n\n"
     "Program threw an unhandled
exception"
         << endl;
    retCode = 32767;
  }
  return retCode;
}
```

If you run this example you will note that the calculations are performed in the blink of an eye (or quicker!) but the printing is rather slow. This is a limitation of the console, not of the *Cashflow* code.

The final projected value for this policy, which has a sum assured of £50,000, premium of £50 per month and term of 25 years, is nearly £60,000.

Richard Percy
106041.3073@compuserve.com

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

by Sean A. Corfield

Introduction

In previous columns I've looked at representation of information within a preprocessor and a parser but I haven't said much about parsing itself. I maintain that parsers aren't really that difficult so this issue might be light relief for some of you who felt last issue's multiple inheritance scenario was a bit of a nightmare!

In the draft C++ standard, the clause describing statements is just about the shortest clause in the document. Statements are simple. Discuss.

Parsing considerations

Apart from expression statements and declaration statements, C++ begins each statement with a unique keyword: **if**, **while**, **do**, **switch**, **for**, **goto**, **return**, **break**, **continue**. This makes parsing quite simple because we can determine what is expected to come next based on a single token lookahead:

```
Token token = lexer.get();
switch (token)
{
case IF:     return ifStmt(lexer);
case SWITCH: return switchStmt(lexer);
case OPEN_BRACE:
             return compoundStmt(lexer);
//...
default:
  // must be a declaration or expression
  return declOrExpr(token, lexer);
}
```

Before you all howl in protest at the multiple returns, let me just say that I wouldn't write it like that – I'm just saving space on the page!

What am I assuming here? This is the body of a general *stmt* function that takes a lexer as an argument (well, actually a phase six token stream) and returns a statement, or rather a *Statement******. The functions called above are similar and each reads from the lexer, constructs a statement and returns a pointer to it. Let's take a look at one of those functions in more detail.

ifStmt is called after we have seen the keyword **if** so the next token expected is **(**. After that we expect an expression, a **)**, and a statement. This is optionally followed by **else** and another statement. It will look something like this:

```
Statement* ifStmt(TokenStream& lexer)
{
  Statement* statement = 0;
  Token token = lexer.get();
  if (token == OPEN_PAREN)
  {
    Expression* condition =

expression(lexer);
    if (condition) // parse succeeded
    {
      if (lexer.get() == CLOSE_PAREN)
```

```
      {
        Statement* trueStmt =
stmt(lexer);
        if (trueStmt)
        {
          Statement* falseStmt = 0;
          if (lexer.lookahead() == ELSE)
          {
            lexer.get(); // skip else
            falseStmt = stmt(lexer);
          }
          statement = new
IfStmt(condition,

trueStmt,

falseStmt);
        }
      }
    }
  }
  return statement;
}
```

For simplicity I've omitted error reporting. As an exercise, you might like to consider different ways to implement robust error reporting and recovery.

As you can see, this calls the general statement parsing function recursively to handle substatements. This top-down parsing approach is called *recursive descent parsing* and is a common and simple technique that allows parsers to be written for complex languages.

Other parsing techniques

Another approach for parsers is to use *yacc* (Yet Another Compiler Compiler) that takes a grammar description and produces a parser that is a complicated state machine. Instead of the fairly readable code above, a state machine parser uses a switch statement to determine what to do with any given token based on the current "state". A state machine to parse C code has several hundred states. For C++, the grammar is too complicated and ambiguous for a mechanical translator like *yacc* and some fairly "clever" tricks need to be performed by the lookahead to seed the token stream with "hint" tokens. Cfront is based on such an approach.

What to do with statements

Having built our statement, what would we like to do with it? If we're going to do source code analysis (my original project, you may recall), we would want a method to check the semantics of statements and their subcomponents. For an **if** statement, that might look like this:

```
void IfStmt::semantics()
{
  condition->semantics();
  trueStmt->semantics();
  if (falseStmt)
    falseStmt->semantics();
  if (condition->type() == ASSIGNMENT)
    warning("Possible = instead of
==?");
}
```

Perhaps, instead, we wish to generate assembly language from it – compile it. A *compile* method might look like this:

```
void IfStmt::compile()
{
  condition->compile();
  Label* label = getLabel();
  branchIfZero(label);
  trueStmt->compile();
  if (falseStmt)
  {
    Label* end = getLabel();
    branch(end);
    label->write();
    falseStmt->compile();
    end->write();
  }
  else
  {
    label->write();
  }
}
```

The same recursive approach that we used in the parser appears naturally in the methods that operate on the statements produced by the parser.

Separation for reuse

You might be asking why doesn't the parser perform the desired operations as it builds each statement? I hope you can answer that question yourself with a little thought – separating the operations from the parser means that the parser can be reused for any other tools we may want to build that accept the same source language. However, as written, the parser is still quite closely linked to the actual operations because it has to know about the types of the statements. In order to produce a truly generic parser we need to somehow abstract out the specific types and their operations. Since the parser creates the objects directly, the solution is slightly harder to find than it might first appear.

A common approach for providing different behaviours for a common type is to use inheritance with the varying behaviour implemented in different derived classes. However, this means that the specific derived class objects must be created directly by the parser. That's precisely what we are trying to avoid.

Offhand, I can think of two solutions, I'm sure there are others. The first solution came to mind because of my fondness for templates. The sec-

ond solution came about as I tried to make the first solution more elegant.

A parser template

For each different application of our parser, we end up with a separate method in each derived statement class that performs a specific operation (code generation, source code analysis etc). If the parsing functions were templates, the operation could be a template parameter somehow, which could be used to create the appropriate derived class instance, e.g.,

```
template<class Operation>
Stmt<Operation>* ifStmt(PhaseSix& lexer)
{
  Stmt<Operation>* statement = 0;
  Token token = lexer.get();
  if (token == OPEN_PAREN)
  {
    Expr<Operation>* condition =

expression<Operation>(lexer);
    if (condition) // parse succeeded
    {
      if (lexer.get() == CLOSE_PAREN)
      {
        Stmt<Operation>* trueStmt =

stmt<Operation>(lexer);
        if (trueStmt)
        {
          Stmt<Operation>* falseStmt =
0;
          if (lexer.lookahead() == ELSE)
          {
            lexer.get(); // skip else
            falseStmt =

stmt<Operation>(lexer);
          }
          statement = new

IfStmt<Operation>(condition,

trueStmt,

falseStmt);
        }
      }
    }
  }
  return statement;
}
```

Note that the *Operation* template parameter is not used in the signature of the parsing functions – we rely on explicit qualification of the template function calls. Unfortunately, this is a relatively new feature and not widely supported. Below I'll show how to work around this.

What is the *Operation* class and how does it solve our problem? Since we are trying to add a method to our framework of statement classes, *Operation* can be used to provide a base class for *Stmt* as follows:

```
template<class Operation>
class Stmt : public Operation
{
//...
};
template<class Operation>
class IfStmt : public Stmt<Operation>
{
//...
};
class Compiler
{
public:
  virtual void compile() = 0;
};
class Analyser
{
public:
  virtual void semantics() = 0;
};
// use these to read and compile a
// statement:
stmt<Compiler>(lexer)->compile();
// and to read and analyse a statement:
stmt<Analyser>(lexer)->semantics();
```

This technique of deriving from a template parameter allows us to introduce methods and other information *above* an existing class and can be very powerful, e.g., when a class is written that depends on, as yet unknown, abstractions or functionality.

To provide the actual method in the derived classes, we can specialise it:

```
template<> // recent syntax for
           // specialisation
void IfStmt<Compiler>::compile()
{
// as above
}
template<>
void IfStmt<Analyser>::semantics()
{
// as above
}
```

Unfortunately, we need to declare the method in each of the derived classes that we intend to specialise it for which means we also have to specialise those derived classes:

```
template<>
class IfStmt<Compiler>
: public Stmt<Compiler>
{
public:
  // other methods
  void compile();
};
```

This isn't really very elegant but at least the parser is now generic.

A parser factory

This mixture of compile-time polymorphism (templates) and run-time polymorphism doesn't work too well in this instance. Our problem lies in having to select the operation (compile, analyse) at compile-time. If we have a hierarchy of statements that "compile" and another hierarchy of statements that "analyse", can we choose at run-time which one we want? Yes, we can use a statement factory to create the appropriate objects for us.

```
class StatementFactory
{
public:
  virtual ~StatementFactory() { }
  virtual Statement*
          newIfStmt( Expression*,
                     Statement*,
                     Statement* ) = 0;
  //...
};
class CompilerStatementFactory
 : public StatementFactory
{
public:
  CompilerStatementFactory() { }
private:
  Statement* newIfStmt( Expression* e,
                        Statement* t,
                        Statement* f )
  { return CompilerIfStmt( e, t, f ); }
  //...
};
```

The parser would then be constructed with the appropriate factory object and would use

```
statement = myFactory->newIfStmt( a,b,c
);
```

instead of

```
statement = new IfStmt<Operation>( a,b,c
);
```

(or however we created the specific types of statement objects). Note that the abstract base class has a virtual destructor but no constructors – the default will be sufficient (we could declare a **protected** default constructor if we really wanted to be more precise). Furthermore, the derived factory classes have only a **public** constructor and then all the methods are **private** – why? Because they are only ever called through the **public** methods in the base class – they are "merely" an implementation detail.

More genericity, please waiter!

Some people are never satisfied! We now have a parser that can be told, at run-time, to build either a compiler representation or an analyser representation. Can we actually make it more generic? How about if we could build a generic representation and then tell that what to do?

If we consider what operations we need to perform, we see that we can use a trick very much like that for the parser factory to defer the operation until run-time. This time we use a generic *operation* method within each statement class that delegates to a factory-like object method for each derived statement class's *operation* method:

```
void IfStmt::operation(
  const Operation& anOperation)
{
  anOperation->ifStmt(condition,
                      trueStmt, falseStmt
);
}
class Operation
{
public:
  virtal void ifStmt( Expression*,
                      Statement*,
                      Statement* ) = 0;
  //...
};
class CompilerOperation : public
Operation
{
private:
  void ifStmt( Expression*,
               Statement*, Statement* );
};
void CompilerOperation::ifStmt(
  Expression* condition,
  Statement*  trueStmt,
  Statement*  falseStmt
)
{
  condition->operation( *this );
  Label* label = getLabel();
  branchIfZero(label);
  trueStmt->operation( *this );
  if (falseStmt)
  {
    Label* end = getLabel();
    branch(end);
    label->write();
    falseStmt->operation( *this );
    end->write();
  }
  else
  {
    label->write();
  }
}
```

Static vs. dynamic

I hope this instalment has shown how sometimes a dynamic (run-time) solution can be more elegant than a static (compile-time) one. I hope you've also seen a pattern in the solutions above, where a (deep) class hierarchy can be mapped onto the methods in a (shallow) factory hierarchy

and the resulting combination provides an elegant double-despatch, polymorphic in both the type of the "factory" and, in our case, type of the statement.

Next time

In the dark prehistory of this irregularly scheduled column, I threatened articles on templates, overloading and a myriad other things. For the next couple of issues I'm going to take a break and then go back and critically review some of the material in the first six articles – now is a good time to send me comments and / or questions on what you've read so far. It's also a good time to make requests about what you'd like to see covered in future articles in this column.

Sean A. Corfield
Object Consultancy Services
sean@ocsltd.com

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!

In this issue I report on the most recent C++ committee meeting and Francis takes a closer look at one of the changes made at that meeting.

The Casting Vote
by Sean A Corfield

Poised on the brink...

As I have explained in previous columns, the process we follow in standardising C++ means that once we reach a certain stage, we can no longer make "large" changes. Stockholm, July '96 was that stage and we resolved to make no further changes unless the National Body comments require us to do so.

This left us in a somewhat difficult position since we had some fairly large unresolved issues on the table. Fortunately, as in so many projects, the impending deadline spurred us on and we made great progress, finally reaching consensus on some long-standing issues.

Lists of lists

Over the last few years, the number of large issues has dwindled and we have quietly got on with solving the smaller problems. Each part of the language and each part of the library has provided a steady stream of minor things to deal with. Each part has had a nominated member of the committee as list-keeper and they have worked with subgroups to produce workable resolutions that the committee as a whole should adopt. For example, in Core III (formerly Extensions) WG, John Spicer of Edison Design Group has handled the template issues, Bill Gibbons (ex-Taligent, now HP) has handled the namespace issues and pointer to member issues and Dag Brück (Dynasim, Sweden) has handled the exception handling issues. Like the other WGs, we worked through each list, discussing and generally approving the suggested resolutions. Each list went forward to the full committee for approval of the resolutions and thence into the Working Paper. We cleared all of the exception and namespace issues and all but two very minor template issues.

The various Library WGs had the longest lists to process and they managed to clear nearly everything (literally hundreds of issues) and that got them a round of applause from the full committee!

This doesn't mean we're finished, just that we can now concentrate on the even smaller issues that make standards such a thrilling business (if you like that sort of thing).

Where should I put my templates? Revisited

An ongoing theme of this column is templates and in particular the source model required for portability. You may remember that in my previous *Casting Vote* column, I said that X3J16 voted in Santa Cruz to remove separate compilation, pending a further vote in Stockholm. Much has happened since! Silicon Graphics (SGI) worked very hard to produce a solution that would be acceptable to enough people that we could vote to keep separate compilation. Their solution involved several changes that restricted templates and made them more intuitive regardless of the source model.

Ultimately we still needed one key piece to solve the puzzle: how to determine whether a template definition should be available outside its transla-

tion unit. SGI's proposal originally suggested that declaring a template **extern** should have the desired effect. Overloading **extern** in this way was not terribly popular with the WG so I suggested **export**. After some further discussion, this was accepted.

This means that existing template code that uses the source inclusion model will continue to work pretty much unchanged. Code that is intended to work when template definitions are compiled separately must be modified to declare the definitions with **export**. Since no two compilers handle separate compilation of templates in the same way, this shouldn't be too much of a problem – such code isn't portable at the moment anyway.

```
// export tells the compiler that this
// definition might be referenced from
// another translation unit so it must
// squirrel the definition away
somewhere:
export template<typename T>
void soSomething(T t) { ... }

// this template is not exported so the
// compiler can assume that it will be
// defined identically in every
translation
// unit that references it - it need
only
// perform any instantiations found in
this
// translation unit:
template<typename T>
void doSomethingElse(T t) { ... }
```

Stringing us along

Some years ago I proposed that string literals be made **const**. The proposal was quietly brushed to one side and I let it lie. The UK panel remained unhappy about the issue and recently Kevlin Henney produced a new proposal to achieve the same goal. Although our proposals were largely identical, so much has changed both within the language and within the committee mood that Kevlin's proposal was accepted. See Francis' article below for more details on this.

Out! Out! Damned name injection!

Yes, we finally got rid of nasty old name injection. Again, this has been mentioned in several of my past columns and various attempts have been made to remove it in the past. Bill Gibbons finally came up with a proposal that solved enough of the problems to gain support from the majority of the committee. Quite simply, if a call to a function *f* involves a type *T*, friends of *T* (declared in *T* or its base classes) are considered in the lookup of the name *f*. This mirrors, to some extent, the operator lookup rules adopted recently – the so-called "Koenig lookup". This lookup has now been uniformly adopted for all operator and function calls, both inside and outside templates.

Note that this is a fairly major change, affecting far more than just friend name lookup: it means that the language now has one well-defined process for all name lookup, regardless of templates, that has the appropriately intuitive behaviour in the presence of namespaces, i.e., whenever an object whose type is from a namespace is used in an expression, all "related" functions and operators are "automagically" considered. This should make namespaces much easier to use.

Inching closer

The amount of consensus in Stockholm means that we should now release the second Committee Draft after the Hawaii meeting, triggering a second ANSI Public Review, and hopefully moving on to a Draft International Standard in the middle of 1997. At that point, the draft becomes something that can be referred to with authority because the remainder of the standards process – bar minor typos – is a rubber-stamping exercise as far as the majority of working programmers is concerned.

Sean A. Corfield
Object Consultancy Services
sean@ocsltd.com

Making string literals constant – a cautionary tale

by Francis Glassborow

In the dim distant past when K&R were dreaming up C the possibility of write-protected RAM had yet to surface from the primordial chaos. The C language as originally designed had no `const` keyword. It wasn't until almost a decade later that C++ had to invent the word because its use of reference parameters for large objects required it. C grabbed the concept as being helpful for some aspects of optimisation, added `volatile` (which is a kind of anti-optimisation qualifier) and placed both in its proposed standard. For a language with such an anti-new-keyword mindset this introduction of two new keywords must have been quite traumatic.

Once `const` had been introduced to the language it became possible for compilers to place

suitable static (Computer Science sense) data in write protected memory. Indeed, the `const` qualification of static data was useful when writing for embedded systems as it indicated that the data could be stored in ROM rather than the very precious RAM.

Unfortunately there was one place in the language where the qualification would have been natural but would make manifest that reams of existing code was already broken. This was the area of string literals. These are clearly conceptually constant, and good optimisers might well economise on string literal space by overlaying one string on another. Ideally programmers should make no assumptions as to how string literals are implemented. In practice a whole generation of programmers have made assumptions and many have written code that writes to the notional storage for a string literal. I know that that provides undefined behaviour, but if it works why change it?

More to the point is that many 'correct' programs include lines such as:

```
char * message = "Invalid input.";
```

If we make a string literal an array of `const char` the above line is broken. As there was no overwhelming reason for C to change string literals from array of `char` to array of `const char` it did nothing. If programmers were stupid enough to write:

```
scanf("%s", "This is a buffer");
```

that was their problem.

C++ is in a different position because it supports function overloading based on the type of the arguments in the call. So:

```
void foo(char *);          //A
void foo(const char *);    //B
main() {
       const char * help = "help";
       foo(help);    // calls B
       foo("help"); // calls A
       return 0;
};
```

This is potentially a nasty surprise, particularly as we are yet to have compilers that tell us which overload choice they have made (note to implementors, many of us would love to have a facility for enquiring about the choice in critical sections of our code, perhaps via a `#pragma` directive).

No, save us from #pragma! Please provide compiler options instead! – Ed.

The obvious step is to make a string literal an array of `const char` in C++. This was resisted on the grounds that it would break existing C code. One design criteria for C++ was to avoid gratuitously breaking C code. In my opinion this design constraint should not have been applied to this case.

Several years ago Sean Corfield produced a paper for WG21/X3J16 proposing that string literals (and wide string literals) should be made into `const` qualified types with special deprecated conversions that would allow them to be treated as unqualified types. Fools who insist on writing to string literals would sometimes get their code compiled (though hopefully with a warning about a deprecated conversion), careless programmers who write lines such as:

```
char * message = "This is careless";
```

will get warnings about using deprecated conversions. Where the issue is a matter of choosing the correct overload the compiler will choose the one the programmer expects.

This paper failed to get to the joint committees because it fell at the vetting stage by the Core working group.

Interlude Ð how the standards committees work

Any proposed substantive (non-editorial) change to the Working Paper (that which will eventually become the Standard) must be supported by a paper that describes the change and the reasons for it. These papers can be very brief, but they must exist. They can be written by anyone but must be funnelled through the C++ specialist group of a National Body such as BSI or directly through ANSI X3J16

These papers are then considered by working groups. Each working group is made up of people who have a particular interest in the parts of the WP it covers. A working group may also identify and generate papers on its own behalf.

When a working group considers a paper they may reject it, return it to the author for reworking, modify it themselves and issue a revised paper during the meeting or simply accept it. In either of the latter two cases it then presents the paper to the joint committees. If there is any sign that there may be disagreement a straw poll is taken to determine the general opinion. Only items receiving substantial support go forward for a formal vote. At this final stage all voting

members of X3J16 who are present are required to vote, they are not allowed to abstain on a vote on a technical issue. Immediately afterwards the ISO WG21 vote is taken. Abstentions are allowed in this vote but if there are more than two negative votes the issue would almost certainly go back for further review.

The early stages are designed to filter out trivial or ill-considered changes. Even carefully considered proposals may have surprising side effects (more about these in the context of string literals in a moment). The latter stages are intended to ensure that work proceeds by consensus. Unfortunately there are a number of holes. Consensus takes time to reach and is not always possible when there are strongly held opposing positions. It assumes that papers will be considered on merit, but a small work group may not represent the attitudes of the majority. The requirement for all ANSI members to vote is based on the assumption that representatives at a Standards Committee will always know enough to have a valid opinion. This last assumption is seriously flawed in the case of C++. The breadth from language design through syntax, environment, C compatibility to a massive library section means that it would be a very rare person who had a firm grasp of the technical implications of a change in more than half the WP.

The most critical stage for any proposal is that of getting through the work group to reach the full committees. It is assumed that those proposing changes will have enough personal interest to be present and to support their proposal. Only the most clear cut changes (such as my proposal to make explicit that `main` returns an `int`) will get through a work group if the proposal's author or another 'champion' isn't present. This works fine for X3J16 members because they can usually find someone to champion a proposal if they cannot do so themselves. It is much harder for the various ISO delegations. These are often only one or two people and they may have papers written by others of their national body which they want to promote. A two person delegation will struggle to cover the ground, even if they are fortunate to have both a library and a language expert.

One thing to keep in mind is that Standards Committees are not there to develop the best possible item. Their purpose is to develop rules to promote commerce. A proposed change to C++ that was technically excellent but that resulted in large scale complaints from users because it broke their code would be likely to fail. Even if existing code was already broken, many commercial implementors might prefer to leave the issue to tool vendors.

Back to string literals

The reason (I believe) that Sean's paper to make string literals array of `const char` failed is that he was not present when the relevant work group considered it. With hindsight he would have made arrangements either to be fetched from the work group he was in when the issue came up, or he would have found someone else to champion it. It takes time to develop relationships so that other people will champion your work and it is in the nature of things that you are best known by members of your own work group.

> *I actually asked the working group chair to include me in the discussion of the proposal but, probably because I was still new to the committee, I was passed over. It doesn't tend to happen to me these days! Ed.*

As the problem of string literals is clearly a bad wart Kevlin Henney independently produced a paper that proposed a solution very close to Sean's. The timing was such that the paper could only be considered in Stockholm. After that it would be considered as too big a change unless a NB made it a stopper issue at CD vote (a very unlikely event—we can live with non-const string literals even if we would prefer not to).

I arranged to be called from my work group (Core I) when Kevlin's paper was considered by Core II. When I arrived I found that most members of the work group were lukewarm. John Bruns of NationsBanc-CRT was very strongly for it but the rest were, on balance, against. Steve Adamczyk of the Edison Design Group was strongly opposed because he believed that there was more code that would be broken than that which Kevlin had identified. I think that given more time the proposal might have been sent back for reconsideration. Here are some of the problems Steve identified.

```
throw ("help")
```

that would currently be caught by:

```
catch (char *)
```

would not be caught if string literals were `const`. Actually, I wonder if cv-qualification

should be stripped from the argument of a `throw` as the alternative would seem to be too error prone.

This is particularly unpleasant because the failure to catch after the change might not be detected till the program failed to handle the exception. The issue here is that programmers should improve their testing of exception catching. The other issue is that you should have some very special reasons for **not** `const` qualifying the type in a catch statement.

IMO, a good compiler could easily warn about any catch that has a pointer or reference to a non-const type: since the thrown object is a copy, such practice is suspect at best. Ed.

Another problem is the following code which is a simplistic example of a fairly common idiom:

```
char * p=0;
int i=0;
p= i ? "non-zero" : "zero";
```

The problem is that, according to the strict semantics, the strings have been converted to pointers before the problem assignment. The return from the conditional operator will be of the most restrictive type resulting from the types of the second and third operands. In this case it will be a `char *` now and a `const char *` in the future. Note that the deprecated conversion Kevlin's paper provides is specifically for string literals and not any other arrays of `char`. This code will fail noisily, it could also be fixed by a sensible vendor extension. In other words we can keep current code working while encouraging better code in the future.

The last of Steve's problems will cause a shock to quite a few of you. Consider:

```
char (& x)[4] = "Yes";
```

"What is that?" I hear you say. Well just read it and see: "x is a reference to an array of 4 char." I doubt that breaking this will break very much code. Note in passing that C++ (C also) does have array types, it is just that they decay for most uses to pointers. I somehow hope that implementors don't even try to mend any code of this kind that is broken by changing string literals to arrays of `const char`.

Normally points such as the above would result in a proposal being returned for reconsideration so that the paper at least mentioned them even if it was to recommend no action. At Stockholm this was not an option. When I arrived to support Kevlin's paper one of the first things I had to say was that either we make the change now (which I believed was for the better) or we would never be able to fix the problem. Somehow, John Bruns and I convinced Jim Welch of Watcom that going with the proposal was the lesser of evils. This shifted the work groups position so that (if I saw correctly) even Dan Saks reluctantly supported it going forward to the full committees. Once it got there it got enough support to go on to a formal vote but not before another problem was aired. Consider:

```
void foo(const void *);
void foo(char *);
```

and a call such as:

```
foo("I am not a void star");
```

With non-const string literals this code selects `void foo(char *)`, with `const` string literals the call is ambiguous. I think this will be a rare problem but at least it fails safe.

I had a nasty moment on Thursday night when I found Jerry Schwarz of Declarative Systems strongly opposed. I have a high regard for Jerry's opinions and I was worried that he would take a substantial block of X3J16 votes with him. In the event X3J16 voted 25-6 in favour and WG21 voted 7-0 in favour.

A cautionary tale?

Well if you want to contribute to language change you have either to get yourself to Standards meetings or you must find someone else who will act on your behalf. Standards are not necessarily about making life better or more consistent in the long term for the working programmer.

By the way, Jerry's reason for voting against the change was that it would produce several years of inconsistency between implementations and if we were going to have that we might as well go the whole way and not bother with deprecated conversions. It is quite possible that in other circumstances no change would have been made because half X3J16 wanted the conversions and half did not even though all wanted constant string literals.

I understand that Microsoft Visual C++ has been quietly implementing `const` string literals for several years. Anyone able to confirm that?

Francis Glassborow
francis@robinton.demon.co.uk

C++ Techniques

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

Alec Ross takes a step back from last year's circular discussion to consider the bigger picture, The Harpist continues his exploration of the Standard Template Library, Francis summarises the responses to last issue's question on member return types and Kevlin continues his excellent template techniques series.

Circles and ellipses revisited

by Alec Ross

The shapes of things to come

Last year's discussion on polymorphic objects, the "circles and ellipses" series [1-7] opened a variety of avenues for its readers to explore.

I find the circle/ellipse problem interesting, not just in itself, but for the thoughts it elicits on some OOA/OOD concepts, and implementation techniques which can be used for this and similar problems. The question has now been raised and well answered, but some further observations might be of interest.

This and following articles will look at some of the issues and approaches which can be used. In summary, these involve changes in views of type, techniques to morph objects defined on the stack as well as in free store, and extensions to related patterns.

Reverting to type

The notion of a "type", particularly as exemplified by an abstract data type, is a basic one for C++ programmers. I guess it informs our intuition, in ways of looking at computing systems informally and, for some, it provides a formal framework.

If we know what a type is by its operational definition, what about a definition which includes morphing some or all of its instances into other type(s)? And, just as a class can have nested classes within it and an object can have members which are objects, is it useful to consider a wider view of types which also includes clusters of related, interacting types in the sense of parameterised patterns? [8]

A programmer's concepts and use of type will be conditioned by their knowledge and experience of what is available and useful, and could be limited by ignorance or lack of imagination as to what might be possible. For example, for decades commercial data processing has used the concept of entities (corresponding to objects) and entity life history (ELH) [9], mapped to designs and implementations with fixed-size records consisting of a given set of data members only, and with no support for inheritance, polymorphism or generics. Whilst C++ has brought OO support, it has static typing plus inheritance polymorphism only and it does not directly support a notion of type where attributes and methods can be added or removed dynamically. This kind of behaviour is more widely in demand than for just a few relatively exotic systems with object frames with slots which can be dynamically populated with different members. Sometimes one might wish to dynamically change the implementation of a member function (on a per ob-

ject or class-wide basis) whilst retaining its semantics; or one might want to cut in different semantics; or perhaps it might even be desired to change the interface itself. The Common Object Request Broker Architecture (CORBA) of the Object Management Group explicitly recognises a possible desire to change an objects type dynamically, as does OLE. What is of interest here, however, are much simpler, more lightweight mechanisms – not spanning any process boundaries.

Life-cycle examples

As an example of this kind of requirement, system designs that use the concept of an object life cycle should be able to reflect this cycle fairly directly in code. That is, it should be possible to construct an object and mutate it through various stages with different behaviours before destroying it.

A given message would have different results depending on the current "type" of the object involved. To give a concrete example, one might have a class "Man" with, say, seven states. It might have a *GoToSchool***()** method in only one of its states. Alternatively, its public interface might offer this method for all age states, but give a different result depending on the state concerned. The first of these approaches could be implemented using derivation, with *GoToSchool***()** declared and defined (only) in a *Schoolboy* class publicly derived from *Man*. The second option could have *Man's* public interface declare the method with suitable definitions being provided in each class. (This second choice would typically result in the *Man* class having an interface which was the union of all derived interfaces.)

The change of perceived type should be simply and cleanly achieved in the client code. As an example illustrating this requirement, consider a personnel system for schools with C++ classes representing pupils and teachers. The system could create a *Pupil* object but might be asked to cope with this same pupil graduating and becoming a *Teacher*. If this single object were defined as an automatic, it would have a single symbol name and fixed storage allocation on the stack. There is something inelegant about implementing the state transition via a destructor-constructor call pair explicitly for each use at the client level. Also, for an instance which was an automatic, one would not be free to deallocate and reallocate stack store for the object which would typically use different amounts of store for the data members used by the two states. It seems more satisfactory if our client code can deal with the evolution of instances of a *Person* class through states of *Pupil* and *Teacher* – with any implementation nastiness hidden away.

More general patterns

These individual morphing transitions (e.g., circle to ellipse), and lifecycle examples can be seen as specific cases related to a number of design and programming problems and techniques. For example the change in type involved can be seen as a "real" change in type, or as a change in state of an object of a supertype whose type is preserved across the transition.

With this in mind, one can conceive a range of design patterns for the supertype:

- A binary switch (i.e., a flip-flop object with two observable states)
- a switch with unlimited state transitions allowed
- a counted flip-flop, i.e. one with a limit on transitions
    - with this limit dynamically adjustable
    - with this limit preset and fixed
- a one-time switch, as a special case of the above
- a bi-directional one-time switch (either state a -> b or state b -> a)
- a unidirectional one-time switch (i.e. the initial state is given, e.g. alive –> dead)
- An N-ary switch (with similar variants to the above)
- A ganged switch (changing two or more linked objects, types, or a mixture in synchronism with each other.)
- A switch carrying history (i.e., information on previous states)
    - retain all previous history
    - retain a sample of previous history
    - retain according to single fixed rule (eg all of the immediately previous state's data, or simply an indication of the previous subtype)

  - retain according to a predefined rule for each type
  - retain according to a run-time determined rule for each type
  - retain according to a run-time determined rule for each object
- A switch allowing dynamic change of its own morphic type
- A switch supporting various client views of its own type
- ... , and so on.

Whilst some of this indirection may seem fanciful, there is a very common use of state change in those schools of systems analysis and design which use the concepts of the Entity Life History (ELH), or Object Life Cycle (OLC).

Since programmed implementations generally have a time-space trade-off, fast implementations may involve carrying around some data from at least the immediately previous state. One might thus be tempted to make a virtue out of necessity here, and add "history" methods almost for free.

Getting hysterical

The data retained after a state transition could be complete or partial state information from:

- the last state
- all previous states
- the last of each different type of state

One would want to pick the simplest switch type suitable. In practice this could often mean a choice between variants of a given pattern with or without hysteresis – depending on the assumptions made. For example, the minimum storage cost associated with a totally amnesiac design would be desirable where there were many objects, and memory was limited. This option might also be appropriate where the lifetime states of an individual object followed an irreversible pattern; but any requirement to reverse the state change, jump to a previous state, or even access states' historic data would suggest a use of a type with hysteresis. A further option which could be used in the second of the above cases would be to keep an object's history record in a separate list, whether or not the object kept details of its own past states.

Storage of all of a previous state's data by the object could also simplify code – since there would be no need to implement any selective storage from previous members whose values were used in construction of the new state. Evidently too, some destructor calls could be avoided.

Has anyone seen or used patterns such as these?

Distinguishable states

In distinguishing types there are issues around boundary conditions. We have the concept of distinct types but also, possibly, limits on distinguishability. There may be issues of accuracy and precision in measurement, calculation and representation of real numbers in the available floating point formats. For example, in a morphable object which could represent a circle or ellipse, is a test for the eccentricity == 0 a valid test for its being a circle? If the eccentricity value were set to 0 by a constructor, and subsequently left unchanged, this test would be reasonable – but if it were the result of computation, computational accuracy would suggest that the test should allow some margin of error. Some values close to zero could correspond to either a circle or an ellipse, due to computational errors. Also, an ellipse with a very low eccentricity might be indistinguishable from a circle when displayed on a given graphics device. If the circle calculation were much faster, one might wish to use it, even if the eccentricity value indicated an ellipse.

One technique to assist in handling these cases is to introduce an intermediate state, or range of values, and deal with such border conditions explicitly. For example, one could have a test for a clear-cut circle and a clear-cut ellipse and for something in between. An object or value which fell into this latter category could be treated according to a defined strategy: being forced off the fence according to predetermined rules; or perhaps being given special treatment as a new, ambiguous state.

The problem here is of course much wider than the context of deciding on a type. There is a requirement for some kind of switch where each case corresponds in general to a range, typically of real numbers. The aggregation of cases might span some number range completely or there could be gaps and in some instances one might even want to allow overlapping ranges. It would be desirable to have some elegant, efficient and generally applicable algorithms and coding idi-

oms to handle this hashing. Would anyone like to contribute here?

Changing type: mechanisms

Further articles will explore some techniques to achieve type evolution of the basic kinds outlined at the start of this article, illustrated with C++ code.

Thanks

The author would like to thank Kevlin Henney for several helpful comments and suggetions on an earlier draft of the material in this introduction and in some following articles.

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

References

[1] The Harpist, "Related Objects", Overload, Issue 7, p 22-25

[2] Francis Glassborow, "Related Addendum", Overload, Issue 7, p 26

[3] Kevlin Henney "Circle & Ellipse – Vicious Circles", Overload, Issue 8, pp 22-25

[4] Francis Glassborow, "Circle & Ellipse – Creating Polymorphic Objects", Overload, Issue 8, pp 26-28

[5] The Harpist, "Having Multiple Personalities", Overload, Issue 8, pp 28-32

[6] The Harpist, "Joy Unconfined – reflections on three issues", Overload, Issue 9, pp 11-13

[7] The Harpist, "Addressing polymorphic types", Overload, Issue 10, pp 15-19

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: elements of Reusable Object-Oriented Software", Addison-Wesley, 1995, especially the "State" pattern, pp 305-313

[9] R. N. Maddison et al, "Information System Methodologies", Heyden/BCS, 1983, reprinted 1986.

The Standard Template Library – sorted associative containers part 1 set & multiset

by The Harpist

Before I discuss the set and multiset containers I would like to spend a little time on an issue that arose from my article in *Overload* 14. When I originally sent the item to Sean via Francis it contained a bug that my compiler could not detect. Fortunately Sean spotted it and corrected it. (I hope you appreciate Sean's comments as much as I do, without such a knowledgeable editor I think articles written by the one-eyed would lead the blind into terrible problems.)

However the problem is one that will bite many of us and is worth looking at. What is the difference between declarations A and B in the following code:

```
class Mytype {
public:
      Mytype(int);  // constructor
// whatever
};
main() {
int i=0;
Mytype m1 = i;       // decl A
Mytype m2(i) ;       // decl B
```

You may well think that it is merely one of style. Declaration A uses the C style of initialisation while declaration B uses the C++ function style initialisation. Indeed most books tell you that there is no difference. That is entirely wrong (though it did not have to be as the language could have defined them as equivalent).

Declaration A works by first constructing a temporary `Mytype` from `i` and then calling the copy constructor to copy the temporary into `m1`. These means that `m1` is technically constructed by a copy constructor. In practice all compilers use the licence given by the Working Paper and optimise away the copy – that is the construct the temporary in the memory for `m1`. However before they do that they have to check that the initialisation would work if the copy constructor was used. Until recently there have been a couple of cases where this would fail. The common one is where `Mytype` has a private copy constructor. The other main case was where the initialisation was more complicated requiring an extra type conversion with the result that two user defined conversions were used to go from the rhs to the

lhs. When the Standards Committee introduced the keyword `explicit` to allow constraining single argument constructors to construction and not for implicit conversion they introduced a new way for the above code to fail. If I qualify the copy constructor as `explicit` then declaration A will fail as that is an implicit use of the copy constructor.

Because of the way `auto_ptr<>` works its copy constructor must never be called unless explicitly required by the programmer who presumably knows what they are doing. This means that you cannot write:

```
auto_ptr< vector<Mytype> > mt =
                          new
vector<Mytype>;
```

Well, you can write it, but as soon as your compiler understands `explicit` it will spit it back. We need to change our coding habits and abandon the C style initialisation (unless it is C that we are writing). Start removing those '=' signs from declarations in your source code.

That brings me to another point that I had entirely missed. There is no point in writing code such as the above declaration. All the STL containers are expandable and so they must be using dynamic memory. We do not need to use `new` to create them. If that is not immediately clear to you, think a bit about why you allocate an object dynamically. Now why would you want to explicitly dynamically allocate a vector (deque, list etc).

On to this issue's topic

set<> and multiset<>

What is the primary characteristic of a mathematical set? It has no repeats. What is the primary characteristic of a STL set? It is ordered. This is true of both `set<>` and `multiset<>`. The difference is that the former is also a mathematical set while the latter may include 'duplicates'. Note the quotation marks, duplicates means objects that compare equal (strictly speaking both `Compare(t1, t2)` and `Compare(t2, t1)` return `false`).

Both `set` and `multiset` take three template parameters though two of them can be defaulted (once your compiler knows how to do that). The first template parameter is the type of object being contained, the second one is the functor (function object) that provides a comparison facility. The third template parameter is the allocator.

The purpose of the last template parameter is to provide a mechanism whereby you can support different memory models. STL provides a `class allocator` which is the default for the last template parameter in all container classes (including list, deque and vector). Compilers that do not support default template parameters have to fix this up behind the scenes because the last thing you need is to worry about allocators until you have to. The whole *raison d'être* of the STL is to save you having to worry about this kind of detail.

The second template parameter for `set` and `multiset` is a type that provides a suitable comparison. Note that this is not a function, to work as a template parameter we need it to be a type. The default for this template parameter is itself a template `class less<>`. The definition of this is:

```
template <T> class less {
public:
  bool operator()( const T & t1,
                   const T & t2) {
    return t1<t2;
  }
};
```

This is a typical function object template, and you will need to get used to these if you are to progress with a C++ style of object oriented programming. Note that `less<T>` depends on `<` being defined for `T`. If it isn't, you will either have to provide the definition yourself or you will have to provide your own functor for `set<T, yourFunctor >` and `multiset<T, yourFunctor >`. In other words you must provide a rule for determining a strict ordering of the objects that you are storing in a `set` or `multiset` container.

`set` and `multiset` each have three constructors. There is the (`explicit` qualified) constructor for an empty set that takes an optional comparison functor parameter and allocator, so I can create an empty set of `Mytype`s (another departure from maths as there are many distinct instances of the empty set, even of the same type) with:

```
set<Mytype> set_of_mytype;
```

or by providing a comparison functor:

```
set<Mytype, greater<Mytype> >
            descending_set_of_mytype
```

N.B. Could Sean clarify why the constructor implies that you could write:

```
set<Mytype> descending_set_of_mytype(
greater<Mytype>());
```

(Note that `greater` is another STL function object).

> *Note quite! Omitting the type in the template argument list causes the parameter Compare to default to less<MyType>. The constructor will accept any comparison "object" of that type. Unfortunately (for you) the type of greater<MyType>() is different to less<MyType>! You could however derive a class from less<MyType> and supply that although since the function call operator()() is not polymorphic you might get a surprise! – Ed.*

There is a straightforward copy constructor which is not declared `explicit`. And finally there is the constructor to allow construction of a `set` (`multiset`) from another container object. This is a template member function (or should that be member template function?) so that you can copy from any container.

> *The committee removed the editorial distinction between template function and function template some time ago – it was clearly too subtle. Originally, one meant the template and the other meant the instantiation... but I can't remember which was which!*
>
> *Note that you can construct a container from any pair of iterators that define a range [lo,hi) so the template constructor to which you refer is more flexible than you indicate – Ed.*

This requires at least two parameters of an appropriate input iterator type. For example given that `lst` is a `list` of `Mytype` then I should be able to write:

```
set<Mytype> example(lst.begin(),
                    lst.end());
```

to create a sorted `set` of the distinct elements of `lst` (duplicates as defined by `less<Mytype>` will be discarded).

Among the member functions of `set` (and `multiset`) you will find several versions of `insert()` and `erase()`. One version of each takes an object of the appropriate type and either inserts or erases an item (only inserts to a `set` if not already there, erases all matching items if any). Another pair of `insert()` and `erase()` member functions take a suitable iterator (pointer) for an object of the appropriate type. The iterator forms of `erase` only remove the specific item. There are also `insert()` and `erase()` for ranges defined by iterators.

Those are the most immediately useful member functions.

Now a task for you (because you certainly will not learn about the STL without using it).

Write a program that creates a container of words (making the first letter upper case) that only contains one word starting with each letter of the alphabet. You must be able to input words in any order, only the first example of a word starting with any letter will be placed in the container. The program should end when you have twenty-six words (those with non-Anglo-Saxon alphabets can use their own if the want), one for each letter.

When you have done that, write a program that will read in a file and print out an alphabetical listing of all the distinct words in the file.

Warning

I am not going to provide you with a solution to either of these problems though I would be happy to look at yours and to publish them. In other words I am not going to provide you with a cheat, if you want to learn to use the STL, you will have to try for yourself.

The Harpist

> *Please send all contributions directly to Francis so he can pass them on to The Harpist – Ed.*

The return type of member functions

by Francis Glassborow et al

In the last issue of *Overload* I asked you to put your thoughts about the return type of a member function on paper and send them to me. I had three submissions which is three better than none. My thanks to Chris Southern, Ulrich Eisenecker and Klitos Kyriacou. It would have been nice to have had a larger mailbag. Perhaps some of you were put off by my offer of a reward.

Chris wasn't, he just declared himself a non-competitor. So I will kick off with his contribution, follow with my bit then conclude with Klitos' winning submission. As Ulrich covered much the same ground as the others I shall leave his response out.

Chris Southern's view

Here is my minor contribution to a discussion of member function return types. It is not to be considered a competition entry. I can well afford to buy my own copy. Indeed it has been on my purchase list since it was referred to as a work in progress in the C++ Report.

First we must establish the parameters of the problem. We require the best choice for the return value for an arbitrary member function of an arbitrary class. The class independence of the problem is our first clue, it must surely preclude the choice of another class as candidate. This leaves us with the built in types and the class itself.

Perhaps one's immediate response would be `int` to return a success or failure result. See page 21 of *Overload* 14. Not under any circumstance a `bool`, as I am given by my betters to understand that this type is broken. Given the unknowable nature of the member function this can not be regarded as a general purpose better choice. Success or failure will not be an appropriate concept for all functions. The rest of the built in types are even less suitable. What general `float` or `char` could be returned from our hypothetical function?

We are left with the class itself. Now I must be getting somewhere as wiser heads have been here before me. The stream classes and section 6.5.1.1 of the *Design and Evolution of C++* bear witness. For a `class T` a good alternative to `void` for a return value for a member function is of type `T&` and of value `*this`.

The style permitted by the returning of the reference is that of cascading member function calls:

```
aSet.add( Oranges ).add( Lemons );
```

The reasons for using a reference are mandated by the functionality being provided here. The set being added to should be the same set, not a temporary that will be deleted shortly!

I am fairly certain that the last reference I came across to `bool` being broken was in an article by [Francis]. I am surely not alone in being far too unsure of my ground to justify such a statement to an employer or colleague. Could we please have a reasoned critique. It would also be nice to know what the standards committee gave as their justification for including it.

Chris Southern
csouthern@brasspaw.compulink.co.uk

What is wrong with bool

I know I am prone to strong language and often describe something as broken when others happily use it. I think that the current type of a string literal (now fixed by the Stockholm meeting of the C++ Standards Committees) is broken. Others happily fix their problems with various bandaids and splints. But you only use such when something is broken.

Conceptually a Boolean type should only support two values, 'true' and 'false'. You should not be able to do any kind of arithmetic with Boolean values and the only operators that should be supported are '=', '==', '!=' and '!'. In other words you should be able to compare for equality, assign and invert them.

The concept of Boolean has nothing to do with arithmetic. I do not believe that there should be any conversions to or from a Boolean type. Indeed, a Boolean type naturally only require a single bit of storage. To put it another way, a Boolean type should be packable in a way that is transparent to the user. Unlike any other type in C++, the conceptual size of a Boolean type is smaller than a `char`. That is immediately problematical because the `sizeof` operator returns an (unsigned) integer type that gives the storage requirement as a multiple of that required by a `char`.

It is possible to provide a user defined type that meets almost all the above criteria (I am not sure about supporting packing, if anyone would like to explore this I am sure that the idea would have more general use).

Unfortunately, such a user defined type has minimal value. The built in logical operators conceptually evaluate to a Boolean value. That means that a useful Boolean type needs to be a built-in type. Under continued pressure the C++ Standards Committees provided a Boolean type and called it `bool`. They also provided two keywords `true` and `false` as representing the two values of `bool`. They then defined the built-in logical operators as returning a `bool`. They

also defined the various conditional clauses (`if()`, `while()`, and the middle expression of `for()`and the easily forgotten conditional operator) as taking `bool` values.

Now C++ is required to support existing C code as far as possible. I do not think that the C++ community would have accepted the wholesale breaking of their existing code that would have resulted had there been no implicit conversions from arithmetic types to `bool`. Of course the code in question is of dubious merit but C never implemented a Boolean type so programmers had to improvise.

Though it is conceptually wrong, I can live with these inward conversions to `bool`. However when the issue was discussed various people pointed out that there is a body of existing code that depends on true/false taking the numerical values 0/1. We recently had an example in *CVu* where the adjustment for a leap year was handled by adding the return value of a function `leapyear()`. Conceptually `leapyear()` returns a Boolean value (either the argument represents a leap year or it does not).

I think the C++ Standards Committees should have had the courage to break such code (actually a little thought would show that very little code would be broken immediately. Only that which uses the values of logical operators would break). In the event, they lost their nerve (or to put it another way, some of the big users believed that too much existing code would break). I think that is sad but it is one of the compromises that occur when a language has escaped from its designers before the design is complete.

I undertook a test implementation of bool and analysed a lot of source code to see what it would break. My findings were that it would break very little although there was one unfortunate and rather common idiom: that of incrementing a Boolean using ++. That lead the committee to support ++ as an anachronism that meant "var = true". See the Design & Evolution of C++ for a reference to my work on this issue – Ed.

What, to my mind, finally moved `bool` from the class of heavily compromised features to the class of broken ones was the later decision to support the increment (++) and decrement (--) operators for `bool`. As I understand it this was to support passing `bool` as a type to a template. I am certain that instantiating a template that uses increment/decrement operators is conceptually wrong. Even if some elements of the template will work with a `bool` those parts that rely on increment/decrement will not work properly. The user should get an error if he/she attempts to use such functions. Now they will not even get a warning.

Decrement is not and never has been supported for bool. Increment was included in the original proposal – see above – and none of this had anything to do with templates! Just for a change :-) – Ed.

The result of all this is that if you want to return true/false from a function, or you want to provide a suitable conversion for a class such as many of the `iostream` ones you need to return a `const void*`. Note the use of `const`. I missed that refinement when I wrote about it previously. While you are getting a pointer, it is not one that is intended for use. There are no implicit conversions from a `void *`, the rules say that a null pointer is treated as false and all others are treated as true. In fact `const void*` provides almost exactly what we want for a Boolean type. Unfortunately the four operators that should work on a Boolean type do not and the built-in logical operators return a different type.

This feature of C++ will remain to cause unpleasant surprises to future programmers who first learn Java and then try to write C++. Perhaps some implementors will provide an extension that makes the C++ `bool` a real Boolean type so that we can check our code for silly hacks.

Why not take advantage of the source code analysis tools available on the market that already perform exactly this sort of checking within the standard definition of the language? – Ed.

My view (member function return type)

Member functions come in various flavours. There are a group of special functions (constructors etc.) which are not an issue here. Then there are the read access functions (that return an appropriate type to provide the required value), comparison functions (that return a Boolean value), operator functions that return an appropriate type (that which is most like the behaviour

of the built-in types). Finally there are two groups of functions that are really procedures (it is what they do that matters), these are the ones that programmers often declare as returning `void`.

This last group breaks into two sub-groups, constant member functions and the rest. A good example of the former is `printOn(ostream &)`. Personally, I am quite happy to have this function return a `void` because I am going to use it to support an `operator<<` function. There does not seem any strong reason for choosing another return type.

Having printOn() return ostream& is much more convenient – Ed.

That just leaves member functions that change the state of the object. It seems perfectly reasonable to use several such functions in succession. Being able to write something such as:

```
turtle.forward(10).left(90);
```

Seems better to me than

```
turtle.forward(10);
turtle.left(90);
```

Or to put it another way, if I have an object and change it, I should be left with a changed object rather than nothing.

Such functions should logically return a reference to the calling object. If you are worried about how this works with virtual member functions, relax because several years ago the C++ Standards Committees fixed that problem. The return type of a virtual member function can vary as long as the later return types are derived from the earlier ones (something I think is called covariance, but Sean will correct me if I am wrong).

This time you are correct! :-) – Ed.

Conclusion, member functions that modify the state of an object should return a reference to the object.

Klitos' view

Your article, "*Return from a member function*" (*Overload* 14) has aroused my interest and started me thinking. I shall arrive at my answer by addressing a number of concerns: (1) theory and concept; (2) conformance with established practice; (3) syntactic convenience; (4) performance issues; and (5) type safety. You mention set/put functions; these, and, I would imagine, most other typically void member functions change the state of an object in some way.

1. Conceptually, the result of applying a modifier function to an object is a modified object (even if only part of it – a data member – has been modified).

2. Throughout the existence of C and C++, built-in (and user defined) types have been modifiable using the assignment operator. Given '`int n = 1;`', the value of the expression '`n = 2;`' is the new value of `n` (an rvalue in C and an lvalue in C++). In the case of structures, given:

   ```
   struct complex {
     double re; double im;
   } c = {1, 2};
   ```

   the result of '`c.re = 3;`' is the new value of `c.re`.

So far, the evidence from both (1) and (2) suggests that a member function should return a reference to itself ('`return *this;`'), but (2) also begs us to consider making a data-member modifier function return a reference to the data member it is modifying. However, there are strong cases against the latter: it would break encapsulation, and many member functions modify more than one data member anyway.

3. Suppose you have a class '`Car`' holding various details, all optional, on a car. A function '`averagePrice(const Car&)`' returns the estimated price of a given car, using whatever information it has on it.

Example using `void` member functions:

```
Car car; // Default constructor sets all
         // details to 'unknown';
car.setAge(3);
car.setMilage(25000);
cout << averagePrice(car) << endl;
```

The above prints the average price of all 3-year-old cars with 25000 miles on the clock. If member functions returned a reference to the object, the above could have been written more conveniently:

```
Car car;
cout << averagePrice(car.setAge(3)
                .setMileage(25000)) <<
endl;
```

Moreover, if all you want to do is find the average price of 3-year-old 25000-milers and do not want to have a '`Car`' object hanging around after that, you can use a temporary object:

```
cout << averagePrice(Car().setAge(3)
                .setMileage(25000)) <<
endl;
```

This is not just syntactic sugar; it is something you cannot do at all with `void` member functions: that is, you can't use `void` member functions in this way on unnamed objects.

4. The extra '`return *this;`' statement at the end of a member function can consume a few CPU cycles unnecessarily if the return value is not used. However, many member functions are inline, and a good compiler will optimise out the extra unused statement. If a member function is too long to code as an inline function, you can make it a private implementation function and have the public member function call it before returning the object:

```
class Car {
public:
  // [other methods...]
  Car& setMake(string mk)
  { doSetMake(mk); return *this;
}
private:
  // [other methods...]
  void doSetMake(string mk);
  // Defined in a separate file.
 }
```

5. There is a small problem with type checking in the presence of inheritance:

```
class Van : public Car {
public:
  Van& setMaxLoad(double);
  // [other methods...]
};

// This won't compile:
cout <<
averagePrice(Van().setAge(3)

.setMaxLoad(100));
// This will compile but is very
// clumsy:
cout << averagePrice(

dynamic_cast<Van&>(Van().setAge(3
))

.setMaxLoad(100));
```

What would you suggest as a satisfactory solution? I can't think of any, but I don't think the problem is a dangerous one. That is, it may prevent code brevity, but it does not allow you to call a method on the wrong type.

It will be interesting to see whether self-returning member functions become more common in the future.

Klitos Kyriacou
kkyriacou@datastream.com

Conclusion (Francis)

I think the most important lesson from all this is do not just emulate code you find in books. Think about it and for all but the best books your solution may well be better.

Another thing for you to consider

Now for something completely different.

I came across the following code recently:

```
class X {
// full class definition
};
class X1: public class X {
// nothing but the constructors for X1
that
// do nothing
// except call the corresponding
// constructors for X
};
```

Can you think of any practical use for such an apparent redefinition?

Francis Glassborow
francis@robinton.demon.co.uk

My experience is that most member functions that return void ought to return a reference (or const-reference) to the class type. Call chaining is such a useful technique that once you start using it you will wonder how you managed without it! – Ed.

/tmp/late/* Constraining template parameter values

by Kevlin Henney

A previous */tmp/late/*, "Constraining template parameter types"* (*Overload 12*), explored how certain type constraints could be enforced at compile time by the programmer. The aim is to attempt to state, in code, specifications that would otherwise be held as comments or detected explicitly at run-time. Unless you enjoy testing and believe that cure is better than prevention, a higher level declarative approach has all the hallmarks of good practice.

Types are not the only thing that can be constrained. More generally, applications rely on certain assumptions that are platform or compilation specific. Documenting these value based constraints is all very well, but the truth is that no matter how good our software process, the spec is in the code. Unless we can state implementa-

tion constraints in the code, they are effectively invisible.

Don't tell the user, tell the compiler

One way to enforce these constraints is to use *assert* within a function. This is not a good idea: unless that code is guaranteed to be executed, the constraint will never be checked. But, for example, checking that a given constant supplied by the user is in a range specified by a library is surely something that, being constant, should be checked at compile time rather than being left to vagaries of run-time path coverage? Appropriate use of the assert macro is about as rare as a fulfilled electoral promise.

The most obvious compile time mechanism is the preprocessor. So, for instance, here is an attempt to ensure that the platform you are running on is a two's complement machine:

```
#if ~1L + 1L != -1L
#error "This is not a two's complement
box"
#endif
```

And here we attempt to ensure that the alphabet is encoded continuously and sequentially:

```
#if 'a' + 1 != 'b' || 'b' + 1 != 'c' ||
...
#error "Character encoding has holes"
#endif
```

The problem is that there are no guarantees that the preprocessor is using the same character set or arithmetic as the execution platform. This separation of translation phases and behaviour is common in cross compilers. Another obvious problem with the preprocessor is that only preprocessor constants may be used, i.e., no **const** or **enum** constants.

The specialist

We can reify an assertion, i.e., treat the actual assertion as an object, and then write

```
static compile_assert<~1 + 1 == -1>

twos_complement;
```

Template specialisation is the key to the solution:

```
template<bool expression>
struct compile_assert;
template<> struct compile_assert<true>
{};
```

Here we forward declare a template class, but provide no definition. We provide a specialised definition for the case where the compile time expression being tested is **true**. And in the case of **false**? Well, the compiler doesn't know what the *compile_assert* would look like in this case so it fails to compile. In other words, what we wanted. You will find that naming your assertion variable meaningfully helps.

The cost of this is minimal: no run-time overhead, and a minimum alignment of static memory taken up. If your compiler does not yet support **bool** and you are concerned that a specialisation on 1 misses all the other valid, non-zero cases:

```
template<int expression>
struct compile_assert {};
struct compile_assert<0>
{ compile_assert(); };
```

Here the class is defined for all non-zero cases, and for 0 the constructor is inaccessible, and hence objects of this type are undeclarable. Note that I have used the older specialisation syntax — if your compiler does not support **bool**, it is unlikely that it supports the newer full specialisation syntax (i.e., **template<>**).

In range

It is possible to make certain kinds of assertion easier to write using derivation. We can specialise expression types, e.g.,

```
template<int value,
         int minimum, int maximum>
  struct in_range : compile_assert
    <minimum <= value && value <=
maximum>
{ };
```

This will fail to compile if value is not in the range [minimum, maximum]:

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

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

The sequential alphabet problem (originally posed in *Overload 12*, *"Rot in L"*) can be solved using a similar method:

```
template<char value, char next>
  struct in_order : compile_assert
    <value + 1 == next> {};
```

We can group assertion expressions together for cohesion, convenience and to save (a marginal amount of) executable space:

```
static const bool alphabet_in_order =
    (in_order<'a', 'b'>(),
     in_order<'b', 'c'>(),
     ..., true);
```

Type based properties may also be asserted on, e.g.,

```
template<typename first, typename
second>
  struct equal_size : compile_assert
    <sizeof(first) == sizeof(second)>
{};
```

Use **class** if your compiler doesn't yet support **typename**. Here we document a common assumption in checkable form:

```
static equal_size<void*, int>

pointer_int_representation;
```

Be warned, however, that in the case of derivation from a failed compile time assert the diagnostic support you get from some compilers is — to put it generously — useless.

In bits

Value constraints can be taken a long way. Here is a slightly frivolous example that indicates how far:

```
template<int digit> struct bit;
template<> struct bit<0>
    { static const int value = 0; };
template<> struct bit<1>
    { static const int value = 1; };
template<int digits> struct bin;
template<> struct bin<0>
    { static const int value = 0; };
template<int digits> struct bin
{
    static const int value =
        bin<digits / 10>::value * 2 +
        bit<digits % 10>::value;
};
```

So what does it do? It allows you to specify constants in binary easily:

```
bin<100101>::value == 37
bin<11001100>::value == 204
```

It has its limitations, and it can be fooled, but it illustrates the possibilities of value constraint techniques.

Summary

Error detection of static value based constraints is too important to be left until run-time; templates provide a construct through which some of these may be expressed — and violations caught.

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

editor << letters;

Sean,

Francis advised me to contact you as you may be able to help by publishing a plea for help in *Overload*.

I need a real "Noddies Guide" to writing programs to use OLE etc from BC5. The BC documentation does assume that the writer knows much, much more about the concept than I do. If anyone can direct me at the "Idiots Guide to OLE and associated matters" it would be a help.

Regards

Allan Newton
amnewton@iee.org

If anyone can help Allan, please contact him directly.

Dear Sean

Is Francis's code (*Overload 13*, page 7) for *setname()* safe yet? Suppose the names were of form "Forename Surname" and I wanted to remove the forenames. It would be tempting to use code like

```
char *q = strrchr(record.getname(), '
');
record.setname(q+1);
```

strrchr() also handily removes the const-ness, so that things like

```
*q = tolower(*q); are possible.
```

Is this what is meant by encapsulation?

Regards

Graham Jones

The C++ Standard Library provides two overloadings for strrchr that preserve const correctness – given a const char, that's what you get back.*

However, if you subvert const somehow and feed setname part of the same storage that it was already using, yes, it will fail!

I thought you might like another bug report for Visual C++ (V4.1)...

```
//
=====================================
// Suspected bug in Visual C++ V4.1
```

```
// ------------------------------------
--
// I wanted a vector of pointers to a
// nested, polymorphic class.  Simple,
// just use the standard vector
template:
//
//  vector<Outer::Inner*> my_vector;
//
// But VC4.1 says:
//
//  ... error C2440: 'initializing' :
// cannot convert from 'struct
// Outer::Inner' to 'struct Outer::Inner
**
// ' (new behavior; please see help)
//  ... error C2439: 'current' : member
// could not be initialized
//
// At first I thought this might have
// something to do with the ObjectSpace
// STL<ToolKit>'s implementation of
// vector<T>, but I have simplified the
// code to the point where there are no
// templates.
//
// There is a workaround (of sorts):
don't
// use a nested class. The problem
// also goes away if the 'current'
member
// is initialised to 0 or from a
// static const data member, but this
means
// changing the STL<ToolKit> code,
// which I am loath to do. Using
typedefs
// doesn't help.
//
// The Visual C++ Knowledge Base
contains 6
// references to C2440; none of
// them documenting this problem (as far
as
// I can see).
// ------------------------------------
--

#ifndef BUG_FIX
    struct Outer {
        struct Inner {};
    };

    typedef Outer::Inner **T;
#else
    struct Inner {};
    struct Outer {};

    typedef Inner **T;
#endif

struct Junk
{
    T current;
    Junk () : current(T()) {}
                    // errors C2440 and
C2439
};

Junk test;
```

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

Looks like a bug to me – keep 'em coming Phil!

The following letter appeared on ACCU.general:

OK, I just found a bug in Visual C++ v4.2 so to stop everyone else tracking down the same problem....

The *CopyElements* function in MFC 4.2 has been broken (it was OK in previous versions). It is supposed to copy *n* elements from *src* to *dest*. The current implementation ends up making *n* copies of the first element of *src*.

```
// \msdev\mfc\include\AFXTEMPL.H Line 76
template<class TYPE>
inline void AFXAPI CopyElements(TYPE*
pDest, const TYPE* pSrc, int nCount)
{
  ASSERT(nCount == 0 ||
         AfxIsValidAddress(pDest,
                  nCount *
sizeof(TYPE)));
  ASSERT(nCount == 0 ||
         AfxIsValidAddress(pSrc,
                  nCount *
sizeof(TYPE)));

  // default is element-copy using
  // assignment
  while (nCount--)
    *pDest++ = *pSrc;
}
```

This last line should in fact be

```
*pDest++ = *pSrc++;
```

Later,

Steven Youngs
steve@ncgraphics.co.uk

How on Earth did that get past Microsoft's supposedly wonderful QA system?

See the inside back page for more information on ACCU.general.

Reviews

Whilst this is not strictly a standalone review – it refers back to a review in CVu 8.4 – I think it gives enough depth to warrant being treated as a review.

Java in a Nutshell
reviewed by Chris Southern

David Flanagan

O'Reilly Associates

ISBN: 1-56592-183-6

Price: £10.95

Soft cover, 438 pages.

My curiosity is currently piqued by this here Java beast. I suspect that this is not an uncommon complaint among the readership of *Overload*. I had been leafing through the available books and refusing to part with cash for thick CD-ROM wrappers on the basis that I run a Macintosh rather than a Sun or a Windows machine. Then I saw a review in *CVu* for the O'Reilly book, Java in a Nutshell by David Flanagan. The price was good, the review favourable, and I have bought Nutshell series books before and been very satisfied.

First I have a minor point of order. Java does have a sort of multiple inheritance. Not, it has to be admitted, of actual behaviour, but of the 'contract to behave' that is made by the class definition in C++. This is done by the interface type which is similar to a pure virtual base class.

Secondly and of much more importance is the quality of code in some parts of the Flanagan book. I know the book's subtitle is 'A Desktop Quick Reference for Java Programmers' and that some may feel that this excuses it from this sort of criticism.

However, the book does not live up to its subtitle. The API Quick Reference that forms part 4 of the book is too brief in its description of methods to survive without part 2, the code examples. These are introduced with 'You can study and learn from the examples, and you can adapt them for use in your own programs.'

The book seems to me to fall between two stools. It is not detailed enough on the API to be a desktop reference book, and has too many pages devoted to cross references, API inheritance diagrams and 'man' pages for the Sun tools to be a first tutorial book. I do think that the inheritance and cross indexing will be useful. I just wish that the book had not tried to serve too many masters.

In the class *AllComponents* (example 5-4 in the book) there is a method 'constrain'. This is essentially coded as one would a global function in C++, all its data are passed as parameters. Java does not have the concept of non-member functions so this method basically has 'random' cohesion with its class.

What is worse is that it is basically a constructor having non-default member values for another class. As such I think that it should have been implemented as a constructor for a class derived from *GridBagConstraints*.

In the example on exception handling (2-3) the 'finally' clause for the function 'b' just prints a newline, however, from the description of 'finally' clauses given in the text the local catch block will be executed first. Therefore when the exception is caught rather than propagated the newline in the output text will be in the wrong place. Not a major disaster but it gives the wrong lesson about finally and catch block handling.

Since Chris is, like myself, a Mac developer, I asked him why he picked the Nutshell book instead of one of the Mac-specific books. He responded:

I bought the O'Reilly book at the recent Mac Shopper/Internet show where I actually saw 'Teach Yourself Java For Macintosh In 21 Days' first. But reading the description of the limited version Roaster got the impression that it was good only for project files on the CD.

While I can't think of a good way of limiting a compiler for this purpose I felt that the learning by doing method needs something more than typing in code.

Luckily the last isssue of my Metrowerks development environment arrived shortly thereafter containing not just one but *two* Java development kits!

The 1.0.2 JDK from Sun was provided only for completeness sake the release notes said. However, the integrated version of the interpreter does not support java.lang.System and so can't run the Hello World example – no streams.

The reference CD also included a copy of Metrowerks book 'Learn Java on the Macintosh' in Acrobat format, with the exhortation to buy a real copy if found useful. The authors of the seven books included do not get royalties for this

distribution, and will no doubt be greatly heartened by this plea.

Chris Southern
csouthern@brasspaw.compulink.co.uk

Is there no smiley for heavy irony?

News & Product Releases

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

OMT User Group Seminar

The OMT User Group is hosting a seminar day on 1st October aboard the HMS Belfast, which is docked on the Thames in London, to discuss Reuse with OMT. There are many myths—and much hype—surrounding the relationship between OO methods and reuse. This seminar day focuses on the practical ideas and technology that can be used to enable different levels of reuse: software tools, project management, development methodology, patterns, frameworks and class libraries.

The day will be extremely useful for any current or potential OMT practitioners and will provide an excellent opportunity for discussion on any aspect of OMT and the forthcoming Unified Modeling Language (UML). The seminar is open to both user group members and non-members.

A separate area has been set aside for demonstrations of some of the main CASE tools supporting the OMT method and UML notation. Representatives from the CASE tools vendors will be available to provide information and answer questions on the products. Leading publishers will be displaying recent and classic OO books.

The price of attendance per individual will be $69 for members and $99 for non-members. Individual membership of the group is $39 pa. Corporate membership is $129 with 5 named individuals, or $199 with 10 named individuals. All members receive a quarterly newsletter and book reductions on selected books from leading publishers. All prices are exclusive of VAT.

For further information on either user group membership or seminar attendance, please contact either:

Jan Bevans
jbevans@qatrain.mhs.compuserve.com

Kevlin Henney
khenney@qatrain.mhs.compuserve.com

on 01285 655 888 at QA Training, Cecily Hill Castle, Cirencester, Gloucestershire, GL7 2EF.

Contact details are also available under http://www.qatraining.com.

ACCU and the 'net

ACCU.general

This is an open mailing list for the discussion of C and C++ related issues. It features an unusually high standard of discussion and several of our regular columnists contribute. The highlights are serialised in *CVu*. To subscribe, send any message to:

```
accu.general-sub@monosys.com
```

You will receive a welcome message with instructions on how to use the list. The list address is:

```
accu.general@monosys.com
```

Demon FTP site

The contents of *CVu* disks, and hence the code from *Overload* articles, eventually ends up on Demon's main FTP site:

```
ftp://ftp.demon.co.uk/accu
```

Files are organised by *CVu* issue.

ACCU web page

At the moment there are still some problems with the generic URL but you should be able to access the current pages at:

```
http://bach.cis.temple.edu/accu
```

Please note that a UK-based web site will be operational in the near future and this will become the "official" ACCU web site. Alex Yuriev has done a great job supporting the ACCU web site from the US – thanks Alex!

C++ – The UK information site

This site is maintained by Steve Rumsby, long-serving member of the UK delegation to WG21 and nearly always head of delegation.

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

C++ – Beyond the ARM

My pages haven't been updated for a while. Now this issue is finally out of the way, I intend to spend time rewriting and substantially updating the information on them.

```
http://www.ocsltd.com/c++
```

Any comments on these pages are welcome!

Contacting the ACCU committee

Individual committee members can be contacted at the addresses given above. In addition, the following generic email addresses exist:

```
caugers@accu.org
chair@accu.org
cvu@accu.org
info@accu.org
info.deutschland@accu.org
membership@accu.org
overload@accu.org
publicity@accu.org
secretary@accu.org
standards@accu.org
treasurer@accu.org
webmaster@accu.org
```

There are actually a few others but I think you'll find the list above fairly exhaustive!

Credits

Founding Editor

Mike Toms
miketoms@calladin.demon.co.uk

Managing Editor

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

Production Editor

Alan Lenton
alenton@aol.com

Advertising

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

Subscriptions

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

Distribution

Mark Radford
mark@twonine.demon.co.uk

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 16* (November/December) should be submitted to the editor by October 21st.