ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 13

April/May 1996

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

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

£3.50

Contents

Editorial

An object lesson in usability

"With all their damned computers, why does it take them so long to allocate bookings to people?" asked the irritable New Zealander standing behind me in Dallas airport as we waited to get new tickets after being stranded by storms en route to our planned destinations. He had a point. We had been queuing for ages and now we could see three staff operating one desk with five other desks each manned by one person. How *could* it take so long for six desks to process this queue of people, and why did it take three people to wrestle with one console at times?

When it was my turn at the desk I tried to figure out what the ticket staff had to go through to process a customer. There's minimal information to enter: destination, seat preference, maybe a few other details. Then the system should surely offer a selection, and after confirmation the ticket should be printed. What I watched was, instead, a flurry of keystrokes mainly using the cursor keys, enter and the escape key. Although I couldn't see the screen, I would hazard a guess that the data entry screens were packed with fields and the menu structure necessitated continuous navigation up and down the hierarchy.

I wondered who was involved with the design. Were the desk staff consulted? Did they understand what the analysts were asking? Perhaps the data entry system was a direct replacement for a paper-based forms system? Who knows. No doubt, the staff had become used to the day-to-day operation of the system but from my point of view it looked very cumbersome.

I expect we've all seen such systems – seemingly simple query and answer situations bogged down by a wealth of options and backtracking choices. I consider myself lucky to have been involved with a project many years ago that deliberately provided a shortcut. I worked at Sun Alliance in their motor insurance division developing the software that would support the Motorist 50+ insurance quotation product. Written in COBOL and assembler on IBM 8100 minis, the system was a complete in-house solution. It was ambitious, integrating a full-screen word-processor (written in-house) with the database system to simplify generation of form letters as part of the quotation system. It was also running late. Someone came up with the idea of a "quick quote" subsystem that could handle the majority of quotes on one screen. The customer (from another Sun Alliance division) loved the idea and sat with me as I worked on the screen design. The idea was that a telephone operator could fill in the form, left to right, top to bottom asking the caller simple questions, then press enter and get quotes back for "third party", "third party, fire and theft" and "fully comprehensive" insurance, again all on one screen. On confirmation from the caller, the operator moved to the billing screen to finish the transaction. Simple.

Any unusual conditions (medical, convictions, etc) would cause a referral to the normal, multi-screen quote system but the key issue was that the vast majority of cases could be handled by the quick quote subsystem efficiently. The data-entry was co-designed by the people who would use it, capturing the most important data in the most natural way.

Was this system so unusual? Ten years later, are more systems being designed this way? I don't know, but from my observations of desk staff struggling with their computer consoles I strongly suspect the answers are "yes" and "no" respectively.

Next time you're involved with GUI design, ask yourself what is the most common problem it is trying to solve – could an additional shortcut screen improve the usability? If so, beat on your analysts and your managers – campaign to make systems useable!

On the move

As I write this, stranded in the Harvey Hotel in Dallas, I don't yet know what this issue will contain beyond the articles I have written myself and the final part of Roger Lever's debugging article. Travel broadens the mind, they say, and the last three weeks driving through California have broadened mine. As usual the ISO / ANSI C++ meeting proved enlightening, I learnt about "guiding functions" for templates and a host of other subtle issues. I encountered the scene described above, and I learnt a hard lesson of being self-employed and having to tackle the sharp end of wrangling over contracts when you work directly for a company without the safety net of an agency. Collegues have been quick to assure me that my experience is not unique although I hope

it is not too common either. I imagine that ACCU's membership contains quite a few contractors so I would be interested in hearing cautionary tales which could be published (author withheld, if desired) in either *Overload* or *CVu*. In my case, half of my last invoice was withheld "pending completion" despite the contract being a fixed daily rate with no written deliverables or acceptance criteria. Given the Deputy Prime Minister's recent admission that as an entrepreneur he used to "string along"

FULL PAGE ADVERT GOES HERE

his creditors, I suspect it may be symptomatic of a more widespread commercial malaise – it saddens me that we, as professional programmers, have to wrestle with issues outside our chosen arena of expertise to avoid being eaten by the sharks.

Sean A. Corfield
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++.

Francis takes a deeper look at the code he presented in his guest editorial in *Overload 12*, I consider the problems with introducing virtual inheritance into existing code, Roger Lever presents the program that led him to develop the debugging code presented in the last few issues and Nigel Armstrong discusses a real-life example of poor class design.

Those problems revisited

by Francis Glassborow

When I wrote the article that Sean turned into a guest editorial I had not intended to say anything more about the two pieces of problem code. My mail has convinced me that this was a mistake. It also showed that a considerable section of the readership of *Overload* needs those less high-powered articles that I was asking for.

Though I had not asked for a response, I received over a dozen attempts to identify the problems. I say attempts because one writer more or less got them right (though he dismissed one real problem as being unlikely to happen in real code) and one identified one of the problems correctly. The rest missed entirely, one answer even ignored details given in the text.

I am afraid that I elided too much code for some of you. Of course, buried in the commented section of my sample code was such essentials as **public:** so that the code could compile. In fact both pieces of code were extracted from code that not only did compile, but must compile.

Problems with new

Most of my correspondents seemed to suspect the use of **new** without checking a return value. The problem with that is that up-to-date compilers use a version of **new** that throws an exception if there is a failure to allocate memory. In the circumstances I did not want to cloud the issue by encapsulating calls in **try** blocks because what I wanted you to focus on was elsewhere.

I hope that there will be plenty of articles on writing code for exception handling environments which will explain the need to consider encapsulating uses of **new** in constructors so that destructors will guarantee the use of **delete** even when an exception is thrown over the dynamic object.

Correct ways of using **new** are just one of the many subjects that you will need to learn about if you are moving from the C++ of the 1980's to that of the end of this decade.

Polymorphic arrays?

Consider the following code:

```
// base.h
#ifndef BASE_H
#define BASE_H
#include <iostream.h>
class Base {
       int i;
public:
       Base ();
       virtual ~Base();
       virtual void printon(ostream &
                              =cout)const;
}
#endif

// derived.h
#ifndef DERIVED_H
#define DERIVED_H
#include "base.h"
class Derived {
       int j;
public:
       Derived ();
       ~Derived();
       void printon(ostream &
=cout)const;
}
#endif

// app.cpp
#include "derived.h"
int main(){
       Base * pb;
       pb = new Base[10];
       for (int i=0; i<10; i++)
              pb[i].printon();
       delete[] pb;
       return 0;
}
```

All I want you to focus on is the compilation of *app.cpp* which is why I have not included any

implementations of *Base* and *Derived*. They would only be a distraction. I am also assuming that **new** will succeed.

Now what code will your compiler generate for *pb***[***i***]**? It will have to generate code to compute the address of the *i*th object of size **sizeof(***Base***)** beyond the address stored in *pb*. The **sizeof(***Base***)** is a static (compile time) property of *Base*, not a dynamic property. The designers of C++ could have arranged for the **sizeof** a polymorphic base class (one with a virtual function) to be stored in the *vtbl* but they did not do so. The **sizeof** objects cannot be determined dynamically (at run time). Well to be strictly correct, you could provide such information via a, possibly **static**, member of a class with a **virtual** member function (cannot be **static**, as this is another facility that C++ chose not to provide) to return the value. However trying to use this information to compute the address of the *i*th element of *pb* dynamically without losing the type information needed to dispatch to a virtual member function is too complicated to even consider.

Of course, in the above code static determination of the address is fine because the static type of an element of *pb* is the same as the dynamic type (they are both *Base*).

Now think what will happen if we change the line *pb* = **new** *Base***[10];** to *pb* = **new** *Derived***[10];**

The compiler will still be required to compute the addresses of elements of the dynamic array based on the statically determined **sizeof(***Base***)**. But *Derived* objects are bigger than *Base* ones. In other words the compiler will get the address of all but the first element wrong. The sad thing is that the compiler knows at the time that you create the dynamic array that it will probably go wrong but remains silent. Actually it has enough information at compile time to determine if your code can work (it knows if there is a mismatch between the **sizeof** the static type and the dynamic type.) I would hope that quality compilers would have a switch that allowed it to classify this as an error.

I hope this convinces you that you cannot call member functions for elements of an array unless the static and dynamic types of the elements are the same (or at least have the same size).

So how does this relate to **delete[]** *pb*? Well the compiler expands that line to something like:

```
for(int temp=0;
       temp<hidden_number_of_elements;
       temp++)
              (pb+temp)->~Base();
release_memory();
```

That is it calls the destructor for each of the elements of the array before it releases the memory. It knows how many elements have to be destroyed because that information is hidden away when you use **new[]** to create an array rather than a single object. Note that I am assuming you are using the library versions of **new**, **new[]**, **delete** and **delete[]**, if you aren't life can get more complicated. Even if you do provide **operator delete** and **operator delete[]** you still cannot fix the problem because use of these functions are determined by the static type of a pointer (i.e., what the pointer has been declared as, not what it actually points to).

The upshot of all this is that **delete[]** *pb* must step through the elements of the array and so the **sizeof** these elements must be correct. This effectively means that you cannot have arrays of a polymorphic type. If you must use C style arrays (rather than using an STL container, or a hand coded container) and want polymorphic behaviour you must deal with arrays of pointers. Note that this restriction does not only apply to explicitly dynamic arrays (that is ones you create with **new[]**) but also to the implicit ones created by initialising an appropriate parameter with the address of an array. For example:

```
void fn (Base array[], int size);
```

Will not work if you try to call it with:

```
Derived darray[10];
fn(darray, 10);
```

I think that C++ programmers must learn to leave raw arrays to the C programmers. Using them in C++ is a recipe for eventual unexpected behaviour. Instead of constantly checking that what you are doing is safe, learn a way of achieving your ends safely (learn to use the STL).

Self assignment

All the good books tell you to start your definition of copy assignment like this:

```
const T & operator = (const & T t) {
       if (this != &t) {
       // rest of code
       }
       return * this;
}
```

This is wrong! Yes, read that again. I am going out on a limb and declaring that the carefully

thought out and justified code found in all the best books is a mistake. It is a sledge hammer to crack a nut, but worse than that it focuses the programmers attention in the wrong place. The problem isn't with **operator=** but with the following pattern of code:

```
delete px;
px = new X(ax);
```

or equivalents. That is, every time you release the resources held by a pointer before attaching new resources to hold a copy of some object. Whenever you do this you must first consider if it is possible for the released object to be the one you are intending to copy. More simply, the problem isn't self-copying but the more specific case of self-copying a dynamic (sub-)object.

If your objects contain no dynamically provided resources then there is no need to check for self-assignment and leaving out this check will almost certainly marginally improve the performance of the class.

If your class does include dynamically provided resources, this provision will need to be handled by various constructors as well as the copy assignment operator. Such code (providing dynamic resources) should ideally only be written once as a member function with appropriate access qualification. The following expansion of my second example from *Overload 12* demonstrates this:

```
class Record {
       char * name;
public:
       void setname(char * s){
              if (s == name) return;
              delete[] name, name
                  =new char[strlen(s) +
1];
              strcpy(name, s);
              return;
       }
       const char * getname()const
       { return name;}
       Record (char * n =""): name(0)
       { setname(n); }
       Record (const Record & r):
name(0)
       { setname(r.name); }
       const Record & operator = (
                     const Record & r) {
              setname (r.name);
              return *this;
              }
       ~Record() { delete [] name; };
};
```

I have placed all the code in the definition to save time. Of course this is greatly simplified code, but note that if I change the mechanism for storing a name I only have to re-implement two functions. Also note that all constructors first initialise the pointer to null. I believe that this is an example of a far more important rule. Pointers should at all times point to either an object or null. That is why I try to follow uses of **delete** by an immediate re-assignment and emphasise that by using a comma instead of the more normal semi-colon. (By the way, for those that do not know, you may safely use **delete** and **delete[]** on a null pointer)

Problems should be tackled at the point they occur.

Breaking data hiding

At least one correspondent believes that the *getname***()** function in the above code is wrong. Now I might agree that the data should implement *name* as a *string* object but even then, how is the user to have access to the name? It seems too restrictive to say that the user cannot have any access. It is usual to provide read access to data by returning constant references even though a silly user could cast the **const** protection away. C-style arrays are handled through pointers (references to arrays are possible but unnecessarily complicated). What options has the class designer? Having decided to use an array of **char** to store characters there are three:

Return a **char *** to the original data. Definitely wrong as it does not protect against accidental access by the user.

Return a **char *** to a dynamically created copy of the original. No use, because the creator of dynamic resources should be responsible for releasing them.

Return a **const char *** to the original. This is semantically equivalent to using a constant reference in other circumstances and so should be as acceptable as they are.

Remember that access control is only intended to protect against accidental use of object data. If you insist on breaching data protection just declare an appropriate **friend** function in the header. This does not change the layout of a class so the compiled implementation code should still work. However if you ever do this expect to be taken into a large field and invited to dig your grave.

Conclusion

Well that is it for the time being. Let me finish with a question for the experts to mull over.

C++ recently introduced a keyword **explicit** to qualify constructors so that the compiler cannot use them for implicit type conversion. Why is it not necessary to extend the use of **explicit** to type converters such as **operator int()**? The answer is obvious when you see it, but it took me several months to get there.

Francis Glassborow
francis@robinton.demon.co.uk

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

by Sean A. Corfield

Introduction

In part IV I looked at the type system and said I would examine some of the implications of converting the inheritance hierarchy to use mixins for templates. First of all, I'm going to take a brief diversion to look at what can be considered good and bad in abstract base classes.

The ABC of ABCs

When I first started designing the analyser, I didn't have much experience with OOD although I had spent about a decade designing and building compilers, interpreters, optimisers and so on. Many of the base classes within my original type hierarchy were concrete classes – I was deriving *StructType* from *ClassType* to start with! During the maintenance cycle, extra classes were added into the hierarchy and base classes were generally made into abstract classes.

One of the ongoing problems this caused – indicative of how heavily poor design is punished in OOP – was that base class constructors tended to have quite a few arguments. In particular, *NamedScope* (see part IV) ended up with four or five constructor arguments. Since there were several layers of classes below that, all those arguments had to be supplied to the most derived class constructor and then passed back up the inheritance chain in the mem-initialisers. Sometimes this will be unavoidable but quite often it is simply due to having inappropriate state information in an abstract base class.

If a base class is truly abstract then there will be no member data – state – within it. Does this sound extreme? Well, consider what member data actually means: it implements state. That means that an implementation decision has to be made as to how to represent that state. Sometimes the representation is easily determined by the operations on the base class (e.g., **void** *set***(int); int** *get***() const;**) but mostly the protocol represented by an ABC is more complex and does not directly suggest an implementation. Such state information can be provided by an implementation class which is either referenced from the base class (using a pointer or reference) or derived from the base class (forming one side of a mixin diamond).

A stateless base class will usually only need a default constructor. This brings notational convenience because the constructor call can be omitted from mem-initialisers. Is he suggesting implicit initialisations? Yes, for this particular case. Let me explain why...

Virtual base classes

A virtual base class must be initialised by every class derived from it – or, more accurately, its constructor is called from the mem-initialiser list of the most-derived class (i.e., effectively it must appear in every derived class's mem-initialiser list). If the virtual base class constructor requires arguments, the constructor call must be made explicit with all the arguments supplied within every derived class's mem-initialiser list.

If virtual base classes have only default constructors, then you cannot forget to initialise them because the compiler default behaviour does the right thing!

Some guidelines

1. abstract classes should be stateless
2. virtual base classes should have default constructors (only)
3. don't explicitly initialise virtual base classes

I think I can justify those based on the observations made above. I'd like to go further but what follows is slightly harder to back up with hard experience. As a corollary to (1), I think it follows that stateless classes should have default constructors (only). Although it doesn't follow logically, I'd argue that virtual base classes should also be abstract classes and many C++ "experts" agree.

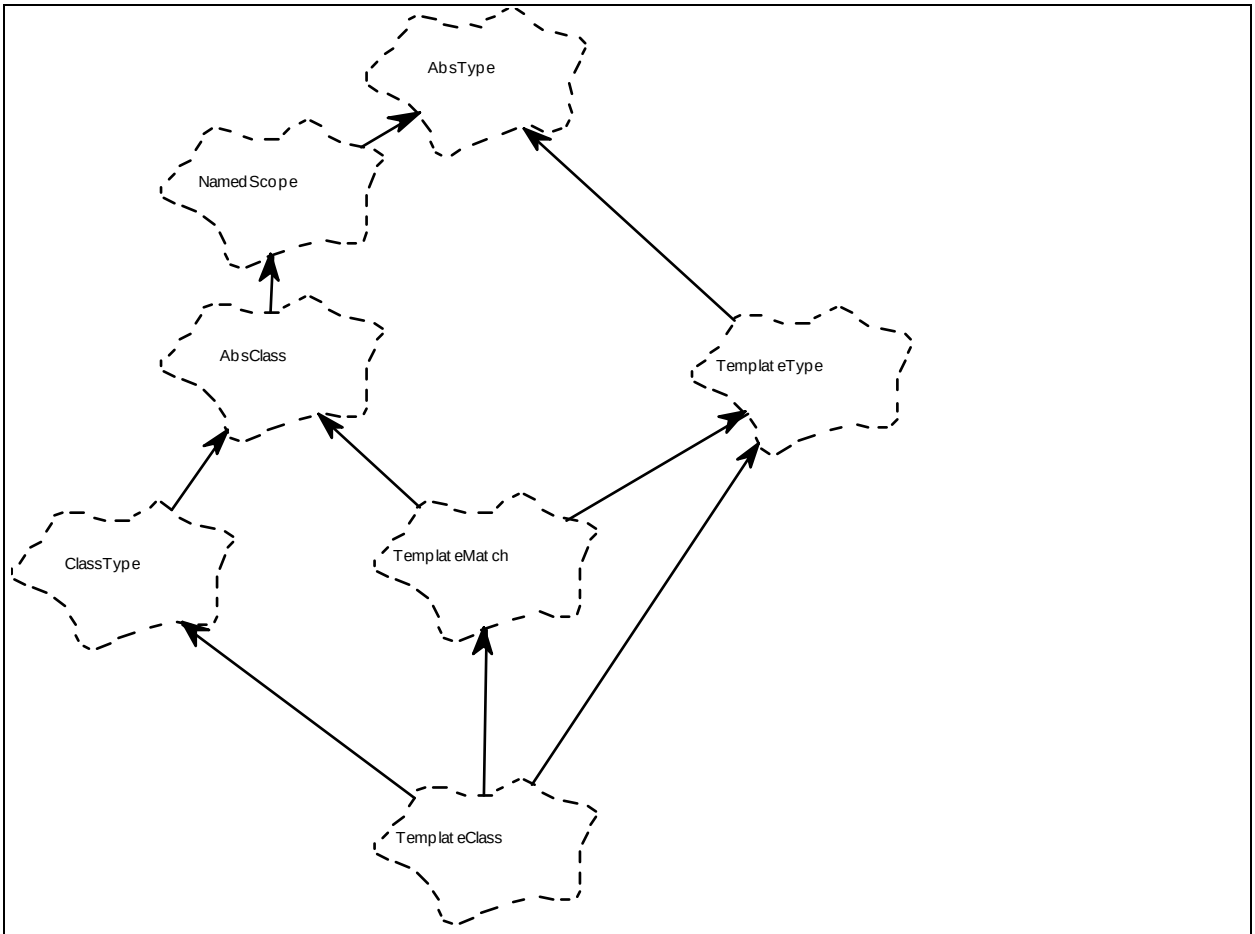

Figure 1: nested mixin diamonds

Back to my problems

That's hindsight, however, and would have been useful in easing the transition from a non-virtual inheritance hierarchy to a mixin-based hierarchy. The first step was to change inheritance from the base class *Type*, which was abstract, to be **virtual**. That was easy. However, during the design of the template argument deduction mechanism, it was decided to abstract out some aspects of the various class types' matching algorithms (mainly so that *A<T>* could be deduced regardless of whether *A* was a **class**, **struct** or **union**). This introduced another mixin diamond – see figure 1. This meant more **virtual** inheritance (from *AbsClass* and *TemplateType*) and that's where the problems really started!

At the bottom of the hierarchy, the classes representing instantiated template classes, structs and unions have three direct base classes, one of which is **virtual**, and two indirect virtual base classes. The first problem was simply changing all the relevant mem-initialiser lists and discovering that, because some of the virtual base classes had state, not all the necessary data was available in the most-derived constructor. Having sorted that out, the next problem was harder to solve: a compiler bug! The combination of virtual inheritance and multiple base classes proved too much for Sun's SPARCcompiler and it generated code that crashed during execution of the mem-initialisers. I tried the code on several other compilers which generated correct code.

I could either back out the changes and redesign things or change compilers. I chose the latter and rebuilt everything with the GNU compiler, g++ 2.6.3, which uncovered a new compiler bug!

Given a combination of virtual and non-virtual inheritance, g++ seemed to lose accessibility of **protected** base class members. The obvious solution was to make the members **public** but this bothered me so I experimented further with g++. I discovered that if *all* the inheritance was **virtual** the problem went away. Given that some people advocate **public virtual** inheritance as the true expression of the "is-a" relationship, this seemed a reasonable approach.

Of course, changing several inheritance relationships to use **virtual** meant more classes to be

initialised by the most-derived classes – classes that had non-default constructors unfortunately.

Next time

Having battled with some basic aspects of the type system, I shall turn my attention next to representing statements and parse trees.

Sean A. Corfield
Object Consultancy Services
ocs@corf.demon.co.uk

Simple classes – Part 4: Game of Life

by Roger Lever

This mini series started in *Overload 10*. The inspiration came from reading a very fascinating book last Christmas, Stephen Levy's *Artificial Life*. (Not a programming book). As a result I wrote a very simple version of Dr Conway's Game of Life, in fact that was January 95 – a year ago!

Originally, the intention was to start with something very simple and then use that as the basis for various different (and improved) versions. Each transformation would be based on a theme, for example portability, using templates, using smart pointers, exception handling, debug and code optimisation techniques. The approach (and presentation) would have been very similar to Ian Cargill's book *Programming Style*, at least that was the intention!

Inevitably, personal events have since overtaken those ideas and also the march of C++ has introduced new ideas such as the STL and Patterns. However, it would be interesting if others took up the gauntlet. Take one of these themes and use the following code as a basis for that transformation. If a number of existing and would-be authors tackled this there would surely be some very interesting reading in *Overload*! The collective effort would also significantly reduce the time required for any one individual and possibly each theme would be expanded and deepened by a succession of input? What about changing one function? One class? Everything!

Gameplan

Conway's Game of Life has three very simple rules based on the number of neighbours of each individual. The individual will:

1) Survive to the next generation with 2 or 3 neighbours
2) Be born or Emerge in the next generation with 3 neighbours
3) Die with less than 2 (isolated) or more than 3 (overpopulated)

Conway's world here is laid out like a 2-D grid with each cell location representing something possibly alive or dead. Despite initial appearances, these cells can appear to move around this world – or be alive! To enable these 'things' to stay on the world a toroidal grid is used, where the top edge wraps around to the bottom and the left edge wraps around to the right. This prevents the simple problem of a thing disappearing off the world via, for example, the lefthand side. Each iteration through the grid constitutes one generation and things live from one generation to the next based on the immediate population.

Basic design

Everything could be written with one class *RWorld* using Borland C++ and its BGI (Borland Graphical Interface) library, however, not everyone has Borland C++ (or wants to! :-). Therefore, it was important to use a hardware abstraction layer – class *Screen*. This enabled the platform specific screen handling to be located in one easily changed class. *Screen* defined an interface for *RWorld* to use, consequently *RWorld* need never be concerned about the particular implementation. This was a useful starting point.

MSDOS and BGI screen handling

Since everything that was implementation specific such as screen resolution was in one class, it seemed perfectly reasonable to leverage the BGI as long as it was here too. So, for example, the Borland specific call used to set the background colour could be exchanged for the Microsoft version instead. On that basis, *Screen* needed very little real work, simply call the BGI version.

Screen

The important methods for *Screen* are to be able to inspect and change the fore and background colours at any point(s). This effectively places a useful wrapper around this simple functionality allowing the implementation to change without affecting *RWorld*.

However, on examining the listing there are some immediate questions:

a) Why is everything hardcoded including magic numbers?

b) Why are the size of the world and screen dimensions not given?

c) Why is the accessor *Colour colourAtPoint***(int** *x***, int** *y***)** not **const**?

Some answers:

a) is not a major problem given the overall approach, portability is one of the themes and maybe good practice should be too!

b) is a problem since *Screen* should contain all the implementation specific code. In fact this code is in the *RWorld* constructor. This should perhaps be parameterised. Naturally, the values themselves should be checked as valid, particularly if they're parameters to *RWorld* or from *main***()**. Currently *main***()** simply instantiates an *RWorld* object and returns on completion.

c) find out!

RWorld

RWorld is the meat of the operation. It was highlighted in the previous section that *RWorld* should not contain the hardcoded screen dimensions. In fact *RWorld* started as an extremely simple class but grew as things progressed. A classic problem of analysis and design, not enough time spent in thinking and planning. Even what would at first sight appear to be an extremely simple program warrants some attention to this very important phase. Certainly more attention than is evident from the *Basic design* section. As a result of starting too soon *RWorld* literally contains code that really belongs elsewhere:

1) hardcoded size of the world

2) *kbhit***()** implementation detail to stop the world cycling on and on

3) glider, tetromino... distinct things added to the world

There are also organisational questions such as:

1) should the *RWorld* constructor really call *startRWorld***()**?

2) should *Status* use colour constants from the BGI?

Implementation details

The key function that starts the world going is firstly the *initialPopulation***()** which can use *randomPopulation***()** or *glider***()**, simply uncomment one or the other. These populate the world with either a random mess to see what happens, an evolutionary approach, or discreet things to see what they do. Secondly, to keep the world going, there is *startRWorld***()** which cycles through the generations determining what happens based on Conway's rules:

```
void RWorld::startRWorld(void) {
  int numNeighbours = 0;
  while (!kbhit()) {
    for (int i = YDimStart;
         i <= YDimEnd; i++) {
      for (int j = XDimStart;
           j <= XDimEnd; j++) {
        numNeighbours =
              howManyNeighbours(j, i);
        nextGeneration(j, i,
              numNeighbours);
      }
    }
    drawNextGeneration();
  }
}
```

This is a very straightforward algorithm:

- Loop until the keyboard is hit, at which point simply stop
- For each location from Top-to-Bottom, Left-to-Right work out for each cell how many of its neighbours are alive
- Based on the number of neighbours determine if it will live in the next generation
- Update (draw) the world with those born, alive, died and dead.

The application of Conway's rules is relatively simple. The important point is to account for all of the various states. Examining this function there is a question: what about the *DEAD*? It would appear from the program operation that everything is in order, however, for those examining it for the first time should it work correctly?

```
void RWorld::nextGeneration(
  int x, int y, int countOfNeighbours) {
  Colour currentState =
              World.colourAtPoint(x, y);
  switch (countOfNeighbours) {
  case 2 :
    if (currentState == ALIVE)
      World.drawPoint(x, y,
WILLSURVIVE);
    break;
  case 3 :
    if (currentState == ALIVE)
      World.drawPoint(x, y,
WILLSURVIVE);
```

```
    else
      World.drawPoint(x, y, WILLEMERGE);
    break;
  default:
    if (currentState == ALIVE)
      World.drawPoint(x, y, WILLDIE);
    break;
  }
}
```

The key function and a clear candidate for optimisation is *howManyNeighbours***()**. I started with the simplest (well it was to me!) version with a view to improving it and optimising it later:

```
int RWorld::howManyNeighbours(int x, int
y) {
       Colour neighbour1 = DEAD;
       Colour neighbour2 = DEAD;
       Colour neighbour3 = DEAD;
       Colour neighbour4 = DEAD;
       Colour neighbour5 = DEAD;
       Colour neighbour6 = DEAD;
       Colour neighbour7 = DEAD;
       Colour neighbour8 = DEAD;
       int count = 0;

       // Alive so check the vicinity
for
       // neighbours using a toroidal
grid
       // N N N neighbours (N) 1, 2 and
3
       //           YPosT
       // N X N neighbours (N) 4 and 5
       //            XPosL X XPosR
       // N N N neighbours (N) 6, 7 and
8
       //            YPosB
       // Toroidal grid wraps around
from
       // right back to left and bottom
to
       // top
       // Adjusting x and y for the left
       // and top of the grid
       int XPosL = x - 1;
       if (XPosL < XDimStart)
              XPosL = XDimEnd;
       int YPosT = y - 1;
       if (YPosT < YDimStart)
              YPosT = YDimEnd;

       // Adjusting x and y for the
right
       // and bottom of the grid
       int XPosR = x + 1;
       if (XPosR > XDimEnd)
              XPosR = XDimStart;
       int YPosB = y + 1;
       if (YPosB > YDimEnd)
              YPosB = YDimStart;

       neighbour1 =
         World.colourAtPoint(XPosL,
YPosT);
       if (neighbour1 != DEAD &&
                  neighbour1 !=
WILLEMERGE)
              count++;
       neighbour2 =
             World.colourAtPoint(x,
YPosT);
       if (neighbour2 != DEAD &&
                  neighbour2 !=
WILLEMERGE)
              count++;
       neighbour3 =
         World.colourAtPoint(XPosR,
YPosT);
       if (neighbour3 != DEAD &&
                  neighbour3 !=
WILLEMERGE)
              count++;
       neighbour4 =
             World.colourAtPoint(XPosL,
y);
       if (neighbour4 != DEAD &&
                  neighbour4 !=
WILLEMERGE)
              count++;
       neighbour5 =
             World.colourAtPoint(XPosR,
y);
       if (neighbour5 != DEAD &&
                  neighbour5 !=
WILLEMERGE)
              count++;
       neighbour6 =
         World.colourAtPoint(XPosL,
YPosB);
       if (neighbour6 != DEAD &&
                  neighbour6 !=
WILLEMERGE)
              count++;
       neighbour7 =
             World.colourAtPoint(x,
YPosB);
       if (neighbour7 != DEAD &&
                  neighbour7 !=
WILLEMERGE)
              count++;
       neighbour8 =
         World.colourAtPoint(XPosR,
YPosB);
       if (neighbour8 != DEAD &&
                  neighbour8 !=
WILLEMERGE)
              count++;
       return count;
}
```

There are at least two key approaches that immediately suggest themselves for optimisation:

1) use a data structure to cache values already worked out
2) use a data structure to cache 'knowledge' of the world
3) must be plenty of others... one question is: *nextGeneration***()** and *drawNextGeneration***()** should these be separate?

Some possible themes

- Program reorganisation and removal of magic numbers
- Templatize *Screen* for different implementations
- Optimisation of *RWorld*, in particular *howManyNeighbours***()**
- Add exception handling code
- Add new data structures using the STL

- Use patterns in determining the organisation of the classes

There are plenty of other ideas you could use. Go for it. No matter how small or large it will be useful, both to you and others. Write it up and send it in to *Overload*.

Roger N Lever
rnl16616@ggr.co.uk

The full code accompanying Roger's article will appear on a future CVu disk and on the FTP site – Ed.

Some pitfalls of class design: a case study

by Nigel Armstrong

What characterises a good utility class? How do you design it so as to make a user's life easy? Something I saw the other day set me thinking about these issues, as it suggested that some of the golden rules of class design are not as well known as they should be.

I was looking at the definition of SNMP++, a set of classes for use in developing SNMP tools. SNMP, for those who don't know it, is the Simple Network Management Protocol, which provides a standard communication mechanism for management information over an IP network. (Don't worry, you don't have to know anything about SNMP to understand this article – just as well, as I recently wrote a fifty-page paper providing just an *introduction* to it!).

Of the classes defined in SNMP++, I want to take just one as an example. This class represents an entity known as an *Object Identifier* (OID for short). The concept of an Object Identifier was developed as part of the OSI undertaking: it is intended to provide a universally unique name for each entity communicated by a protocol.

An OID is essentially just an array of integers. Each integer in the array may be of arbitrary size, but the overall array is constrained to a maximum of 128 entries.

So how would you start to design such a class? The first thing that might strike you is that because this is simply an array, much of the donkey work could be done by a template class:

```
typedef Array<int32> Oid;
```

assuming, reasonably in fact, that 32 bits is big enough for any single element, or if you want to be absolutely safe, but probably at the expense of performance:

```
typedef Array<BigNumber> Oid;
```

But neither of these is a satisfactory solution. An OID is a specialised object, not just any old array. Apart from anything else, it has a special printed representation, with the elements separated by dots, such as "1.3.6.1.4.1.1503". Any OID class worthy of the name must at least provide a constructor from a string in this format, and some way of converting an OID to such a string.

Given that we need a special class, let's look at part of the definition of the *Oid* class in SNMP++:

```
Oid::Oid();  // construct an empty oid
Oid::Oid(const char *dotted_string);
       // construct from a dotted string
Oid::Oid(const Oid &Oid);
       // copy constructor
Oid::Oid(const unsigned long* data,
        int len);
       // construct with
       // a pointer & length

char *get_printable(const unsigned int
n);    // n is how many elements
char *get_printable(const unsigned long
s,     // s is start position
             const unsigned long n);
       // n is how many elements
char *get_printable();
       // returns entire string
```

Let's start with the constructors. The default constructor it may be argued, is not strictly necessary, as it encourages a C style of coding, where objects are created before they are initialised. However it may be needed to integrate with certain template libraries, so we shall regard it as a necessary evil.

The constructor from a dotted string is obligatory, in my opinion, as this is the one that will be used to initialise constant *Oid*s:

```
const Oid nigelsOid =
              "1.3.6.1.4.1.1503.22.1";
```

Note: this actually uses the dotted string constructor and the copy constructor although the copy constructor may be elided – Ed.

The copy constructor is also obligatory. I would have liked to see some variants on this by which an *Oid* can be constructed as a sub-string of the source, but there are member functions (which I

shan't discuss here) by which the *Oid* can be edited after construction.

The only constructor that is of real concern is the one that constructs from a pointer and length. The comment is not entirely clear but suggests that the *Oid* is to be constructed from an array of integers. The question I have is where is this array supposed to have come from? How was it created? What is it for? To me such interfaces only encourage a poor quality of coding, in which developers continue to exercise all their ancient C vices, simply because they can.

Now we move on to the set of functions *get_printable*, which is really ill-conceived. All these functions return a non-**const char***: who owns this storage? Is it the *Oid*? Is it static? Is it dynamic? Or is it the client code? The fact that it isn't clear is the first problem. But even if it were clear from the documentation – and it isn't, though my guess would be that the *Oid* owns the string – there are still further problems. Let's be generous and assume that the *Oid* allocates the string and frees it on destruction.

But consider the following code:

```
Oid nigelsOid = "1.3.6.1.4.1.1503.22.1";
char *string1 =
nigelsOid.get_printable();
char *string2 =
nigelsOid.get_printable(3);
string2[0] = '\0';
char *string3 =

nigelsOid.get_printable(1,2);
```

After this code is executed what does *string1* point to? A null string? The string "3.6"? A freed memory area? It might be any of these, by my reckoning.

The documentation of SNMP++ asserts that "a user does not have to be an expert in C++ to use SNMP++". I would paraphrase this as "a user *only has to be able to guess how the classes have been implemented* to use SNMP++" !

This is the sort of class which inexperienced C++ developers feel comfortable with. It resembles the C APIs they are used to, where historically such horrors are commonplace. (You only have to look at the C and U**X standard libraries for plenty of examples).

So how should one design an *Oid* class? As a starting point, here's the equivalent part of my own *Oid* class definition:

```
Oid(const Oid &, long start = 0,
                 long length = 0);
Oid(const Oid &, const Oid &);
Oid(OidEleIterator &);
Oid(const char *);
Oid(OidEle);

friend ostream &operator<<(ostream&,
                           const Oid&);
```

Notice that there are a variety of constructors. The first two allow construction from other *Oid*s, one by substringing, the other by joining two *Oid*s together. The third uses a special class called an *OidEleIterator* – this is an abstract base class, an example of the design pattern *Iterator*. Its definition is:

```
class OidEleIterator
{
public:
       virtual OidEle First() = 0;
       virtual OidEle Next() = 0;
       virtual bool NotAtEnd() = 0;
       virtual ~OidEleIterator() {}
};
```

The idea behind the *OidEleIterator* class is that it provides a general interface for the development of efficient initialisation mechanisms, whatever the format of the originating data. *OidEleIterator* has to be sub-classed by the developer, who then provides the appropriate code to traverse the source data structure. So for example if the source data is in an array of **long**s, a developer would create a class to walk the array, returning an element of the *Oid* at each step.

Going back to the *Oid* class, the next constructor creates an *Oid* from a dotted string, and the last creates one which has only a single element.

Finally, there is a **friend** function in classic C++ style to allow the *Oid* to be printed (or if required converted to an *ostrstream*, if a true string form is needed). There might also be interfaces for conversion to and from *string*, but I didn't have that need at the time of development.

*Note: ostrstream has been deprecated in favour of ostringstream. Furthermore, the friend is unnecessary if a ostream**&** print**(***ostream**&)** member is included – the* ***operator<<*** *can simply be syntactic sugar for such a call – Ed.*

You will note that I didn't include a default constructor. I mentioned before that I thought it was probably necessary, in order to allow a user to code:

```
Array<Oid> arrayOfOids;
```

or (if you must):

```
Oid o[20];
```

but perhaps there is some other way of permitting the class to participate freely in unanticipated data structures. Can the readers of *Overload* throw any light?

In a short article it is possible only to touch on the vast subject of good class design. However, it is clear how in the design of even a very simple class there are potentially many issues to be considered.

Nigel Armstrong
nigel_armstrong@tertio.demon.co.uk

The Draft International C++ Standard

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

This issue sees a report from the March '96 meeting in Santa Cruz.

The Casting Vote
by Sean A. Corfield

At this point in the process, you should not be expecting large changes. In fact, France took the position some time ago that there should be no more extensions and the UK also wants to see stability. So now we are attempting to make only small changes. Sometimes those small changes can be very illuminating, especially when the committee have to adopt a change that makes the draft more compatible with existing practise rather than the other way around.

Stabilisation

At the Santa Cruz meeting we were supposed to be ready to ship out the second Committee Draft, triggering a second ANSI Public Review. However, discussions within WG21 on Sunday evening made it clear that too many small changes were planned for this meeting and we would have no choice but to slip the schedule by one meeting. Most committee members are now confident that we can achieve an appropriate level of stability by Stockholm (July '96) to be able to move on to the next ballot. It's possible that we might make this up later as the improvements in the draft should help it go through subsequent ballots more smoothly.

Despite the noticeable slowing of change, a remarkable number of issues were dealt with this time around.

Conversions

Two conversion-related issues were cleared up, one of which brings the draft in line with existing practice:

```
struct A {
       operator int();
       int    a;
};
struct B : A {
       operator long();
       int    b;
};
B b;
int i = b;
```

The draft used to say that *B***::operator long()** would be called here but most compilers throw all the conversion operators into the pot and pick the best match, *A***::operator int()**. The committee agreed this was more intuitive and decided to make the majority of compilers a little more conforming!

The other conversion issue concerns "slicing" where a derived class object is assigned or passed to a base class variable by value. This is often not what was intended and can lead to subtle bugs.

```
A a = b;      // from above
// b.b is silently thrown away
```

The committee decided to remove this conversion since it does not fit in with the other "polymorphic" conversions and code can easily be fixed by using references to the base class instead.

Template cleanups

A common question asked regarding templates is "when are they instantiated?". The draft doesn't say. Conceptually, instantiation occurs some time between actual compilation of a file and link time. For some compilers, that actually means during compilation. Why is this important? It gives guidance to implementors, library vendors and users about how they should be expected to deal with compilation and linking of template code and what should be expected of libraries. A proposal by myself and Dag Brück of Sweden, accepted in Santa Cruz, replaces the basic eight translation phases, inherited from C, with nine – the extra phase being instantiation

between translation (phase 7) and linking (originally phase 8). The exact wording may change as a result of other template resolutions but the draft is now quite clear that libraries can be built from code that uses templates and that instantiation is a pre-link-time activity that may require template source code to be available. Of course, implementations behave "as-is" so they may roll instantiation into either compilation or linking depending on how the implementation is constructed.

Another comment that is heard regularly amongst those who've read the draft is "what on Earth does clause 14 [templates] actually mean?". Partly from its ARM heritage and the result of many, many changes being applied since, the clause describing templates had become very difficult to follow and was imprecise at best. I have been trying to find time to rewrite the clause for over six months so my hearty thanks to Josée Lajoie, the X3J16 vice chair, who has done a tremendous job of reorganising the clause and clarifying the wording. The committee voted unanimously to adopt her rewrite and several committee members have reviewed it and made comments, most of which have been integrated already. I shall be integrating the remaining comments and completing the cleanup over the next couple of weeks.

Controversy<>

A much-debated issue finally came before the full committee. Since the days of the ARM, the intent has always been that template definitions were somehow "found" when needed for instantiation. The first C++ compiler, Cfront, worked this way as have some other compilers since. There are difficulties with this approach, however, and many compiler vendors, especially on PCs, decided to require that template definitions were available at compile-time by forcing users to include the entire definition and all supporting machinery in header files. Instantiation is then performed during, or immediately after, compilation.

Eventually, in November '94, the committee agreed to sanction this extension and made some minor changes to the draft to allow multiple copies of template bodies to appear in multiple translation units, breaking with the traditional interface / implementation separation granted for normal functions.

However, the pro-"inclusion" faction within the committee continued to lobby – they wanted the "separation" facility removed so they would not have to support it. The battle has been generally waged with hot air and Fear, Uncertainty and Doubt. In Santa Cruz, several hundred man-hours were devoted to rehashing old ground and hearing the same tired arguments over and over again. No new technical information was presented. Separation was "slow" and "hard to implement". Inclusion "leaked names" and "went against sound engineering principles". The pro-inclusion group put forward a proposal to remove support for separation, thus breaking Cfront-developed template code, and it was clear this would get a majority vote within X3J16. It was not clear that WG21 were in consensus on this and, after much political wrangling, the formal vote was in favour but with adoption deferred to Stockholm to allow further technical work to be done. That technical work can focus on the shortcomings of both models and hopefully generate a clear consensus within the international committee.

Library fashions

Transparent locales are out, bidirectional streams are in!

The locale issue has proved controversial at previous meetings and at least it is now laid to rest. If you don't know what a transparent locale is, the decision is unlikely to affect you.

The lack of bidirectional streams might be more surprising. *iostream* and *fstream* are so common in implementations that it seems incredible that the standard didn't support them. Now it does, although I noted that no national body had objected to their absence, i.e., their addition does not resolve a specific ballot comment.

Still to come

After all the minor issues dealt with this time (there were around sixty formal motions), the issues lists are shrinking to encouragingly manageable sizes and most issues should be closed out before we ship the second CD.

One of the "big" outstanding issues is name injection which just refuses to die. Every time we think we have got it licked, a new problem with our solution crops up and we go back to the drawing board.

Implementation experience with the combination of templates and namespaces is beginning to

throw up some interesting (i.e., hard) problems and member templates are sure to provide similar puzzles to be solved.

The library still needs a lot of work: some of it has never been implemented because it relies on language features that simply aren't supported by compilers yet. As more people focus on the library, more issues come to light and so the issues lists ebb and flow.

Our best guess says Stockholm will see sufficient stability to move on. Further slippage would certainly be bad for C++ and its users.

Sean A. Corfield
Object Consultancy Services
ocs@corf.demon.co.uk

C++ Techniques

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

STL is the focus of this issue: Peter Wippell tells the tale of his introduction to STL, I revisit the issue of input iterators that I raised a few issues back and also address Jiri Soukup's criticisms of pointer safety and The Harpist starts a series explaining the high-level issues behind STL. Kevlin Henney also continues his excellent series on template techniques.

I do not love thee, STL!

by Peter Wippell

Inspired by the publicity in *Overload* and elsewhere, and not wishing to be the last person on the planet without it I decided at last to try out STL. Although the experience has made me a strong supporter of it, I met several obstacles on the way – hence my title! Here is an account of what happened, in the hope that flagging my problems may clear the way a bit for other people.

Version and source

I used the version of STL, dated February 1996, which I got from the Borland Programmer's Resource Disk. This version is later than the STL library on *Overload* Disk No 5. Maybe there is a later version still, but I haven't searched for one.

Source file format

On attempting to view the source and readme files, they looked like echoes from a bygone age, owing to a shortage of carriage returns. Furthermore, their ReadOnly attributes were set. To make them suitable for use in a PC environment, I had first to clear the attributes, MS-DOS command

```
ATTRIB -R
```

and then run CRLF.EXE – a useful utility, which can be found on *CVu* disk 7.1!

Documentation

Unfortunately STL documentation comes in a PostScript file which my 9 pin dot matrix knows nothing of. To read it I had to get a the shareware application, GHOSTSCRIPT, from somewhere on Compuserve. Having figured out how GHOSTSCRIPT's FORTH style (?) syntax worked and a considerable time later I ended up with 20 out of 65 pages of an excellent manual, no printer paper and needing a new printer ribbon! Luckily, before I could work out how to tell GHOSTSCRIPT to restart the manual at page 21, I was rescued by the discovery of the same manual, in Windows help format, in the Borland libraries forum on Compuserve. This file is excellently put together and solved my documentation problem – especially so because a few mouse clicks links it into Openhelp for the BC++ IDE, enabling you to call up the help on any STL keyword directly by just clicking on that word in your source code.

Compiling an example program

The introduction to the manual contains two example programs, and compiling these, I supposed, would at least show if STL could work with my BC++ 4.5 IDE.

The first example consists of a program which takes a single integer as a command line argument, It then reads a stream of integers from *cin*, and writes, to *cout*, all members of the input stream not divisible by the argument. The program is certainly impressive since it does a great deal with a single function call.

The manual didn't say which include files are required. It turned out that STL's ALGO.H was all that was needed. However, even with this, the example would not compile immediately for two reasons:

(1) *Error: Too few arguments in template class name 'istream_iterator' in function main(int, char * *).* Apparently, either STL or Borland or both haven't yet implemented default parameters for templates. The error is corrected by giving *istream_iterator* another parameter of type, *ptrdiff_t*.

(2) *Error: Body has already been defined for function 'max(const T &, const T &)'* and a similar message for the *min* function. This is corrected, in BC4.5, by defining the Borland manifest constant *__MINMAX_DEFINED*, above "**#include** <*algo.h*>"

With these two corrections the code compiled and ran with no problems under MS-DOS and under Windows using Borland's EasyWin.

I did have one niggle with the example code, though. If no command line parameter is provided, an exception is thrown of type **char***, "usage: remove_if_divides integer\n". This doesn't work because the "throw" isn't in a **try {}** block, and there is no **catch {}** block. If the authors omitted these for brevity, it would surely have been better to write the message to *cerr* rather than throw an exception.

It is up to each implementation how it handles an uncaught exception – I expect the example code assumes that the implementation will say something like "uncaught exception X" – Ed.

A second example

The second "more realistic" example, which the authors give, is of a program, that reads a file into a vector of strings, randomly shuffles the lines and writes the result to *cout*. (Incidentally, this program is remarkably like Francis' random number program published recently in .EXE. STL uses its own random number generator which seems to have a greater range than *rand***()**).

Compiling this code became a nightmare, once I had included CSTRING.H, which contains the Borland ANSI *string* class. There were many error messages, the most common one reporting redefinition of "**operator >=(***string***,** *string***)**". The IDE itself kept reporting fatal errors, and crashing, which this compiler has never done to me before. Eventually after writing my own simple *string* class, disabling pre-compiled headers and using EasyWin, it suddenly started to compile without error. After that, to my surprise, I found that it compiled and ran even using the original ANSI *string* class and other settings. Only one warning remains:

"Warning STL\ITERATOR.H 366 Functions containing some return statements are not expanded in line."

Why did it give so much grief? The reason why, I cannot tell. I can only guess that some typo in the code, in conjunction with the new templates, put the compiler into a state which it couldn't deal with. I have noticed confusion before about which functions were defined and which not, when I ported working Borland template code from Windows to MS-DOS, so there probably are compiler problems in this area.

Note: the string class is part of the draft ISO / ANSI standard (not just ANSI!) and Borland have tracked it closely. However, the draft says that the class definition should be accessible from the header <string> – <cstring.h> is supposed to be the same as the C <string.h> header! – Ed.

The second example doesn't meet its specification

There are some oddities / errors in the way this example works:

(1) The STL description says that it shuffles the order of the "lines" in the input file. It doesn't. It shuffles the order of the words in the input. This is because the *string* extraction operator copies *scanf*'s odd behaviour.

(2) The output is all on the same line with no spaces between the strings

(3) You have to terminate input from the keyboard with a CTRL-Z which may surprise modern PC users.

(4) The random generator is not seeded. Given the same input, the program always produces the same output.

Conclusion

I intend to use STL from now on and expect it to improve my programs considerably. But I'll watch out for UNIX / MS-DOS differences.

Peter Wippell
101612.3202@compuserve.com

You can't get there from here – a closer look at input iterators

by Sean A. Corfield

Introduction

Back in *Overload* 10, I touched on the subtleties of input iterators in my *cOOmpiler* column. At the time, I wasn't sure what the committee would decide in terms of requirements on input iterators so I didn't know whether my iterator-style lexer was "valid", i.e., whether it satisfied the requirements that would be specified in the draft standard. As things turned out, it didn't. In fact, it didn't even satisfy the requirements that were in the draft at the time – I just didn't understand them!

Recap: the wrong semantics

Essentially, the semantics I had implemented were that multiple applications of the ***** operation would yield successive values and **++** was a no-op. Since input iterators are suitable for one-pass algorithms this seemed appropriate. My iterator looked roughly like this:

```
class Lexer {
public:
       Token  operator*();
       Lexer& operator++()
       { return *this; }
//...
};
Token Lexer::operator*()
{
       // get the next token from the
       // input stream and return it
}
```

Recap: the right semantics

The draft actually requires that ***** can be applied multiple times and yield the *same* value, with **++** being used to "advance" the input position. Since the first operation on an iterator is likely to be ***** this means that input iterators are required to maintain a buffered value. So a framework something like the following is required:

```
template<typename T>
class InputIt {
public:
       T&            operator*()
       { prime(); return buffer; }
       InputIt&      operator++()
       { prime(); ready = false;
         return *this; }
private:
       T      buffer;
       bool   ready;
       void   prime();
};
template<typename T>
void InputIt<T>::prime()
{
       if (!ready)
       {
              // obtain the value to
store
              // in the buffer - this
might
              // be expensive
              ready = true;
       }
}
```

This implementation delays the processing until the item is needed which might make testing for the "end" harder. An alternative is to prime the buffer at construction time and at the end of each **++** operation. This has the disadvantage that an iterator that is constructed but never used still "reads" an item.

Copying an iterator

At the Tokyo C++ meeting in November '95, Andrew Koenig gave a presentation about input iterator semantics. A controversy had arisen within the committee about what happens when you copy an input iterator. Sounds simple? What about this example:

```
Iterator i = // something
Iterator j = i;
++j;
*i;    // what happens here?
```

Although both *i* and *j* are iterators on the same source, incrementing *j* might not be expected to affect the value returned by ***i* – otherwise the sequencing semantics of *i* would be compromised by actions performed on copies. Suppose *i* was incremented as well – where do both iterators point? At the same element? At consecutive elements? The latter compromises the sequencing semantics, but the former would require an unbounded buffer to retain an image of the source being iterated over.

Clearly, copying an iterator is a special operation. Koenig explained that there were several options for restricting the semantics which generally meant "invalidating" certain operations after a copy. The most restrictive possibility is to deem a copied iterator as "invalid" – in the above example, having copied *i* to *j*, *i* becomes invalid and so ***i* has undefined behaviour. This turns out to be too restrictive – writing functions that take iterators as value arguments becomes very diffi-

cult because calling the function invalidates the original arguments in the caller.

A less restrictive approach is to say that incrementing an iterator invalidates all copies – again this invalidates *i* in the example above but pass-by-value is less likely to invalidate an iterator. The fact that most useful algorithms need to increment iterators proves how incredibly restrictive an input iterator is – most algorithms require forward iterators unless written very carefully.

A common idiom

Much of this discussion would be irrelevant were it not for a very common construct inherited from C that is intended to be preserved for iterators: ***i++**. The post-increment operator must perform special magic in order to side-step the restrictions on copied iterators – the obvious implementation is:

```
Iterator Iterator::operator++(int) {
       Iterator temp = *this;
       ++*this;
       return temp;
}
```

The iterator is copied to *temp*, the original is incremented and then *temp* is copied to the return value object. No matter which approach we take, there is no alternative to rendering *temp* invalid after the increment.

The "special magic" is to preserve the semantics of ***t** given that *t* is the result of *i**++**. This can be achieved by a proxy object, constructed from the (cached) current value referenced by the iterator:

```
IteratorProxy Iterator::operator++(int)
{
       IteratorProxy temp(**this);
                     // this->operator*()
       ++*this;
       return temp;
}
```

The proxy takes a copy of the "current" object and remembers it. The only operation applicable to a proxy is ***** which yields that remembered object. Strictly speaking the proxy should provide the same interface as the iterator for consistency but the draft does not require this, which makes writing the proxy and specifying the draft easier.

Summary

When writing algorithms, it is important to bear in mind whether the algorithm should work with an input iterator or one of the less restrictive forms. Similarly, when writing iterators it is important to consider whether they satisfy the strict requirements of an input iterator and, if not, whether they actually satisfy the different requirements of a forward iterator – they must be copyable and support multiple passes over the data set.

In my original article, I could not easily have satisfied the requirements of a forward iterator but with some extra work I could have implemented the correct semantics for input iterators.

Sean A. Corfield
Object Consultancy Services
ocs@corf.demon.co.uk

The Standard Template Library – first steps: sequence containers

by The Harpist

Everyone kept telling me what an excellent thing the STL was. My initial problem was that they kept using terms for which I had too imprecise an understanding. I suspect that the majority of programmers have similar problems. What I am going to do in this article and subsequent ones is to try to explain the STL so that you will have some map that you can use to navigate through all the apparent complexities.

I am going to assume that you know what template classes and template functions are, though it would be helpful if one of the standards experts could explain the more recent developments because I keep hearing about things that certainly are not in the ARM.

Some terms

There are several terms that you must understand if this article is going to make any sense to you.

Iterator:

This is a generalisation of the concept of a pointer. It comes in many flavours but for the purpose of this article you can think of it as a pointer with possibly added ability so that it can navigate through a container of objects.

Container:

A container is a data structure that actually contains objects. It is responsible for the lifetime of those objects. The simplest container is a plain C-style array.

Collection:

A container of (smart) pointers or references to objects. It is not responsible for the objects themselves and so an object can be in more than one collection. The simplest instance of a collection is an array of pointers.

Examples:

Container of *T*'s

```
class T;            // some type T
T arr[10];
```

This code creates an array of 10 *T*'s. If *T* is a user defined type with a default constructor that constructor will be called 10 times. When *arr* goes out of scope the 10 *T*'s will be destroyed.

Collection of *T*'s

```
class T
T* parr[10] = {0};
for (int i=0; i<10; i++) parr[i]=new T;
// code using parr
for (int i=0; i<10; i++) delete parr[i];
```

Note that the *T*'s have to be created (in this instance dynamically) and destroyed by some action of the programmer.

There are some less obvious differences between collections and containers. One is the issue of assignment. If you assign a container to another you have just destroyed the contents of the second one and copied the contents of the first. That could be quite expensive. Copying a collection has no direct implications on the collected objects (though in the case above, you might just have thrown away the only handles (pointers) to the collected objects).

If you have a choice, opt for a container as it will look after its contents for you. From now on I am going to focus on containers, because collections are containers...

Types of container

There are two main classifications of container:

Sequential:

That is a container in which objects are in some sense contiguous to each other. The concepts of 'next' and 'previous' have a well-defined meaning. An array is a sequential container. The concepts of 'first' and 'last' have a meaning for sequential containers.

Associative:

These are the containers for which there is no natural concept of ordering. Such things as maps and sets are examples of associative containers. A good example of an instance of such a container is a dictionary. You would not normally look up the 200th word in a dictionary, instead you look up a definition for a given word. There may still be some concept of ordering, but it is no longer the dominant method for accessing the container.

In the remainder of this article I will only consider sequential containers. I will deal with associative containers another time.

Performance characteristics

When you choose a container type there are a number of things that you will want to consider. For example, how easy is it to insert a new element at the beginning, or how easy is it to access an element somewhere in the middle.

STL uses two criteria for determining the answers.

Overhead:

What extra do you have to pay for extra facilities as compared with a plain array? This is really a measure of the complexity of the implementing code. For example, an array can be accessed by a plain pointer, some containers require more sophisticated iterators and hence the overhead is higher.

Performance Time:

This may be 'constant' which means the time taken for the specified action is independent of the number of objects in the container. It may be linear which means that in the worst case the time taken will be proportional to the number of objects in the container. In theory it could be other things such as quadratic (related to the square of the number of objects). In practice all simple container operations are either constant time or linear time.

STL's sequential containers

STL provides three container template classes to provide sequential containers. In general you should choose one of these in preference to a C-style array. They are more versatile and are capable of handling polymorphic types as long as you choose an appropriate iterator type.

The three template classes are: *vector*<*T*>, *deque*<*T*> (double ended queue) and *list*<*T*>. All other sequential containers provided by STL are based on one of these via an adaptor template (more about those another time).

vector<T>

This is closest to an array but it has a couple of added features. The first is that it is expandable. When the currently allocated memory is full, it obtains a larger block of storage, copies itself into the new block and then frees the old storage. Generally adding a new object at the end of a *vector* is a constant time operation (using currently unassigned memory) but there will be occasional instances when there is no spare memory and the move to new storage must be initiated. The reason for this arrangement is to ensure that objects in a *vector* are really contiguous, just as they are in a plain array.

Inserting / deleting objects other than at the end is a linear time activity because all the objects latter in the *vector* have to be moved up / down to deal with the space for the object being inserted / deleted.

All access to *vector* elements is in constant time. The overhead for a *vector* is small because it can be handled by simple C-style pointers. Even if you elect to use some form of smart pointer as the iterator type, these can still be very simple with minimal extra work (bounds checking etc.).

Where you previously used a plain C-style array, you need good reasons for not at least replacing it by a *vector*<>. You pay a small overhead for the level of indirection that allows the expansion mechanism to work.

deque<T>

The double ended queue (often just used as a single ended one, but the extra functionality comes at very little extra cost) is also expandable but uses a different expansion mechanism. Memory for objects is provided in blocks which need not be contiguous. *deque*<> maintains iterators (pointers) to each block, the first object and the last object. This means that you can add / delete extra elements in constant time to both the beginning and the end of a *deque*<> (there is a slight performance blip when a new block of storage has to be obtained, but nothing like that suffered in the equivalent case for *vector*<> because the existing objects are left where they are). Access to all elements is also constant time, but the overhead is higher because the storage in blocks has to be handled via an extra level of indirection. This is the price paid for the extra versatility and avoiding the need to move the whole lot around when available memory has been used. Inserting / deleting from the middle of *deque*<> is a linear time action because other objects have to be shuffled up or down. This process is optimised because the adjustment can be made from the closer end.

list<T>

The previous two sequential containers are really improved variations on the array theme. This one is something different. Those of you who have done a computer science course will be familiar with the various linked list data-structures. The particular one implemented in the STL is the doubly linked list. That is, each node in the list contains an object and has a pointer (iterator) to both the previous and the next node. This means that it is a little 'fatter' than the minimalist singly linked list, but you get back a little extra by way of performance. You can move both forward and backward in the list, you can add items anywhere in constant time, you can get both the first and the last object in constant time, but accessing any other object means that you must traverse the list from one of the ends.

In addition to this extra cost for accessing an internal object, there is also a considerably higher overhead because such things as iterators have to be smarter. Adding objects involves constructing nodes etc.

Unlike *vector*<> and *deque*<>, *list*<> does not come with a built in **operator[]** because that function does not readily fit the concept of a list.

Problems

STL as specified by ANSI X3J16 / ISO WG21 includes a number of excellent features that have yet to be implemented by most compiler vendors. (You should note that the version of STL distributed by Microsoft is the original Hewlett-Packard version and not either the first Committee Draft version nor the version which will be in the final C++ Standard). The completed STL will include facilities by which the user can provide an allocator (of memory) function of their choice as well as a range of other enhancements. Full implementation of STL must wait for compilers that can handle the latest refinements in templates.

However the above does not mean that you cannot get good use from the current versions. The sooner you get used to using STL containers instead of C-style arrays or hand-coded list classes the better. As you move to programming in exception handling environments you are going to

find increasing pay-back for using STL and not having to worry about resource leaks. Of course you will need to focus on the code you write, but as there will be less of it you will have more time to get it right.

The biggest problem with using STL is when things go wrong. If you think the error messages generated by current compilers are unreadable, wait till you see a few from template compilations. Two things are major causes of problems with templates.

Never use **#define**s of your own (the ones in standard headers should be OK because the writers of STL could allow for those – well, sentinels on header files will squeeze in as long as you stick to the industry standard form for these). As **#define**s have no respect for scope, the chance of your stomping on an internal STL identifier are just too high. If you do so, you are going to waste many hours tracking the cause of the problem – you will be getting error messages about template compilation resulting from something that has been changed by the pre-processor. That means the code you will be looking at will not be the code the compiler is complaining about.

The second problem is instantiating (or attempting to) a template with a type that does not meet the templates requirements. For example *deque*<> has four requirements: a public copy constructor, a public default constructor, a public destructor and a public copy assignment operator. Built-ins have these anyway, and the compiler provided versions exist for user defined classes that have not done anything to inhibit them. However, if you try to instantiate a *deque*<> for some type that is missing one of these you are going to get an error message when your code causes instantiation of the specific template code that required it.

Until you are well familiar with using STL containers, check the specific requirements of the one chosen before you use it. That will save you much grief.

Conclusion

As you can see, I am not trying to teach you how to use STL but trying to give you enough feel so that you can get started for yourself. Get using *vector*<> as soon as possible. Then become a little more discriminating and add *deque*<> and *list*<> into your range of choice.

Do you have any particular preference as to what I should tackle next? Would you like to contribute some documented code using one of the above containers? If we are to make progress we need some form of interaction. Unless you share your experiences each of us will have to learn from our own isolated mistakes (and in doing so generate our own, probably faulty, mental model of the STL). One of the things that an organisation such as ACCU is about is helping people climb the learning curve faster by sharing experiences. So get coding, documenting and sharing.

The Harpist

Using STL with pointers

by Sean A. Corfield

In *Overload* 9, Jiri Soukup criticised the Standard Template Library (STL) as being unsafe with pointers since it was a breeding ground for dangling pointers and memory leaks. The comment instantly offended me but I thought I would let a few issues go past to see if anyone else would rise to the bait. Bryan Scattergood touched on the issue in *Overload* 8 with his article on memory management and now I will take it up again.

Breeding dangling pointers

First of all, let's see why Jiri thinks STL is so dangerous. STL's containers are value-based, i.e., they expect the contained objects to obey simple construction, copy, assignment and destruction semantics. Consider the following simple example:

```
list<Shape*> shapes;
shapes.push_back(
           new Triangle(1.0, 1.0, 1.0)
);
shapes.push_back( new Square(2.3) );
shapes.push_back( new Circle(42.0) );
```

The *shapes* list now contains three pointers to different *Shape*s, but who "owns" them? Who is responsible for deleting them? If we were now to copy the list and attempt to free up an element of the old list, what would happen:

```
list<Shape*> newShapes(shapes);
                    // copy list
delete shapes.front();
                    // delete the
Triangle
shapes.pop_front();
```

When pointers are "copied", what they point at is not – this leaves the first element of *newShapes*

dangling because the object it pointed at has been deleted.

Similarly, if we pop all the elements off the lists we get memory leaks because, now, no-one owns the objects – they are no longer accessible.

A deep copy pointer

It looks as if our problem is to do with the semantics of the builtin pointer type. Perhaps we can solve the problem by encapsulating a pointer and changing its semantics? As a first cut, we can design a very simple deep copy pointer:

```
template<typename PointedAt>
class Deep {
public:
  Deep(PointedAt* x = 0)
  : p(x) { }
  Deep(const Deep& d)
  : p(new PointedAt(*d.p)) { }
  Deep& operator=(const Deep& d) {
       if (this != &d) {
             delete p;
             p = new PointedAt(*d.p);
       }
       return *this;
  }
  ~Deep() { delete p; }
  PointedAt& operator*() const
  { return *p; }
  PointedAt* operator->() const
  { return p; }
private:
  PointedAt* p;
};
```

Returning to our first example:

```
list< Deep<Shape> > shapes;
shapes.push_back(
            new Triangle(1.0, 1.0, 1.0)
);
shapes.push_back( new Square(2.3) );
shapes.push_back( new Circle(42.0) );
```

The actual arguments to the *push_back* calls are really constructed temporary *Deep*<*Shape*> objects that own the **new**'d object. The first thing that happens is that the temporary is copied into the actual list element, which duplicates the *Shape* object, and then the temporary is destroyed, which deletes the original **new**'d object. Not very efficient but it looks like it works.

If we copy the list, each element will be copied so the new list will have its own copies of the *Shape*s. When we pop elements off, the encapsulated pointer is destroyed which in turn deletes the allocated *Shape* object. So, no memory leaks either.

What is a copy?

I said it "looks like it works" – can you see what is wrong? Look closely at the *Deep* copy constructor. When it duplicates the contained object, it uses the copy constructor of that object. This may seem reasonable and is guaranteed to work for a large range of types, including builtins, but it doesn't do what we want in this case. When we push a *Triangle* onto the list the compiler constructs a *Deep*<*Shape*> from a *Triangle******. When we copy that element, the *Deep* copy constructor creates a new *Shape* constructed from the old *Triangle* object – assume the *Shape* copy constructor takes a **const** *Shape***&** argument. Oh dear! We started with a *Shape****** pointing at a *Triangle* but we end up with a *Shape****** pointing at a *Shape*! Worse, and even more likely, is when *Shape* is abstract – we cannot instantiate *Deep* for abstract types because both the copy constructor and assignment operator require *PointedAt* to be a complete, concrete type.

The problem is that our copy constructor is not polymorphic. Nor can it be, given that it is tied to the static type of its class. We need a polymorphic pseudo-copy constructor and we can achieve this by placing a restriction on the objects we use with containers. In every class *C* we must define a cloning method:

```
virtual C* clone() const
{ return new C(*this); }
```

Redefining *Deep*'s copy constructor (and assignment operator) in terms of this has the desired effect:

```
Deep(const Deep& d)
: p(d.p->clone()) { }
```

The call to *clone* is polymorphic, despatching (in our example above) to *Triangle***::***clone* which copies itself and returns a pointer to the copy. This also allows *Deep* to work with abstract classes.

Note that as written, this relies on covariant return types where a derived class method may have a return type that is not identical to that in the base class:

```
struct Base {
       virtual Base* clone() const;
};
struct Derived : Base {
       virtual Derived* clone() const;
};
```

Some compilers do not allow this yet but it works almost as well when the return types are the same:

```
struct Base {
       virtual Base* clone() const;
};
struct Derived : Base {
       virtual Base* clone() const;
```

```
};
```

Unfortunately, this is rather error-prone because it is too easy to omit an overriding definition of *clone* in a derived class.

A more complicated alternative

Having shown a simple solution above and noted two serious flaws (it is inefficient and error-prone), we need to consider a more appropriate solution. The root of both flaws is deep copy semantics. The deep copy semantics led directly to the inefficiency and the bug in our "obvious" solution led to the error-prone *clone* mechanism.

What is the problem we are really trying to solve? Memory management for container elements. Since we are working with pointers we probably want shared objects, i.e., multiple pointers pointing to a single object. We want copying to be quick and we want objects to "go away" only when no-one else is pointing at them. In other words, we need a reference-counting pointer.

Barton & Nackman give an example of this. What follows is my own variant which is slightly more efficient. The key is to maintain a count of "owners" alongside each object.

```
template<typename PointedAt>
class Ref {
public:
	Ref(PointedAt* x = 0)
	: p(x), r(new unsigned long(0)) {
}
	Ref(const Ref& x)
	: p(x.p), r(x.r) { ++*r; }
	Ref& operator=(const Ref& x);
	~Ref() { dec(); }
	PointedAt& operator*() const
	{ return *p; }
	PointedAt* operator->() const
	{ return p; }
private:
	PointedAt*   p;
	// pointer to count of other
owners
	unsigned long*      r;
	void dec();
};
template<typename PointedAt>
void Ref<PointedAt>::dec()
{
	if (*r)      // there are other
owners
	{
		--*r;
	}
	else
	{
		delete p;
		delete r;
	}
}
template<typename PointedAt>
Ref<PointedAt>&
Ref<PointedAt>::operator=(
	const Ref<PointedAt>& x)
{
	if (this != &x)
	{
		dec();
		p = x.p;
		r = x.r;
		++*r;
	}
	return *this;
}
```

Since copying a *Ref* no longer involves copying the object pointed at, we no longer have to worry about the polymorphic type of that object nor the cost of duplicating it.

Some caveats

This solution isn't perfect because you can still trip over memory leaks and dangling pointers but you have to work harder to do so:

```
Triangle t(1.0, 1.0, 1.5);
Ref<Shape> rt = &t;        // bad!
// when rt goes out of scope, t will be
// "deleted"
Ref<Shape> rc = new Circle(7.7);
return rc.operator->();    // bad!
// this hands back a pointer that
// immediately gets deleted
// when rc goes out of scope
```

Again, imposing stylistic restrictions can solve the problem:

1. only use *Ref* for heap objects,
2. use *Ref<T>* everywhere instead of ***T****.

Reworking our example, we get:

```
Triangle t(1.0, 1.0, 1.5);
Ref<Shape> rt = new Triangle(t);
              // force a copy on the heap
Ref<Shape> rc = new Circle(7.7);
return rc;    // use a return type of
              // Ref<Shape> not Shape*
```

For common pointer usage, it does make STL "safe" which was our original goal. It removes the housekeeping effort involved in keeping track of pointer ownership and it effectively means that code need never **delete** anything – a simplistic form of garbage collection. Using *Ref* everywhere instead of raw pointers has a further beneficial side-effect: it makes code exception safe.

Exception safety

How does memory management relate to exception handling, you may ask? Consider the following fragment:

```
Shape* p = new Square(1.75);
throw AnException();
```

When the exception is thrown, local variables are destroyed but the *Square* allocated above will

become a memory leak. Change this to use *Ref* and the problem goes away:

```
Ref<Shape> p = new Square(1.75);
throw AnException();
```

When *p* is destroyed, it is the last (only) reference to the allocated *Square* so the object will be deleted and the memory will be freed.

More speed!

Since one of the criteria for a good solution was efficiency, you might be interested in some further efficiency gains which can be obtained by:

1. using a pool allocator for the reference count so that space for counts can be obtained and released very quickly,
2. using a shorter type for the count if appropriate, e.g., if you know that you will not have more than 65535 owners per object you could use **unsigned short**,
3. modifying the assignment operator slightly to remove the self-assignment test by incrementing the reference count of the *rhs* before decrementing the reference count of the lhs (**this**).

These are left as an exercise for the reader. You may also like to consider what impact using *Ref* would have on your legacy code and on your coding style.

Sean A. Corfield
Object Consultancy Services
ocs@corf.demon.co.uk

/tmp/late/*
Specifying integer size
by Kevlin Henney

The exact specification of a C and C++ integer is based on the ignorance and apathy model: you shouldn't know and you shouldn't care. That's the theory. You have some minimum guarantees such as **short**s have at least one's complement 16 bit precision and **long**s at least 32, with **int**s being neither shorter nor longer than **short** or **long**. However, more than once in your programming career you will want to use an integer that is guaranteed to be a certain size, whether for portability or calculation. That you do not specify the size is one of C and C++'s strengths, as well as a weakness. It prevents you from making unnecessary and unreasonable assumptions; it can prevent you from carrying out necessary and reasonable implementation decisions.

The traditional solution is to stash a bunch of **typedef**s away in a header, and maintain it as required for portability. This homegrown solution has been rerolled countless times over the last two decades. The C9X standardisation process is under way and looks set to address the issue for C. There have been a couple of proposals on the table. I do not intend to go into them here, but if you are interested I would recommend taking a look at the following:

- Ian Cargill, "C9X: The State of Play", *ISDF Newsletter*, September 1995
- Rex Jaeschke, "Standard C: An Update", *Dr Dobb's Journal*, August 1995
- ftp://ftp.dmk.com/DMK/sc22wg14/c9x
- http://www.lysator.liu.se/c

I thought I would take a look at how we can handle this simply and elegantly in C++ without adding anything to the standard language or library.

No more, no less

Many coding guidelines caution the gentle programmer away from the use of bit fields. Much of the concern is because of the way they have been abused in the past to dodge the mask and map onto low level bit layouts. Such a mapping makes strong assumptions about alignment and a whole host of other unportable features. But bit fields have their occasional uses, and they have some handy properties. Just as with any other low level feature the place for them in C++ is hidden away inside a class. Here is a sketch of an adaptor class that allows you to supply a base type and an exact bit precision:

```
template<typename int_type,
         size_t bit_size>
class exact
{
public:
    // construction (all defaults are
OK)
    exact() {}
    exact(int_type initial)
    : value(initial) {}

public: // integer behaviour
    operator int_type() const
    { return value; }
    exact &operator++();
    exact operator++(int);
    ...

private: // state
    int_type value : bit_size;
```

```
};
```

For discussion of **typename** see my last column ("Constraining template parameter types", *Overload* 12); use **class** if your compiler does not support it. The idea here is that you specify actual types by providing a base type, which is how the type will overload, and a reduced bit size for the representation.

```
exact<char, 5> c5;  // a 5 bit char
exact<int, 12> i12; // a 12 bit int
int regular = c5 + i12;
i12 = regular;
```

The idea of the overload class is an important one. It determines what type your specified type is pretending to be:

```
void overload(char);
void overload(int);
overload(c5);  // calls overload(char)
overload(i12); // calls overload(int)
```

Clearly you would also provide all of the regular operations, such as negation, compound assignment, etc. This class is a good one to explore as you are trying to emulate a built-in type and so you must follow form — note that this includes leaving the value uninitialised by default. For instance you must provide both pre- and post-increment operators:

```
template<typename int_type,
         size_t bit_size>
exact<int_type, bit_size> &
exact<int_type, bit_size>::operator++()
{
    ++value;
    return *this;
}

template<typename int_type,
         size_t bit_size>
exact<int_type, bit_size>
exact<int_type,
bit_size>::operator++(int)
{
    return value++;
}
```

The postfix operator looks like a binary operator with the right operand missing, and the explicitly named version can be though of in these terms.

One neat feature of bit fields is that they will perform all of the hieroglyphic masking code behind your back, so that an *exact*<**unsigned, 4**> type will wrap around from 15 to 0. A word of caution: there is no guarantee that these types will fit into the smallest number of bytes, you still have a platform's alignment preferences to deal with. So *i12* above is not guaranteed to be no more than two bytes in size. The aim of this class is to deal with precision not overall alignment.

Something to watch out for is that the signedness of the plain signed types, such as **int**, is not guaranteed in a bit field. The signedness will follow the signedness of **char**. This will not affect the public interface in any way and will only affect promotions in the internal implementation. Rather than force the class client to specify **signed int** where they meant **int**, judicious casting of value to *int_type* in the cases where this would make a difference is a good idea.

This class can also be used to constrain the precision of **enum** types. Granted that enums do not support *lvalue* arithmetic operations unless you overload them to. However, unless otherwise stated, a compiler is required to instantiate only member functions that are actually used. So as long as you don't attempt to use **operator++** on an enum adaptor, the compiler won't either. For the moment you may find that your compiler does not implement these semantics — no great loss, as the intended audience of this class is regular integers.

Precision decision

It is often desirable to simply use an existing type directly rather than adapt one for such common bit widths as 8, 16 and 32. Using template specialization we can create an automatic compile time lookup for our required type. A trait, bit size in this case, allows us to look up a type. This is the inverse of looking up traits based on a type (something I will cover in a future column). By default the general template is a lifeless affair that serves only as a place holder:

```
template<size_t bit_size> struct
int_exact
{
   static const bool is_specialized
=false;
   class type {};
};
```

The **bool** constant — which must also have an uninitialised definition elsewhere in a program — allows a piece of code to query whether or not a type size maps to an actual type. If your compiler does not support **static const** initialisation in the class body, simply use an anonymous enum value. It is important that this value is a compile time constant.

The null *type* also acts a place holder for any declarations, although clearly there are no useful operations on it. As a point of style I have used **struct** rather than **class** for the template. There is nothing particularly object-oriented about this type: it does not describe anything with identity,

behaviour or state. It is all public and, in truth, it is acting like a templated **namespace** would do if such a beast existed.

I digress. This template is of no use without its specializations. For a 64 bit platform, such as Digital UNIX (née OSF/1) running on DEC AXPs, the following do the trick:

```
template<> struct int_exact<8>
{
   static const bool is_specialized =
true;
   typedef signed char type;
};
template<> struct int_exact<16>
{
   static const bool is_specialized =
true;
   typedef short type;
};
template<> struct int_exact<32>
{
   static const bool is_specialized =
true;
   typedef int type;
};
template<> struct int_exact<64>
{
   static const bool is_specialized =
true;
   typedef long type;
};
```

I am using the new specialization syntax here, but if your compiler does not support this simply drop the **template<>**. A full implementation would also provide a parallel trait lookup for unsigned types. The actual values and types supported are clearly platform specific, but we have abstracted the lookup mechanism so that the use of the types may be made portable:

```
uint_exact<8>::type octet;
uint_exact<32>::type hash_value;
cout << "64 bit ints are "
     << (int_exact<64>::is_specialized
         ? "" : "not " )
     << "supported" << endl;
```

Less is more

A more common and slightly looser requirement is to require integers with a minimum precision. The principle is the same, but now we want to perform a linear search through our traits at compile time rather than simply a straight lookup:

```
int_least<7>::type   ascii; // at least
7
                            // bit
precision
int_least<9>::type   extended;
uint_least<31>::type shift_buffer;
```

This involves creating loops and taking decisions at compile time. We can achieve this with templates using recursion to loop and the conditional operator to decide:

```
template<size_t bit_size> struct
int_least
    : int_least<(bit_size > 64)
                ? -1 : bit_size - 1> {};
```

This is pretty serious code, right? We are looping using recursive inheritance up to a maximum of 64 bit precision, beyond which we map the value to a known out-of-band value. What we are doing is searching through the range based on the induction that if we need at least *N* bits, then a type with *N + 1* bits will also do. The lookup actually looks backwards because of the way we implement it (see below).

Great, so we've got a potentially infinite inheritance loop that allegedly does something clever, but what? To bottom out the induction we need to plug in the values we know about, i.e., specialize when we hit the required values and inherit from the exact size definitions given in the previous section to give us our type and flag:

```
template<> struct int_least<-1>
    : int_exact<0>  {};
template<> struct int_least<0>
    : int_exact<8>  {};
template<> struct int_least<9>
    : int_exact<16> {};
template<> struct int_least<17>
           : int_exact<32> {};
template<> struct int_least<33>
           : int_exact<64> {};
template<> struct int_least<49>
           : int_exact<64> {};
```

We catch that out-of-band value by mapping it to a type that represents no value. All the others make sense when you look at them in terms of ranges: anything requiring at least 0 to 8 bits can use an 8 bit integer, anything requiring at least 9 to 16 bits can use 16, etc. The reason the search looked downwards was to find the first specialization that used the next precision up.

One implementation point to note here is that I added the specialization *int_least***<49>**. Why not let *int_least***<33>** carry the can? This is a minor portability issue relating to the minimum portable depth of recursive instantiation you can expect from a compiler. This magic value is 17, so I have provided two separate specialisations for 33 to 48 and 49 to 64 bits rather than one for 33 to 64.

Summary

Templates can be used for a whole lot more than simple containers and algorithms that are oft quoted as their rationale. The standard library

makes great use of adaptors that allow one type or object to masquerade and be plug compatible for another; in this case we have also used the wrapper concept to hide away a low level implementation detail. Getting the compiler to perform a type look up for you at compile time is a neat and safe way of expressing something that cannot be done with either macros or any other form of hackery.

You can take these ideas further and have an alternative to *exact* that defaults the actual integer type based on *int_least*. Or you can provide a wider implementation type if the base type would be too small for the specified number of bits. All this, as they say, is left as an exercise for the reader.

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

editor << letters;

Some more Microsoft oddities feature this month but it is not the sole topic this time!

In a response to Dave Midgeley, Peter Wippell writes:

"Borland C++ Insider" by Paul Cilwa ISBN 0-471-30338-0) Wiley 1994, 457 pp, £23.95 deals with version 4.0 concentrating on the "Experts" and much of the book takes you through an extended example using Doc/View. I enjoyed using the book as a tutorial and learnt a lot from it. However it seems to have been put together rather hastily and the racy style might not appeal to everybody. I am using BC++4.5, and I discovered that Borland had made many improvements to the "Experts" since 4.0, making some criticisms in the book out of date.

By contrast, Doc/View does not appear to work properly with persistent streams. This bug had clearly frustrated the author, but he claims to have provided a work around. Unfortunately I experienced the same problem and couldn't get rid of it!

The review will be in the next *CVu*, I hope. But of course, I'll happily answer any further questions on the book now, if you want to know more. There is also some information on Doc/View in the Borland Tutorial, and there is more basic instruction on the use of "experts" in BC++ 4.5.

Peter Wippell
101612.3202@compuserve.com

I wonder if you or any of the *Overload* readers can explain the following peculiarity in VC++4.

If I declare a **char** array as

```
char myarray[12];
```

and a function

```
myfunc(char *&);
```

and try to call

```
myfunc(myarray);
```

the compiler quite rightly complains that it cannot convert *myarray* to a non-**const char *&**.

However, if I overload *myfunc***()** with a further definition thus:

```
myfunc(char *);
```

and make the same call (actually the call I wanted to make in the first place), the compiler complains that the call is ambiguous. How can it be ambiguous, when one possibility is quite clearly illegal?

Now, VC++2 handles the call with no problem. Is this something to do with the new tighter typeing in VC++4?

Dave Midgley
100117.2522@compuserve.com

> *My first reaction is that the call should be ambiguous – both are callable within the type system with no preference for either call, but the restriction on binding a converted type to a non-**const** reference is then applied. That is why the first call is illegal but the second call is ambiguous. Many compilers will tell you that the second declaration of myfunc is illegal (without a call) because T and T& cannot be distinguished – the committee recently changed the rules so that ambiguity is only detected at the point of call.*

Francis forwarded the following bug report from Roger Woollett:

```
/* This code appears to show a bug in
 * Visual C++ version 4
 * It compiles ok but fails to link
with:
 * unresolved external symbol
 * "public: __thiscall
 * RList<double>::RLink::RLink(void)"
 *
 * works fine if RLink constructor is
 * defined inline.
 */

template <class Type>
class RList
{
public:
        RList()
                {m_pBase = new RLink;}

private:
        class RLink
        {
        public:
                RLink();
        };

        RLink   *m_pBase;
};

// could I have this syntax wrong?
template <class Type>
RList<Type>::RLink::RLink()
{
}

int main()
{
        RList<double> List;

        return 0;
}
```

Regards

Roger Woollett

It certainly looks like a bug to me, Roger – your syntax is correct.

Dear Sean,

Francis tells me that you are still short of material for publication. I must say that I find this very disappointing. Until those with less experience start asking questions, expressing opinions etc. it is hard for the experts to respond. An additional problem is that the small band of regular contributors are giving up time to write for nothing which they might well be using constructively either for earning money or for their personal enjoyment.

I know that Francis often wishes he had more time to devote to doing things for the hell of it rather than concentrating on a never ending process of learning new things that he can then write about for the benefit of others. I do not think that the membership is being fair by expecting their research to be done for them, particularly when this appears to include researching what questions need answering.

Enough of the grumbling. I attach an article (probably the first of several) in which I distil some of my experiences as I have attempted to get to grips with the Standard Template Library.

The Harpist

The Harpist's article appears in C++ Techniques in this issue.

questions->answers

Hopefully, starting a regular series, Kevlin Henney has volunteered to host the Question and Answer section – your questions can be sent direct to Kevlin but please mark them as for publication in *Overload*. I will continue to take questions but may well pass them to Kevlin.

Everyone has questions. When it comes to C++ and OO development this is certainly *very* true. What I hope to do in this occasional series is to try providing some answers, perhaps prompting further questions but hopefully casting more light than darkness. The problem faced in starting up such a column is where does the initial stream of questions come from? As I was asked a few at the AGM I will fall back on these. For future columns I hope you will put finger to keyboard with any problems or queries you have: What *exactly* are templates? Should I be using **void** pointers here? Why does the compiler say this? Is this function portable? Was it Henry the mild mannered janitor?

Getting started in OO

Programming in C++ does not imply that you are doing object-oriented programming. Neither does the use of inheritance and virtual functions in your code; just as the simple absence of **goto**s does not mean your code is structured. These are all language features and not, as it were, features of the mind.

Object-orientation is way of thinking about software organisation and there are any number of

books to help and hinder you on your way. The question "What book(s) should I read to get into OO?" is therefore a common one. The following is a book I stumbled across some time ago and would heartily recommend to all, regardless of position or experience:

> David A Taylor, *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990, ISBN 0-201-56358-4

This is a well illustrated book weighing in at only around 150 pages. Don't be misled by the title: it targets one sector of the market, but probably because focused and directed marketing is fashionable. What is meant is that it is an introductory and relatively non-technical guide.

The history of software development ideas — from chaos through structured design — is covered and the motivation and terminology of OO is introduced. It has stood the test of time well, and that it is simple and to the point — without being simplistic or curt — is a welcome change from a number of books. Once you have read this book you may feel better prepared to return to your code and go further with both your ideas and choice of reading.

Writing C++ rather than C

There is a lot to cover in teaching C++ as a vehicle for OOP. In their rush to educate, a number of books and courses inevitably fail to teach some of the mundane features of the language. That C++ is also a better C should not be forgotten: not everything is about classes and inheritance. Examples of this include

- declarations as proper statements and conditional expressions,
- support for proper compile time constants over macros,
- aggregate initialisers are not constrained to being only compile time expressions, and
- tag names are also type names.

The last one is a convenience that has passed a number of C++ programmers by, potentially to the detriment of their code's credibility. In C the tag names for **struct**, **union** and **enum** types do not name types: the tag must be prefixed with **struct**, **union** or **enum** as appropriate. To get a convenient type name, the following approach is typical:

```
typedef struct point
{
    int x, y;
}   point;

struct point a_point; /* legal C and C++
*/
point another; /* ditto */
```

This is also perfectly legal, but slightly pointless, C++. A common question, when encountering **class** and its parallels to **struct**, is given that the **class** keyword is optional in declaring an object of that type, are **struct**, **union** and **enum** also optional? The answer is "yes":

```
struct point
{
    int x, y;
};

struct point a_point; /* legal C and C++
*/
point another; // legal C++ only
```

There are a number of good uses for **typedef** in C++; compensating for a historical quirk in C is not one of them.

Empty classes

How large is an empty class? This seemingly simple question has a number of different answers depending on the context. Let us clarify what we mean by empty by first listing classes that are definitely not empty:

- Classes with any non-static data members have at least the cumulative size of those data members; possibly more where the compiler inserts padding for alignment purposes, e.g.,

```
class has_members
{
    ...
private:
    int a_member;
    string another;
};
```

- Polymorphic classes, i.e., those with at least one virtual function. Virtual functions may be introduced in the current class or by one of its bases, but the principle is the same: there must be some hidden member that in some way relates to the actual type of the object (traditionally the *vptr* referring to the *vtable*, an aggregate of function pointers).

```
class polymorph
{
public:
    virtual ~polymorph();
};
```

- An otherwise empty derived class with non-empty base classes, eg.

```
class otherwise_empty : public
has_members
{
};
```

The following class is definitely empty:

```
class empty
{
};
```

But what of its size? The bottom line is that no object can have a zero size, i.e., **sizeof(***type***)** > 0 is true for all declared objects.

What would it mean for a free object to have zero size? One practical way of looking at the result of **sizeof** is that it is the alignment adopted in an array. Any array of zero size objects would, by this definition, also have zero size. This would spell trouble for the technique commonly used to determine the number of elements in an array:

```
empty array[size];
const size_t array_size = sizeof array /
                          sizeof
*array;
```

Another consequence is all members of the array would exist at the same zero offset. In other words, allowing zero sized objects would lose us the simple guarantee that different objects have different addresses.

So free standing empty objects need non-zero size for separation and not for data. Alignment requirements vary from system to system: anywhere from one to eight bytes.

What about empty classes used as base classes? This area has been the subject of some relatively recent work by the joint ISO and ANSI committees. The result is that in such cases an implementation can ignore the standalone size of an empty class so long as it satisfies the unique address requirement. Thus the address of the inherited empty part must be different from the addresses of any other members or inherited parts.

What has this bought us? Surely we still require a separately aligned space to satisfy our requirement? As it happens in many classes there is a great deal of existing space we can reuse. Consider an **int** member: taking its address gives us a pointer to a valid object. What about halfway through the **int**? This is not a pointer to a valid whole object: what type is it? Half an **int**? There's no such thing! The upshot of this is that we can choose for the address of our content free subobject to be part way through another object.

In closing I will leave you to think about a rather interesting consequence: the size of a class may be smaller than the cumulative sizes of its base classes and data members.

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

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

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

Distribution

Mark Radford
mark@twonine.demon.co.uk

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 14* (June) should be submitted to the editor by May 31st. The intention is to claw back the delay in production of *Overload 13* over the next few issues.

FULL PAGE ADVERT GOES HERE!