ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group
Issue 21
August 1997

Editorial:
John Merrells
4 Park Mount
Harpenden
Herts
AL5 3RA
john.merrells@octel.com

m

Subscriptions:
David Hodge
31 Egerton Road
Bexhill-on-Sea
East Sussex
TN39 4EL

101633.1100@compuserve.co

Contents

Editorial

ACCU Conference

Last month's highlight was the ACCU conference in Oxford. I was pleasantly surprised by the professionalism with which it was pulled off. Compared to the usual corporate junkets this represented real value for money. There's a full review from Ray Hall in the News Section, so I won't say too much – except to report a few amusing asides.

Bjarne gave a presentation on Friday about various approaches to class design. He talked a lot about the benefits of abstract interfaces, without once mentioning COM, and a bit about the new paradigm of generic programming, without ever mentioning the STL.

He also talked about the design and evolution of C++ on Saturday morning. He used the metaphor of the craftsman's toolbox for C++: it contains a wide variety of tools, some of which are only suitable for expert use. Of course the people who don't know how to use the more advanced tools still like to have them in their own toolbox, but mainly just for show. He contrasted this with other languages which attempt to adopt a simpler approach but which are inevitably limited in the areas in which they can be used, at least until they are expanded - he raised a lot of laughter when he said that the Nutshell guide to Java is now 670 pages long.

He was most enlightening when taking questions from the floor. He managed to expound insights from even the most simple of questions. In response to the query 'How do you feel about and `const` and `mutable` relationship?' He paused to consider, and thoughtfully stated, 'I have nothing to say about `mutable`. Next question please.'

Me & Bjarne

The day before the conference I discovered that Bjarne and myself are best mates – Well actually we're just working together – Well actually my current employer, Octel, just got bought by AT&T - Well actually it was Lucent, but they used to be part of AT&T. So Bjarne and myself are *almost* best mates. Just as well I didn't stagger up to him on Friday evening and inform him of this exciting news.

John Merrells
(Almost a Bell Labs Engineer)
john.merrells@octel.com

Software Development in C++

Circles and Ellipses Revisited: Coding Techniques – 3
By Alec Ross

Around Again - This Time using Coplien's Envelope-Letter Approach

The morphing circle/ellipse problem [1-8] raises general questions on OO modelling. This note gives some further perspectives, and coding techniques. Coplien has described some relevant techniques, as pointed out by Kevlin[3]. In particular Coplien's description of an Envelope-Letter idiom offers a useful approach . Some discussion and illustrative code for this idiom is given below.

Mental Models and Classes

At a particular level in our mental model of a system we can chose to model the our types with C++ classes and supply them with implementations. As soon as we do so, we make particular design decisions (compromises) which are appropriate to some range of applications.- and not to others. For example, we can model conic sections by defining a class, objects of which we simply call "conics". If the conics were to being used in a mathematics application one might wish to be able to effect an exact symbolic differentiation of the curve. A given defined conic class might or might not support this well. For most purposes however, it will be sufficient to approximate the parameters of the conic: e.g. to represent the eccentricity by a simple built-in scalar such as a double.

A conic object might be considered to be an infinite set of points along its locus: but morphing it changes the set of points, and so even this set of points is a transient attribute of our morphable conic object.

A conic is not a physical object; but it can represent an "ideal" for a physical object - e.g. the shape of a wheel is represented by a circle.

Distortions of the wheel might be represented by distortions of the its initially circular perimeter: but in no way is the initial circle object, considered as an infinite set of points, the same as the that corresponding to the distorted one. The persistent aspects are: the wheel, which could be considered an object, and its idealised perimeter, which again might be considered an object.

Morphable Conics - Getting Round to an Implementation

In this vein we can look at a morphable conic, in a "real world", in a modelled abstract word, and as an instance of a corresponding C++ class. We will continue to call it a conic, and give it an individual existence even as it changes dimensions and conic type - even allowing it to assume values of eccentricity illegal for a mathematical conic.

Envelope-Letter Implementation

With this conceptual framework we can design a class hierarchy using the envelope-letter idiom described by Coplien[9]. Briefly, this approach involves using a base class object as a handle (the Envelope), with a member which is a pointer to a derived object. This pointer is set up at run-time (in the Envelope constructor) to point to the derived object as it is created in free store with *new*. The derived object is the Letter, and the above mechanism allows its type to be determined at run-time. The Envelope is thus a type, objects of which can be created and passed around carrying differing contents (the methods and data of the various derived classes), accessible via the base object's pointer member.

In this implementation we have a single envelope base class, and derived classes for circle, ellipse, parabola, hyperbola, and illegal. (The use of this last class allows objects to be created and morphed into and out of eccentricities illegal for a (mathematical) conic.) The envelope class contains a pointer to the appropriate derived class; changing the eccentricity changes a corresponding data member; a change of conic type involves deletion of the old letter, and creation of the appropriate new one, retaining connection via the envelope->letter pointer.

Some design decisions remain, and are of interest in generalising the approach for use with other sets of classes. For example the eccentricity member might be placed in the base class, and thus will occur in both the letter and envelope objects. These members could be simultaneously updated, or given two uses in the different components (such as that noted in the source here). Alternatively some such data need not be held in the envelope at all: it could be replicated in all the letters, but not the base; or abstracted into a further class below the base envelope, which would serve as the base for the different conic types as represented in differing "letter" classes. This latter approach might be appropriate if several such members are involved, as it could save storage-related costs of having multiple copies. One trade-off here is the loss of previous state information. A similar decision arises as to what use if any to make of the pointer to letter member, where it occurs in the letter itself. Here it is simply set to 0 in the letter; but other uses could be made.

The code below sketches an illustration of this idiom for conics. It follows Coplien's original fairly closely. It makes use of `bool`, `true` and `false`. The effects of these will need to be emulated if they are not available on the target compiler. The idiom could be developed further in various directions. For example, there could be multiple letters associated with a single envelope. A modification to replace the pointer member with an auto_ptr is straightforward. (If, as may still be the intention, the standard auto_ptr class will be defined with a non-virtual destructor, one might hesitate to use it as a base class.) Finally, the pattern, and variations of it, could readily be provided as templates.

CONIC.H

The interface for the Conic class, which offers conic objects whose eccentricity can be changed at run time. Logically there is one Conic class, which serves as a handle for implementation classes which are themselves derived from the Conic class; ie the handle Conic object contains a pointer to an object whose type can be set up and changed dynamically.

```
//Conic class, and derived classes

class Conic
{
public:
  Conic(
      double eccentricity = 0.0,
      bool isenvelope = true);
  virtual ~Conic();
  void ShowName();
  virtual void Display();
  void Setup(
      double eccentricity = 0.0);
protected:
  void Setname(const char *cp);
  double Getecc();
  Conic *Getp();
  void Setecc(
      const double e_in,
      int setenvelope = 1);
private:
  char * name;
  // initial value in base type
  double e;
  // modified/viewed value in base
  // component of derived type
  Conic *p;
  bool envelope;
};
```

The derived types are Circle, Ellipse, Parabola, and Invalid - based on the value of eccentricity called for.

```
class Circle : public Conic
{
public:
  Circle();
  void Display();
};

class Ellipse : public Conic
{
```

```
public:
  Ellipse(double ecc = 0.5);
  void Display();
};

class Parabola : public Conic
{
public:
  Parabola(double ecc = 1);
  void Display();
};

class Hyperbola : public Conic
{
public:
  Hyperbola(double ecc = 2.0);
  void Display();
};

class Invalid : public Conic
{
public:
  Invalid(double ecc = -1);
  void Display();
};
```

CONIC.CPP

Enumeration of conic types and a helper function to categorise a conic.

```
enum Conic_type {
  invalid, circle, ellipse,
  parabola, hyperbola};

Conic_type ConicType(
  const double ecc)
{
 return (
  ecc==0?circle  :
  (ecc<1 && ecc>0) ? ellipse :
  ecc==1 ? parabola  :
  ecc>1 ? hyperbola :
  invalid);
}
```

The Conic constructor takes two parameters; the eccentricity of the conic, and a flag defining if this object is the letter or the envelope. If it's the later then Setup() is called to force the construction of a letter object.

```
Conic::Conic(
    double eccentricity,
    bool isenvelope)
 : e(eccentricity),
 envelope(isenvelope),
 p(0)
{
  if (isenvelope)
  {
    Setup(eccentricity);
    name = "Base Conic: Envelope";
  }
  else
    name = "Base Conic: Letter";
}

Conic::~Conic()
{
  delete p;
}

void Conic::Setup(
    double ecc)
{
  // if we have a derived object,
  // but no change in type requested
  if ( p &&
    ConicType(ecc)==ConicType(p->e))
  {
   // simply set up new eccentricity
   p->e = eccentricity;
  }
  else
  {
  // need to set up derived object
  // (first time, or changed type)
  delete p;
  switch (ConicType(ecc))
  {
  case circle:
   p = new Circle;
   break;
  case ellipse:
   p = new Ellipse(ecc);
   break;
  case parabola:
   p = new Parabola(ecc);
   break;
  case hyperbola:
   p = new Hyperbola(ecc);
   break;
  case invalid:
   p = new Invalid(ecc);
   break;
  }
  }
}

void Conic::Setname(const char *cp)
{  name = (char *) cp;  }

// used in envelope, and in letter
inline double Conic::Getecc()
{  return e;  }

// get from envelope
Conic *Conic::Getp()
{  return p;  }

void Conic::ShowName()
{
  cout << "Name:" << name << endl;
}

void Conic::Display()
{
  cout << "\nIn Conic Display ";
  cout << "Eccentricity = "
       << e << endl;
  if ( p == 0 )
    cerr << " Pointer p == 0.\n";
  else
    p->Display();
}
```

The Circle conic class.

```
Circle::Circle()
  : Conic (0, false)
{
  Setname("Circle");
}

void Circle::Display()
```

```
{
  cout << "Logically a Circle. ";
  cout << "Ecc = " <<
                  Getecc() << endl;
}
```

The Ellipse conic class.

```
Ellipse::Ellipse(
                double eccentricity)
  : Conic(eccentricity, false)
{
  Setname("Ellipse");
}

void Ellipse::Display()
{
  cout << "Logically an Ellipse. ";
  cout << "Eccentricity = " <<
                  Getecc() << endl;
}
```

I think you can probably guess that the Parabola, Hyperbola, and Invalid conic classes are practically identical to the Ellipse conic class. Identical except for the logical type name they display.

Example client code:

This demonstrates the run-time polymorphism of the Conic class.

```
Conic C1;    // set up as default
C1.Display();

C1.Setup(0.7);  // change to ellipse
C1.Display();

C1.Setup(1);  // change to parabola
C1.Display();
```

The Conic object has changed internal type from Default, to an Ellipse, to a Parabola.

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

References

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

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

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

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

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

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

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

[8] Alec Ross, "Circles and Ellipses Revisited", Overload, Issue xx,

[9] James O. Coplien, "Advanced C++ Programming Styles and Idioms", Addison-Wesley, Reprinted with corrections 1992, especially Section 5.5, pp 133 ff (Envelope and Letter Classes), pp 148ff, (virtual constructors using globally overloaded operator new), and Section 9.2, p 311 ff, (a canonical form for the Envelope-Letter idiom).

The Draft International C++ Standard

The Casting Vote
by Sean A Corfield

London, July 1997. The circus comes to town.

The second Committee Draft ballot is closed and the votes are in. The scores on the doors were: 11 yes without comments, 6 yes with comments, 5 no (with comments), 1 abstain and 1 not voting. According to the rules, the committee has to try to address all the 'major' issues raised in the comments from the no votes so that those no votes become

yes votes. In reality, the committee had given an undertaking that comments with yes votes would also be addressed if possible.

The UK had a couple of showstopping issues that made us vote no. Resource leakage from containers was one issue and the awful mess that is `auto_ptr` was the other. We also raised other issues but they were considered less important. In London, the Library Working Group was incredibly productive and resolved about 200 issues, including exception safety and policy for container classes. The upshot of this is that the resource leakage issue has also been addressed. Unfortunately, `auto_ptr`, which was also on Sweden's comments (with their yes vote) was not touched. Even though the UK were the primary cause of adding the class in its original, simple form, and said they would be happy to remove the whole class if the copy semantics were not removed, the committee decided - for now at least - to leave it alone.

Germany raised some concerns about the specification of template template parameters being rather weak. The committee's initial response was to propose removing the language feature - a first! However, this met with a lot of opposition from several National Bodies who view the feature as potentially very useful (it hasn't been implemented yet). This issue was deferred until New Jersey in November.

To help writers of templates, return `void_expression` is now valid in template functions, allowing for code like:

```
template<typename R, typename A>
class Func
{
public:
       R f( A );
};
template<typename R, typename A>
R Func<R,A>::f( A a )
{
       return (R)g( a );
}
```

I still think passing void expressions to void parameters would help but that seems too much of a change at this point. It's a step in the right direction though.

One of my other bugbears did not go the way I would have liked. In the container classes, there is a potential ambiguity between:

```
template<typename T>
class Container
{
public:
       Container( size_t, T );
       template<typename Iterator>
       Container( Iterator, Iterator );
};
```

When `T` is an integral type, the expression `Container(100,42)` actually matches the template constructor with `Iterator` parameters instead of the '`size_t, T`' version. Prior to London, this meant that instead of constructing a 100-element `Container` full of the value 42, you'd get a compilation failure because the template constructor was a better match (with '`Iterator == int`'). There appeared to be several ways to resolve this, including adding overloads or removing some of the signatures. My favorite would have been to remove the '`size_t, T`' signatures as I believe these are confusing and error-prone.

However, the committee decided to make implementers 'do the right thing' by effectively saying that if `Iterator` turns out to be an integral type, the constructor behaves as if it was the '`size_t, T`' version. My objection to this is that it places a burden on authors of standard-like containers. I also have reservations about teachability. This resolution was the only issue I voted against this time, however, which I took as confirmation that the committee are actually converging and the document is improving. I found it rather encouraging given my harsh words about the committee in my last column.

What else was changed? Lots of small issues were dealt with which meant minor changes to overload resolution (which has become the trademark of a committee

meeting), clarification of copy optimisations and a host of other tweaks.

Next stop: Morristown, New Jersey in November where we will attempt to produce and submit the Draft International Standard, assuming that the resolution of comments in London is acceptable to the National Bodies that voted no (and changes their vote to yes). If that stage is also successful, our March meeting in France may well be somewhat celebratory as we should be able to submit the International Standard itself at that point.

Then we can settle down to deal with the torrent of Defect Reports that you all submit!

Sean A Corfield
sean@ocsltd.com.

Getting the Best
By Francis Glassborow

One common pre-occupation indulged in by programmers is deciding how to make their code smaller or faster. One consequence of this (possibly unhealthy) attitude is the degree to which suppliers of compilers try to provide optimisation. They constantly vie with each other to generate smaller faster code (in as little time as possible) even though the results are almost always bugged. That parenthetical comment is important because when I first started programming one used two distinct breeds of compiler. The first type just compiled what I wrote as closely as it could. These produced fat slow code in a reasonable time. When I was happy that I had a viable application I could then put it through an optimising compiler and go away for a long lunch break while it chewed away at my code. The result was thin, fast but it had taken its time getting there. Nowadays programmers seem to expect maximal optimisation (according to whatever specifications they switch on) in little more time than completely un-optimised code.

Quite distinct from user selected optimisations there are many optimisations provided by good compilers under what is called the 'as if' rule. Basically this says that if the program cannot determine that it has been optimised then the compiler can do it. Much of the licence given to the compilation of C code is aimed at allowing as much optimisation as reasonable. Of course one person's reasonable is another's disaster.

One of the major differences between C/C++ and Java is the attitude to how much licence shall be granted to the compiler. For example, consider:

```
x = fn(++x) + gn(++x);
```

Do not worry about what `fn` and `gn` do. In C/C++ this is very suspect code. I am not actually convinced that it exhibits undefined behaviour because there are sequence points both before and after the call of each function, however it certainly has indeterminate behaviour because you cannot know in which order the three sub-expressions are evaluated. Before the '`+`' is evaluated its operands (`fn(++x)` and `gn(++x)`) must have been evaluated. Before the assignment is evaluated (with the side effect of storing the value of the right hand side in the storage for the left hand side) the address of `x` and the value of the rhs must be determined. However note that this places no limitations on the order of evaluation of `&x`, `fn(++x)` and `gn(++x)`. If the order matters, you must unroll your code with something such as:

```
temp0 = fn(++x);
temp1 = gn(++x);
x = temp1 + temp2;
```

Even then the compiler is at liberty to mess with your code but it better come up with the answer you expect because we have now provided a strict sequence. Let me pin this down a little further for the benefit of those that are unfamiliar with the order of evaluation problem.

Suppose:

```
int fn(int val) {return val+2;}
```

```
int gn(int val) {return val*3;}
int i=0;
```

Now `temp0` should become 3 (and `x` becomes 1). Then `temp1` becomes 6 (and `x` becomes 2). Finally the last statement makes `x` become 9; But if we look at the original (assuming that the sequence points in the function calls eliminate the undefined behaviour) we still get two alternatives. If `fn(++x)` is evaluated before `gn(++x)` we get 9 but if they are evaluated in the opposite order `gn()` will return 3 and make `x` be 1, then `fn()` will return 4 (making `x` become 2). The end result will be that `x` finishes up as 7.

By the way, as a result of a question (defect report) raised by me, the C Standards Committees claim that a 'close and careful reading shows that terms must be evaluated (as if) serially and not in parallel.' The need for such a restriction is demonstrated by the above code.

The reason that C/C++ allows this unspecified order of evaluation is to permit the compiler to arrange the order of evaluation to best advantage. This can be quite advantageous, but the price is that programmers have to watch for places where the reorganisation of their code can result in different behaviour. In the context of the design of C, had this licence not been allowed the language would have been considerably less popular. Remember that one of the prime targets of C was to support porting of Unix. Operating systems need fast slim code because they are essentially large applications whose performance effects everything else.

Java has a very different set of design criteria. These result in the desire for stable, predictable code that always does the same thing regardless as to the platform on which it is running (the fact that this is not as achievable as some believe is an entirely different issue). Another feature of Java is that the target users include many people with less insight into the consequences of allowing liberties to compilers (the fact that many C/C++ programmers also lack these insights is a quality of training issue.) The result is that Java strictly defines the order of evaluation of operands as well as operators. In the above code, the address of `x` must be evaluated before the right hand side. `fn(++x)` must be evaluated before `gn(++x)`. You may consider this a good thing™, but it is not cost free. It constrains the compiler so that many potential optimisations are unavailable.

Now once we allow compilers liberties we have to consider what to do with problems such as reading a memory mapped input port. Consider:

```
char *inport= 0xFFFE;
int i;
i = *inport * 256 + *inport;
```

Now an optimising compiler is going to convert that into:

```
i = *inport * 257;
```

Definitely not what I intend, but how is the compiler to know that the effect of evaluating `*inport` is to change its value (to the next value in the input stream)? It cannot possibly know this. We want to allow the compiler to optimise our code so that we can write easily maintained code which the compiler will, none the less, compile to compact and efficient executables. What we need is a mechanism to switch off this normally desirable optimisation. That is the major purpose for which `volatile` was introduced to C. If I change my declaration of `inport` to:

```
volatile char * inport = 0xFFFE;
```

the compiler must not optimise away evaluations of `*inport`.

Almost ten years later C++ was faced with the problem of data that must always be modifiable even in the context of a `const` object. The problem here was that the compiler could do various things including marking `const` objects as ROMable. It

knows that `const` objects cannot be changed and this opens up a whole panoply of possible optimisations. Again we need a mechanism to warn the compiler off. Another keyword, `mutable`, was introduced to manage this problem.

What I am trying to emphasise is that it is part of the shared spirit of C and C++ to give compilers the maximum licence to optimise the code we write. On the other hand it is part of the spirit of Java to give the compiler as little room for change as is possible. Now let me come to the major issue that has caused vigorous debate among those responsible for C++, copy construction.

In C there is no real issue because we can define copying as a strict bitwise copy of the original. The compiler can do all sorts of things behind our backs but the code must always behave 'as if' a bitwise copy has been made every time we pass a value or initialise a variable. The compiler knows exactly what copying means and can determine when it can avoid actually doing so. For example:

```
static int treble(int param)
{ return param * 3; }

int main ()
{
  int i = 3;
  i = treble (i);
  return 0;
}
```

allows the compiler to do all kinds of things because the process of passing the argument into `treble()`, and returning a value cannot result in any odd behaviour behind the compiler's back. The point I wish to make is that the 'as if' rule allows a C compiler considerable liberty when it comes to passing values (or not actually passing them) around.

The concept of a copy constructor was to provide a mechanism whereby the programmer can handle the times when bitwise copying was either unsafe (the object includes a pointer to a dynamic resource, and so needs a deep copy) or undesirable (inefficient, and lazy copying can be used). The problem is that we can no longer optimise away copies and rely on unchanged behaviour because the programmer may have included non copying semantics in their copy constructor. For example I often instrument my copy constructors (arrange for them to output messages via `cout`, `clog`, `cerr` or whatever) so that I can track the process.

All that a compiler can deal with is syntax. When I declare and define a copy constructor I follow a specific well defined syntax by which the compiler can determine that what I am writing is a copy constructor, in other words the mechanism by which a value can be passed. Currently the compiler has a licence to assume that what is syntactically a copy constructor will also be semantically one. That means that it is allowed to use it whenever it deems it desirable to copy a value and to elide its use whenever it deems that doing so will comply with the 'as if' rule if the programmer has not 'cheated' by writing something that only appears to be a copy constructor.

This presents us with problems. First, it is not easy to lay down specific rules to determine exactly what the semantics of copying are. It is one of those things that we all believe we understand (though I suspect we all understand different things) but find nigh impossible to specify. If we could exactly specify what is meant by copying we could either define a breach of this to cause undefined behaviour or even, possibly, diagnose breaches. I am not convinced that such would be desirable even if achievable. In other words I think that there is nor merit in attempting to constrain what is syntactically copying to being semantically no more than that. Even if I did not think that, I think that there are many who would and any attempt to get consensus on this issue would be doomed from the start.

Equally well I am certain that many programmers would not wish to pay the price of constraining the compiler so that it

could not elide copy constructors to produce more efficient code. To understand the problem, consider:

```
class Mytype {
  // something
};

inline Mytype fn (Mytype m)
{ // something
  return m;
}

int main()
{
  Mytype example;
  example = fn(example);
  return 0;
}
```

How many times should the code call the `Mytype` copy constructor? Regardless of what `Mytype` might be I do not think that (after checking that it could call a copy constructor) it needs to make any actual calls. (There is an interesting secondary issue here that I have not seen raised before, and that is the potential for optimising away copy assignment. This also involves issues about copy semantics. I have no doubt that good optimising C compilers elect to optimise away assignments but I do not think that a C++ compiler can do so in the presence of a user defined `operator =()` when the parameter is a reference to the relevant class, i.e. it is a copy assignment.) As the C++ working paper currently reads I do not believe that any call is required. However by not calling the copy constructor any side effects will not happen. You might suggest that I could use a `const` reference parameter to for `fn()` to circumvent the problem, and make the return a reference. But a little tinkering with code should convince you that this does not work, I could not then make changes to the parameter in the body of the function, and I could not return it other than through a `const &`, and that is not likely to be what I want. Passing by a straight reference does not work because that will inhibit conversions to the argument passed. In addition it will allow changes to the original even if I do not desire those. You see, I may pass a value and return a value because in general I want to modify a copy and yet there will still be cases when copies are unnecessary for specific application code.

I have thought intensely about this issue and can come to only one conclusion, no matter how skilled the writer of reusable code there will be times when the compiler can determine in the context of the whole that some copies can be dispensed with. The ordinary programmer wants this, and compiler writers will provide it even if they have to turn it off to comply with some well intentioned constraint added to the standard.

I believe that this approach is the right one for C++ (though almost certainly the wrong one for Java). We need to attack the problem from an entirely different direction. There are idioms in C++ that rely on destructors being called. Some of these need support of a guarantee that a copy constructor will be called to pass by value regardless of any apparent gain from eliding the copy constructor calls.

If you are still with me, you will realise that we are back to almost exactly the problem that `volatile` solves in C/C++. We have an optimisation scenario that is almost always one that we wish to permit, indeed encourage. None the less there are times that can be determined by the class designer where such optimisation will be dangerous and result in behaviour other than that intended.

I think all must accept that optimisation by eliding or completely eliminating copies will always be with us in C++ and that most will want it that way. What I am arguing for (and only time will tell if others accept the argument) is a way for the class designer to switch off that optimisation. Because it serves no other useful purpose (note the useful) I am proposing that we deem that a copy constructor that takes a `volatile &` or a `const volatile &` parameter shall be deemed to be one that may never be optimised away. Personally I would also extend the licence and technique to copy assignment. But…

I hope that this provides you with food for thought and at the very least convinces you that you should never rely on a pass by value invoking a copy constructor.

For the record, the compiler is not supposed to optimise away a copy constructor if both the original and the copy are subsequently used. Even this requires a little more word-smithing to allow the optimisations most expect. The concept of use is tightly defined by the C++ working paper. This means that if I write:

```
Mytype m0;
Mytype m1(m0);
m0, m1;
```

Regardless as to any surrounding context the compiler MUST call the copy constructor to create a distinct object `m1` as a clone of `m0`. The fact that any halfway competent compiler optimises away the third line as doing nothing is entirely irrelevant. That line uses (in the terms of the WP) both `m0` and `m1` and so both must exist and the copy constructor must be called to create `m1` from `m0`. In other words the client programmer can, in extremis, force a call of the copy constructor. What we need is a way for the class designer to insist that his copy constructor is used.

Postscript

The above was written before the London meeting of WG21/X3J16. I decided to leave it as is and add a section explaining a little of what was decided there.

A number of avenues were explored including consideration of allowing optimisation of copy constructors based on the behaviour of the corresponding destructor. A number of horrible pathological examples persuaded those involved that they had to provide some constraints on copying that were not dependant on decisions made by class designers. The most damning code was:

```
struct X {
  int i;
  X(X &);
};

int main (){
  X x;
  int & xr=x.i;
  cout<<xr;
  return 0;
}
```

Actually this is vicious and outlaws just about all elision of copy constructors. Personally I would be happy to make such code result in unspecified results (not undefined because I think that whatever happens `x.i` should contain some readable value.) That is, if programmers insist on aliasing sub-objects then the consequences should be entirely on their own heads. The rest of the C/C++ community pays a high price in enforcing rules to make such coding practices work. However that is just my opinion and it is one that is harder to argue than simply to accept that compilers should not be allowed to optimise away copies.

What actually happened is that an attempt was made to provide a list of places where elision of copies was always acceptable. Unfortunately only two instances were agreed upon (return values and something else that slips my mind at the moment, and this is already three days late).

What many of us were concerned about was that the list did not include passing by value to inline functions. Fundamentally the problem is that it is hard to pin down exactly when this is safe, but many of us including Bjarne Stroustrup are certain that not allowing elision in such circumstances is bad news. The issue was left with an agreement to look for some formulation during the next few months but if one does not come up before the New Jersey meeting in November it seems probable that conforming compilers are going to be hamstrung. I am not sure that this does not potentially severely damage idioms that use forwarding functions (wrappers).

Francis Glassborow
francis@robinton.demon.co.uk

C++ Techniques

Safe Assignment? No Problem!
By Kevlin Henney

In the last issue [1] I examined, amongst other things, the problems of self assignment and exception safe assignment in response to an article in the previous issue [2]. A pattern based on these thoughts was presented. The pattern addressed the problem of exceptions arising from failed construction, but what of failed destruction? This was rather tantalisingly – and perhaps irritatingly – "left as an exercise for the reader to resolve". This time I will present a solution and the revised pattern.

Recap

In implementing something like the Handle/Body idiom [3] we separate the outer user object (the handle) from the object used for internal representation (the body). The body is typically dynamically allocated, implying that the default shallow copy semantics for copy construction and assignment provided by the compiler will not result in the right behaviour. A first stab at an alternative assignment operator might be something like the following:

```
type &type::operator=(const type &rhs)
{
  if(this != &rhs)
  {
    delete body;
    body = new rep_type(*rhs.body);
  }
  return *this;
}
```

But what if the constructor or `new` operator throws an exception? The handle object is left in an unstable and undestructible state: it has a pointer to an invalid, already deleted object. An attempt to remove the handle object (now in a state of confusion) will inevitably result in undefined behaviour – preventing object destruction is almost impossible, especially with auto variables and value members of other objects.

The challenge is to make this exception safe; the temptation is to put up all kinds of complex scaffolding using `try`, `catch` and `throw`. The solution is significantly simpler. Rather than using the following flow:

```
1. release existing resources
2. take a copy of rhs's resources
3. bind copy to self
```

The following code structure is implicitly safe with respect to failed allocation:

```
1. take a copy of rhs's resources
2. release existing resources
3. bind copy to self
```

This makes the following code exception safe:

```
type &type::operator=(const type &rhs)
{
  // self assignment
  // safe control flow...
  rep_type *new_body =
             new rep_type(*rhs.body);
  delete body;
  body = new_body;
  return *this;
}
```

There is the interesting side effect, as noted, that a check for self assignment is not strictly necessary. But what if the destruction results in an exception?

Repercussions

How could such an exception arise? Either the body's destructor throws an exception or the `delete` operator does. It is generally accepted that throwing an exception from a destructor is a *bad* idea, but this does not mean it will not happen or that you may have identified a particular case where you want

this capability. Also, whilst the regular operator `delete` will not throw an exception, there is nothing to stop developers providing their own allocation and deallocation operators that do so.

First we must understand why throwing exceptions from a destructor is a bad idea. Philosophically we might consider an exception a cry for help, but in the case of a destructor there is nothing we can do to help as the object ceases to be. Pragmatically throwing an exception from a destructor may terminate your program: if the destructor is being called as part of the stack unwind initiated by another exception being thrown, what would the presence of a second exception mean? In practice it means that `terminate` will be called – you can provide your own program termination using `set_terminate`, where the default is to call `abort`.

As far as your program is concerned this is, well, pretty severe. Can it be prevented? Yes:

- Throw no exceptions from destructors, which means ensuring that no exceptions are thrown by any functions it may call as well as not throwing them explicitly.

- A stronger recommendation is that the destructor should be declared with an empty throw spec, i.e. `throw()`, in which case any thrown exceptions will trigger a call to `unexpected`, whose default action is to call `terminate` but which may be customised using `set_unexpected`.

- If you still wish to propagate an exception, you can use `uncaught_exception` to filter whether an exception is thrown or not. This function returns `true` if there is currently an `uncaught_exception`.

Whichever way you look at it, it is a delicate business.

As an aside, there is an interesting idiom that allows you to extend the resource acquisition is initialisation idiom [4]. The intent of this is to grab a resource in a constructor and release it in the destructor. Such a resource may be a mutex, a file, etc. At one level this is a convenience idiom that abstracts control flow, at another it is the fundamental building block of exception safe programming.

But what if we want to take a different action in the case of failure? Consider the case of a transaction, or any kind of fallible action, that on success will be committed otherwise its changes will be rolled back. If we can assume that failure is indicated by the use of exceptions, within the destructor we can express this branching control flow:

```
transaction::~transaction()
{
  if(uncaught_exception())
      ... // rollback
  else
      ... // commit
}
```

There are few guarantees that can be delivered in the presence of exceptions thrown from destructors. The current ISO draft has definitions for what constitutes exception safety, and not throwing up in the destructor is one of the criteria.

But what can be done if it does occur? What do you do when it all goes horribly wrong? Damage limitation is the name of the game: it is probably not safe to attempt redestruction, so attempting to complete the operation with as much grace as possible, leaving things in a well defined state, seems the best approach. You can grade the severity of the problem: absolute exception safety where all is in a well defined and recoverable state, not possible here; accept resource loss but continue execution (the cut your losses approach); chaos. We'll opt for the second if we can.

Resolution

A simple control flow solution ensures that assignment is optimistic and fail safe in the event of resource deallocation failure:

```
1• alias existing resources
2• bind a copy of rhs's resources
   to self
3• release old resources via alias
```

This is disarmingly simple, preserving the previous exception safety and allowing a stable assignment to complete even if resource tidying fails:

```
type &type::operator=(const type &rhs)
{
  // self assignment
  // safe control flow...
  rep_type *old_body = body;
  body = new rep_type(*rhs.body);
  delete old_body;
  return *this;
}
```

This implies that failure to create aborts the assignment but leaves the object in its previous state, whereas failure to destroy completes the assignment but has a potential resource leak.

Although we are adopting the cut your losses approach, you may still instinctively feel the need to patch up this leak. Lets recap a moment: the object to destroy, for whatever reason, could not be destroyed and you still want to destroy it. Sounds like a tricky one. There are two ways of looking at this: one is that the problem is a real show stopper for your system, in which case the program should terminate; the other is that there is probably nothing sensible you can do with such an object, and losing it is no great loss. Either way, letting the exception propagate out of the function and losing the reference is an adequate solution.

However, you may feel that a stubborn object should be permitted to go out with some dignity and not simply be forgotten, its thrown exception the last grumble anyone hears of it. In certain cases you may also know how to deal with such beasts:

```
type &type::operator=(const type &rhs)
{
  rep_type *old_body = body;
  body = new rep_type(*rhs.body);
  try
  {
    delete old_body;
  }
  catch(...)
  {
    throw
failed_to_delete<rep_type>(old_body);
  }
  return *this;
}
```

This approach commutes the exception to a new exception that contains all the info about the indestructible object. Whoever catches it can decide to retry the deletion or take an alternative course of action. The `failed_to_delete` template class could derive from a more general deletion failure class if the catcher is not likely to be interested in the specifics. This code clutters our basic function somewhat, and a little factoring out can provide us with a simple utility function:

```
template<typename type>
void try_delete(type *ptr)
{
  try
  {
    delete ptr;
  }
  catch(...)
  {
    throw failed_to_delete<type>(ptr);
  }
}
```

This makes the assignment operator simpler, and allows us to switch strategies easily without impacting the basic flow of the function:

```
type &type::operator=(const type &rhs)
{
  rep_type *old_body = body;
  body = new rep_type(*rhs.body);
  try_delete(old_body);
  return *this;
}
```

Revision

In closing I present a modified version of the pattern:

Exception Safe Handle/Body Copy Assignment

Problem

- Ensuring copy assignment in C++ is exception safe.

Context

- A class has been implemented as handle/body pair.
- The body is copyable – type shallow or deep as appropriate.

Forces

- Any of the steps taken in performing the assignment may fail, resulting in a thrown exception. Partial completion of the steps may leave the handle in an unstable state.
- The result of assignment, successful or otherwise, must result in a stable handle.
- Self assignment must also result in a stable handle.
- After successful completion of the assignment the handle on the left hand side of the assignment must be behaviourally equivalent to the handle on the right hand side.
- Assignment, successful or otherwise, must be non-lossy, i.e. no memory leaks.

Solution

- Alias the existing body before taking the body copy.
- Perform the body copy and bind to the handle before releasing the existing body via the alias.

Resulting Context

- The existing body is not deleted before the body copy has been attempted. Therefore, a failed body copy will not result in an unstable handle.
- Failed body release may result in resource loss, but the assignment will have succeeded and have left the handle in a stable state.
- The ordering accommodates safe self assignment at the cost of a redundant copy.
- If the body copy preserves behaviour equivalence, a successful assignment will preserve it for the composite handle/body object.
- The solution can be used in conjunction with the schema for copy assignment from the Orthodox Canonical Class Form.

There is nothing left for the reader to resolve this time, but I would leave you with this thought: it is a myth that exception safety requires a maze of explicit exception handling code; carefully consideration of ordinary control flow and helper objects will often provide a simpler route.

Kevlin Henney
kevlin@acm.org

References

[1] Kevlin Henney, "Self Assignment? No Problem!", *Overload* 20.

[2] Francis Glassborow, "The Problem of Self Assignment", *Overload* 19.

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

[4] Bjarne Stroustrup, *The C++ Programming Language*, 1991, Addison-Wesley.

Make a date with C++ Independence of Declaration

By Kevlin Henney

In the last article (*Overload* 20) I covered some of the differences between C and C++ when defining traditional data types, i.e. `struct`, `enum` and `union`. Many of the differences are minor, but are sufficient to make the C++ less quirky than C in this area. On the other side of this is the dynamic aspect of dealing with types, i.e. declaring and initialising variables.

Declare anywhere

In C++ a declaration is also considered to be a statement, meaning that you can declare pretty much anywhere that a statement is acceptable.

```
cout << "Xmas of which year? ";
int year;
cin >> year;
date xmas = { 25, 12, year < 100 ? 1900
+ year : year };
```

Part of the reason for allowing this is to encourage the practice of declaring as close to the point of use as is possible, and the earliest such opportunity is when enough is known to initialise the variable. Both of these points are valuable as they discourage the separation of the variable from the point at which it becomes safe to use (i.e. when it receives a well defined value) and the place where it is used:

```
cout << "File name to store diary: ";
char name[FILENAME_MAX];
cin >> name;
FILE *out = fopen(name, "a+");
...
```

Excessive use of variables is not a practice that is criticised often enough. Many developers treat variables as an end in themselves, resulting in large declaration blocks skulking at the beginning of functions. Declaring all variables used by a function in a single place can be a hard habit to break; it is part of the very definition of languages like Pascal, FORTRAN and, to an extreme extent, COBOL. Languages like ALGOL 60, C and Ada have always permitted declarations per blocks; now C++ has taken up ALGOL 68's lead and allowed a freer, and to many peoples' minds, and more logical approach.

Absolutely anywhere

The scope of variables follows pretty much the rules you would expect, with a couple of additions and a significant change. As bona fide statements declarations can appear on their own as the body of an conditional or loop, and without the explicit scoping of a block[1].

One of the most obvious cases of bound variables is in the expression of a counting loop, normally written as a `for` loop. The counter is effectively part of the control structure. In C++ we can express this conveniently:

```
for(size_t day = 0; day < 7; ++day)
    cout << day_name[day] << endl;
```

The variable `day` is in scope over the `for` loop from its point of declaration. This means that it is not in scope outside the loop: the implication is that if you want it to be, then declare it... outside the loop! Simple as this logic may seem, there was originally no precedent for it and C++ used to take the view that the code above was equivalent to

```
size_t day = 0;
for(; day < 7; ++day)
    cout << day_name[day] << endl;
```

This has now changed, so this is an assumption to watch out for in older code. C9X will also be adopting this extension, but without the historical detour that C++ took.

[1] Note that this was not originally the case with C++, and older compilers may not support this. I must confess that outside of a certain completeness it lends to the language design, it is not often of much practical use.

Another change that C++ now supports is declarations in conditions. A whole condition can be replaced by the initialised declaration of a single variable which may be used as a logical value. This includes `bool`, `int` and pointer variables. The declared variable is in scope over the whole statement:

```
if(FILE *out = fopen(name, "a+"))
  // non-null therefore can
  // use it for I/O
else
  cerr << "Could not open " << name <<
endl; // null
```

This declaration syntax is supported in the condition of an `if`, a `switch`, a `while` and a `for` statement. It is left as an exercise for the reader to figure out why it is not supported for `do while` loops.

Jumping backlash

The support for initialisation is something that C++ emphasises above all. It is not taken as lightly as it is in C:

```
/* legal C, illegal C++ */
goto after;
{
    FILE *out = fopen(name, "a+");
after:
    ...
}
```

This is a slightly pathological piece of code (and the block is only there to allow it compile as C), but it serves to illustrate the difference between the languages: in C you can jump past an initialisation, leaving a variable in an undefined state; in C++ you cannot jump past any initialisation. Given that the use of `goto` results in something tantamount to excommunication in most circles, is there a practical point to this? Yes, consider the following illegal code:

```
switch(today)
{
case sunday: case saturday:
  cout << "Where do you want to go
today? ";
  // not legal
  char response[80] = "nowhere";
  cin >> response;
  ...
  break;
default:
  cout << "A weekday :-(" << endl;
    break;
}
```

A `switch` is just a glorified set of jumps, and even though you may have included a `break` it is important to remember that `case...break` does not define a scope: `{...}` does, and therefore jumping to `default` constitutes jumping past the initialiser for `response`. Here the intent is that if the user terminates input, the initial value of response remains as its default, i.e. `"nowhere"`. Therefore, if you want variables local to a `case` in a `switch` that code must be enclosed in block.

Dynamic initialisation

There are some constraints in C that have always bugged me as being purely historical and without rationale for either the application programmer or compiler writer. I am referring here to the requirement that aggregate initialiser lists must contain compile time constants. This is no longer a restriction in C++, and examining the following will hopefully highlight why I consider it to be the removal of an arbitrary constraint rather than a feature extension:

```
date valentine = { 14, 2, 1997 };
date propose = valentine;
int day   = propose.day + 7,
    month = propose.month,
    year  = propose.year + 1;
date hitched = { day, month, year };
```

What this serves to illustrate is that we can initialise a `struct` object from a constant aggregate initialiser list, that we can initialise one object from another non-constant `struct` object, and that we can also initialise built-in types from non-constant expressions – in this case there are many in a single declaration. This is true of both C and C++. The last line, however, is illegal in existing C (for no good reason) and legal in C++ (for obvious reasons). It is likely that this state of affairs will change in C9X.

The first example in this article also took advantage of this feature. In C++ any expression will do, and this includes complex expressions involving function calls. In these cases the order of execution is left to right.

Before time

What may perhaps be a little surprising is that globals, and `static`s by implication, may also be initialised from runtime expressions. One thing that has never been true is that program execution begins with `main`. It is the well defined entry point to the programmer's code and that has been as close to what is in practice the beginning of the program as makes no difference. In the case of anything at file scope the initialisation effectively occurs before entry into that translation unit:

```
const bool using_local_time =
getenv("USELOCALTIME");
```

Any initialiser that is not a compile time constant would be rejected by C compilers, but is permitted in C++. The initialisation takes place before execution of the first function in that translation unit, which means that it may take place before `main` is executed or on first call to a function in that module. Note that it is unwise to rely on guaranteed execution before `main`, as a couple of techniques unfortunately do; such automagical behaviour is not portable. The weak requirement allows dynamic loading of modules at runtime and module initialisation on demand.

Another issue that tends to bite is the order of initialisation: within a translation unit all file scope initialisations occur from top to bottom, but there is no guaranteed ordering between translation units. Complex initialisations that depend on other translation units are discouraged for this reason.

Summary

- Declarations are statements.
- A `for` loop variable can be declared with scope only within the loop.
- An initialised variable within a `switch` body must be enclosed in another scope, i.e. a block.
- Declarations can also be used as conditions.
- Initialisers need not be compile time constants for aggregates and non-`auto` variables. For file scope entities it is not wise to rely on a total program ordering.

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

Whiteboard

Recently I've been interviewing candidates for an engineering vacancy which we have open at the moment. My current approach, after the initial pleasantries, is to hand them a marker pen, gesture to the whiteboard, and to say, 'Tell me about the project you're currently working on.'

I generally try to navigate them towards explaining a few things; a class hierarchy they work with, or have designed, the dynamic relationship between these objects, and some aspect of the C++ implementation. So far, it's been working quite well. I can soon tell the level of their communications skills, and the depth of the understanding they have of the concepts they're trying to put across.

It's interesting how infrequently people use formalised notations for their diagramming.

There's the odd glimpse of a bit of Booch here, and a bit of OMT there. But, no sign of UML yet.

So, this introduction is just a reminder of the purpose of this new section. It's a forum for you to exchange design and implementation ideas. It doesn't matter that the idea might be flawed. It's the discussion which is important.

Well, go on, write a page about a piece of design work you completed recently. You might even find a pattern in there…

John Merrells
john.merrells@octel.com

`inline` delegation
By Francis Glassborow

I frequently hear of programmers rejecting the use of `inline`, especially implicit inline in a class interface, on the grounds that it makes the executable larger. A side effect of that can be to slow the program down if paging to virtual memory is necessary. The warning is valid but the thinking behind it is flawed. Every programmer who learnt to use macro assemblers knows that there are two critical decisions regarding code size. If a piece of code takes less space than that of a call on the underlying hardware, you always inline the code (used a macro). The second critical point is more complicated and requires a decision based on how many times the code is to be used. We can probably ignore the latter in the context of C++ but the former is certainly still valid.

A forwarding (wrapper) function does nothing except relay arguments, access etc. In its very nature code size considerations cannot influence the decision to use inline for such a function, all we are doing is wrapping one function call in another. There may be other reasons to hide forwarding functions in an implementation file but I cannot imagine what they might be.

Francis Glassborow
francis@robinto.demon.co.uk

A Finite State Machine Design
By Einar Nilsen-Nygaard

Finite state machines (FSMs) are very useful tools for keeping programming "features", like over-sized switch statements, under control.

I'm going to present an approach to a FSM design which I believe is very generic and fulfils the criteria of being "run-time polymorphic". I'll explain what I believe this means. It is the ability to change the behaviour of the classes at run time, and to me this means providing for their configuration at run time, probably by providing an interface that allows internal state to be changed.

What I'll detail here is a slightly simpler version of some state machine work I recently carried out as part of a larger project. The design is presented in Booch notation (hopefully most people will be reasonably familiar with that), and I'll try to work in an example to justify the existence of the state machine classes. Also, as a new departure for me I'll try to use templates and the STL, so please bear with me if I make some mistakes in their use!

Finally, this article is not purely design or implementation, it is more a mixture of requirement, analysis, design and implementation, so if you don't like the style please get in touch and I'll change it for my next article... if I'm asked to write another!

The Problem

The project I'm involved with just now is concerned with the management of distributed hardware devices, and as such is required to control the hardware. This is performed by either polling the hardware and sending device specific commands (e.g. over a RS232 line) or via some standard protocol such as SNMP (Simple Network Management Protocol).

It was recognised that the devices we managed quite often had some form of "state" associated with the value of certain hardware attributes. However, the attributes were often not all of the same type, so the design would have to work for multiple types.

Further, we wished to perform certain "actions" based on the current state. The scope of these actions was widened to include the following categories:

- *Before Actions* - actions performed prior to confirming the new state as entered.
- *During Actions* - actions carried out while in a particular state.
- *After Actions* - actions performed upon exiting a state.

Another important decision made was to separate state values from the external input required to drive the state machine as if it is a black box, so I decided on the standard FSM technique of an external stimulus triggering state transitions. This allowed the separation of state values and stimuli.

Some Candidate Classes

After the initial requirements and analysis, I was left with the following core class candidates:

- ***StateMachine*** - the main controlling object.
- ***State*** - an object that encapsulates the value of a state, which state is next in response to a stimulus, and what actions to perform.
- ***ActionInterface*** - an abstract class presenting an interface to allow user derived classes to implement actions to perform.

The Design Bit

So now we move onto some design of the class hierarchy. What are the relationships between the classes I identified previously, and are they up to solving the problem we looked at before? Figure 1 shows the design as it stands just now. What isn't so clear are the interfaces these classes will present to a user and how we'll manage requirements such as varying state value and stimuli types. That'll be looked at in the next section on implementation.

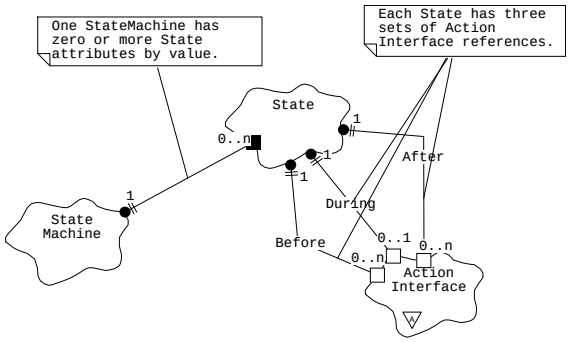

Figure 1 -- Basic FSM Design

I'll now fill out a few more of the details of the classes:

StateMachine - This is the main interface presented to the user. It will allow the user to add and remove states, attach and detach actions from states and stimulate the state machine. State values and stimuli can be of different types, but all states must have the same state value type and the same stimulus type.

State - These objects will be created by the user of the state machine and added to the state machine. They will hold three lists of pointers to the abstract class *ActionInterface* - one for actions executed before a state is entered, one for actions executed while in the state, and one for actions to be executed while leaving a state.

ActionInterface - A simple abstract class presenting to pure virtual methods, *start* and *stop*. *start* will be called once for before, during and after conditions, and *stop* will be

called once to stop during actions when a state is about to be left.

So, Let's Start Coding! (or The Implementation Section)

I've quickly moved through some of the main parts of the software lifecycle (requirements, analysis, design) and shamelessly skipped over the details, so now I'll get onto actually implementing the state machine. The first step is to get down some first cut interfaces for the three main classes identified so far.

sm.h

The declaration of the StateMachine class.

```
template<class StateValue,class Stim>
class StateMachine
{
public:
    // create the state machine with the starting state as a parameter
    StateMachine(const StateValue &initialState);
    ~StateMachine();

    // add or remove states from the state machine by value
    bool addState(const State<StateValue,Stim> &state);
    bool removeState(const State<StateValue,Stim> &state);

    bool attachAction( const StateValue &sv, ActionTime at,
                       ActionInterface<StateValue,Stim> *ai );
    bool detachAction( const StateValue &sv, ActionTime at,
                       ActionInterface<StateValue,Stim> *ai );

    bool stimulate(const Stim &stim);
private:
    // our current state
    StateValue currentStateValue;

    // storage for all the states in the machine
    typedef map< StateValue, State<StateValue,Stim>,less<StateValue> >
                                                          StateContainer;
    StateContainer stateMap;
};
```

state.h

The declaration of the class State class.

```
template<class StateValue,class Stim>
class State
{
public:
    State() { }
    State(const StateValue &sval);
    State(const State &pattern);
    ~State();

    // Assignment operator for use by STL collection class(es)
    State<StateValue,Stim> & operator=(const State<StateValue,Stim> &pattern);

    // Access to the value of this state
    const StateValue &value() const;

    // Add a transition to this state - a transition is defined as the
    // stimulus and the value of the state that the stimulus takes you to
    bool addTransition(const Stim &stim,const StateValue &nextSval);

    // Work out from the transition map what the next state should be and
    // return this in the StateValue reference parameter. If no next state,
    // method returns false
    bool getNextStateValue(const Stim &stim,StateValue &nextSval);

    // Add/remove actions to be carried out at certain times
    bool attachAction(ActionTime at,ActionInterface<StateValue,Stim> *ai);
    bool detachAction(ActionTime at,ActionInterface<StateValue,Stim> *ai);
```

```
    // For the state machine to inform the state that it's time to kick off
    // or stop appropriate actions and do one-shot actions
    void enter(StateMachine<StateValue,Stim> *sm);
    void leave(StateMachine<StateValue,Stim> *sm);

private:
    // the value represented by this state
    StateValue sval_;

    // the transition map
    typedef map<Stim,StateValue,less<Stim> > TransitionMap;
    TransitionMap tmap_;

    // the action sets -- I use sets as I don't want to have duplicate actions
    // attached to the same time
    typedef set<ActionInterface<StateValue,Stim>*,
                    less<ActionInterface<StateValue,Stim>* > > ActionContainer;

    ActionContainer before_;
    ActionContainer during_;
    ActionContainer after_;

};
```

actionif.h

The declaration of abstract class ActionInterface.

```
template<class StateValue,class Stim>
class StateMachine;

template<class StateValue,class Stim>
class ActionInterface
{
public:
    virtual bool start(StateMachine<StateValue,Stim> *sm) = 0;
    virtual bool stop(StateMachine<StateValue,Stim> *sm) = 0;
};
```

This article is the start of a series on state machines. Next month I'll be beefing up this design or altering it to provide more facilities and improved performance. Let me know what you think so far. All comments are welcome directly to me.

Einar Nilsen-Nygaard
EinarNN@atl.co.uk
einar@rhuagh.demon.co.uk

Object Counting
By John Merrells

Software Leaks!

Over time software systems often leak resources. Process resources like heap memory, and system resources like handles to kernel objects. Even when you're being really careful you can mislay bits of memory and the occasional system handle. For quick and dirty programs, which are run infrequently and for a short period of time, it doesn't generally matter a great deal. But, for systems which must exhibit long-term reliability no leakage can be tolerated. So, if even careful engineers can't write leak free software, how can we build highly reliable software?

Defensive programming must be the answer.

Memory Leakage

There's been plenty of discussion of debug memory allocators over the years. Some simple book keeping will ensure that every allocation and deallocation is accounted for. Most compilers come with a debug memory allocator, or there are commercial tools such as HeapAgent, Bounds Checker, and Purify. Along with 'memory overwrites' and 'reuse

after free' errors they will provide a leakage summary. These reports are often volumous, slow to generate, and hard to map back onto the original code.

For memory which is allocated to construct an object we should take the book keeping one step back, from the memory to the object. Rather than log the allocation, let's log the construction. Rather than the free, the destruction.

Object Leakage

Many programs contain a set of classes of which some are instantiated once, and others created millions of times. I'm extending the 80/20 rule here to mean that a program users 20% of its classes 80% of the time. To ensure that there are no orphan objects at the end of a program run, count the object constructions and destructions. The difference is the leakage.

The simplest implementation of this would be to add a static member variable and method to the suspect class. For example:

```
class Suspect
{
public:
  Suspect();
  ~Suspect();
  static void ReportLeakage();
private:
  static int m_total;
};

Suspect::Suspect()
{
  // Original stuff.
  m_total++;
}

Suspect::~Suspect()
{
  // Original stuff.
  m_total--;
}

void Suspect::ReportLeakage()
{
  cout << "Suspect: " << m_total <<
endl;
}

int Suspect::m_total = 0;
```

A single line has been added to the constructor and destructor to keep track of the total number of instances of the class in existence at any point in time. If the class had multiple constructors then each constructor would need to increment the counter.

Of course the application main function will need to call the `ReportLeakage` method at shutdown.

```
void main()
{
  DoStuff();
  Suspect::ReportLeakage();
}
```

The function which does the real work of the application might appear as follows.

```
void DoStuff()
{
  Suspect s;
  Suspect *ps= new Suspect;
}
```

In this case the corresponding `delete` for the `new` has been omitted. The resultant console output for the application will be:

```
Suspect: 1
```

With this type of object counting built into your software from day one you will instantly be aware of when some mistake has caused some resource wastage.

Performance Extension

This object counting mechanism can also be used as an algorithm efficiency metric. Keeping track of both constructions and destructions rather than just the difference allows us to see how many of each type of object was required for a particular run of the software. Our changes to the `Suspect` class would be:

```
class Suspect
{
public:
  Suspect();
  ~Suspect();
  static void ReportLeakage();
private:
  static int m_create;
  static int m_destroy;
};
```

```
Suspect::Suspect()
{
  m_create++;
}

Suspect::~Suspect()
{
  m_destroy++;
}

void Suspect::ReportLeakage()
{
 cout
  << "Suspect: Created=" << m_create
  << " Destroyed=" << m_destroy
  << " Leakage="
  << m_create-m_destroy
  << endl;
}

int Suspect::m_create = 0;
int Suspect::m_destroy = 0;
```

The output for our dodgy `DoStuff` program would be:

```
Suspect: Created=2 Destroyed=1 Leakage=1
```

Creation Number

So far this object counting method has provided us with some everyday metrics which indicates how leaky our software bucket is. But, once leaks have been identified we need a mechanism for tracking them down. We could store the creation number with each object. For example.

```
class Suspect
{
public:
  Suspect();
  ~Suspect();
  static void ReportLeakage();
private:
  int m_serial;
  static int m_create;
  static int m_destroy;
};

Suspect::Suspect()
{
  m_create++;
  m_serial= m_create;
  cout
   << "Suspect Constructor "
   << m_serial << endl;
}

Suspect::~Suspect()
{
  m_destroy++;
  cout
    << "Suspect Destructor "
    << m_serial << endl;
}
```

The output for our simple DoStuff program would be:

```
Suspect Constructor 1
Suspect Constructor 2
Suspect Destructor 1
Suspect: Created=2 Destroy=1 Leakage=1
```

In a simple case such as this it is simple to identify the errant piece of code. In a more complex system we'll need something more sophisticated. Keeping a list of pointers to all the objects currently in existence would reduce the program output and the effort required to match up all the serial numbers.

```
class Suspect
{
public:
  Suspect();
  ~Suspect();
  static void ReportLeakage();
private:
  int m_serial;
  static int m_create;
  static int m_destroy;
  static list<Suspect*> m_orphans;
};

Suspect::Suspect()
{
  m_create++;
  m_serial= m_create;
  m_orphans.insert(
            m_orphans.begin(),this);
}

Suspect::~Suspect()
{
  m_destroy++;
  m_orphans.remove(this);
}

void Suspect::ReportLeakage()
{
  cout
  << "Suspect: Created=" << m_create
  << " Destroy=" << m_destroy
  << " Leakage="
  << m_create-m_destroy
  << endl;
  list<Suspect*>::iterator i;
  cout << "Suspect Remaining= ";
  for (
    i =  m_orphans.begin();
    i != m_orphans.end();
    ++i)
    cout << (*i)->m_serial << " ";
    cout << endl;
}

int Suspect::m_create = 0;
int Suspect::m_destroy = 0;
list<Suspect*> Suspect::m_orphans;
```

Our improved summary becomes:

```
Suspect: Created=2 Destroy=1 Leakage=1
Suspect Remaining= 2
```

At this point I'll stop refining, but I'll seed a couple of thoughts in your mind.

- How might we find out more about the leaked objects?
- How might we extract this object counting method into a mix-in class?
- How could system resource handles be similarly tracked?

John Merrells
john.merrells@octel.com

Rational Values by The Harpist

Introduction

This article, suggested by Francis, introduces the design of a value based class. I hope that Overload 22 will include some comments from you experts, along with my attempt to implement the class.

The problem is to design and implement a rational number class. For those that dozed in the back of their maths classes (actually in my experience the best place to doze was the front, teachers looked straight over you and worried about the level of attention they were getting from those further back) a rational number is one that can be expressed as the ratio of two integers. In other words these are the much dreaded fractions that send most human beings into screaming fits. If you doubt this, try asking ten people to help you with a small piece of arithmetic with fractions, you will be very lucky if even one of them does not immediately discover a reason that they need to be elsewhere.

A rational number can be represented by an ordered pair of integers, one called the numerator (conventionally written first, or on the top) the other being the denominator (second or below). Actually we can be more general than this and merely require that the denominator and numerator themselves be rational.

One advantage of rational numbers as compared with floating point ones (so called real numbers which can only be approximated in a computer) is that there is no approximation involved when using them for computer arithmetic, as long as the two integers remain within the range of available values. On the other hand we must take care to reduce them to a canonical form (unique and agreed standard representation) this involves a process of reducing the two integers by dividing by their highest common factor. This is a simple process as there is an excellent algorithm for finding the highest common factor that was well known to the ancient Greeks, but more of that in good time.

A Skeleton Design

First we need an integer type. At some stage we may want to go to some form of big integer with unlimited range (well only limited by machine resources) but we better choose something simpler to start with while we develop the class itself.

This is the first place that inexperienced programmers overly constrain their solutions. Consider the following:

```
class Rational {
public:
  typedef unsigned int integer_t;
private:
  integer_t numerator, denominator;
  bool negative;
  // to come
};
```

Note my choice of `unsigned int`. There is really nothing to be gained and much to be lost by confusing the issue with signed values for the numerator and denominator. Instead the sign is stored separately and a `bool` seems ideally suited to this purpose. The public `typedef` publicly documents

the type associated with the numerator and denominator. Dirctly using the underlying type will damage the portability of your code. It is worth taking note of such uses of `typedef` and using the defined alias, even at the cost of some extra typing.

From time to time you might decide that you wanted a floating point approximation to the value of a `Rational`. This requires a floating point division which is still one of the most expensive arithmetic operations, even on modern machines (and remembering that we may be using a user integer type, is likely to remain potentially expensive). We should not want to carry out this operation unless needed. A simple optimisation should be used to avoid recalculating it if it has already done.

That suggests two more items of data:

```
long double fp_value;
bool converted;
```

At this stage I am not sure whether the code to do the conversion will be used other than to provide a value for an appropriate `get` function. It does not matter because I can implement it directly initially, and provide a `private` utility function later if the code needs to be extracted for reuse. Now why `private` and not `protected`? Well there will be no `protected` interface for `Rational`. This is a pure value class, it is not intended as a base class and if you need to use it that way either I got it wrong or you did. Try to think about this because if it is not immediately clear, you still do not have a firm grasp of the difference between values and objects. (One way to look at the distinction is by considering what will be meant by saying that two identifiers compare equal (i.e `ident1==ident2` is true). For objects this basically means they are the same object (you compare addresses), but for values you have to compare the representations.)

Why did I choose a `long double` to store the converted value? Well that ensures that I get the greatest possible number of significant figures. If I need less in the application, that is OK because conversion to `double` (or even `float`) at the application end will handle that. As the designer of `Rational` I want to meet the needs of the largest possible group of clients.

There is one small problem with this; I have considerably increased the storage required for a `Rational`. That is a typical design decision and much depends on how often I think you will want to use a floating point value. If I thought the class would be rarely used I might remove the storage and rely on direct calculation every time. Actually, as the storage is `private`, I could remove it without changing the public interface/behaviour of the type. Perhaps a good example of the advantages of `private` data.

However, if I choose to retain the storage there is one extra refinement that I need to think about. What happens if the user creates a `const Rational` and then asks for its floating point value? Either the constructors need to initialise `fp_value` in all cases (not desirable because of the potential computational overhead) or `fp_value` and the related `bool` flag must be changeable even in a `const` instance. Fortunately C++ now provides a mechanism to support this need. We need to use the qualifier `mutable`. Thus far our class is:

```
class Rational {
public:
  typedef unsigned int integer_t;
private:
  integer_t numerator, denominator;
  bool negative;
  mutable long double fp_value;
  mutable bool converted;
public:
  long double get_value() const;
  // to come
};
```

Now we have one utility function whose purpose is to reduce the numerator and denominator to their lowest terms. The only question is whether this should be automatic or determined by some criterion. Again this will be an implementation detail. If it is

automatic (in other words will be performed every time there is a change to the primary data—numerator and denominator) then we do not need to track if it has been done. If it is based on some other criterion we will need a `bool` value to track whether the current state is the canonical (fully simplified one) or not. Let us keep it simple for now and specify that all functions that change primary data will call the `private` utility function `simplify()` to reduce the primary data to its simplest form. Note that all such functions must also set the value of `converted` to `false`.

Now, let us consider what we need by way of constructors. We will certainly need a default constructor (because we are sure to want arrays of `Rational`). We will also want to be able to convert integers to Rational, and to be able to construct a `Rational` from two integer values. Actually we can package this up into a single constructor:

```
Rational(
     integer_t numer=0,
     integer_t denom=1);
```

So the default `Rational` is 0/1 and integer n converts to n/1. That seems entirely satisfactory. Do we need any other constructors? Yes, we must have one to convert floating point types to `Rational` ones. I think that an implementation of:

```
Rational(long double);
```

will be enough. Note that in context this particular parameter hardly needs a name in the prototype. I am not covering implementation this time but you should think carefully about this one because it will almost certainly be possible to pass a value that cannot be represented by a `Rational` because it is outside the achievable range (this might not be the case if `integer_t` was a user type with unlimited range). How should we deal with such a case? Note that we could use 0 for a `denominator` to represent effective infinity. There are some advantages to such a technique, though if we use it, any use of `fp_value` will need special treatment. What do you think? Try to weigh up the merits of different solutions.

What about a copy constructor? Do we need one? And copy assignment? I do not think either of these needs user definition because as far as I can see the compiler generated ones will be correct and efficient for any reasonable implementation. We can always revisit this decision if necessary.

The same applies to the destructor. Now look back at the constructors and ask yourself what we have missed and how we might fix it. Yes, there is a problem and it does need fixing, but there are alternatives and as class designers we need to weigh up the merits and come to a solution.

What else do we need? I guess that many users will want to inspect both `denominator` and `numerator`, but should we also allow them to change them directly? Think carefully about this because we should not just provide functions because we think of them. For what it is worth, I do not think that we need `set` functions for the primary data, but see if you can spot why I think we can do without them.

We certainly need functions for basic arithmetic. Remember that I am of the school that abhors use of `friend` unless you can justify it from some perspective other than that you think it convenient. The following member `operator` functions should provide most of what we need:

```
Rational operator +=(const Rational &);
Rational operator -=(const Rational &);
Rational operator *=(const Rational &);
Rational operator /=(const Rational &);
```

Any others? Well you might consider the possible increase in efficiency that could result by providing functions such as:

```
Rational operator *=(long int);
```

Note however that these do not provide extra behaviour only the possibility of alternative implementation when desirable. It is not

often that you can add or remove functions without adding or removing behaviour. Also note that we must be careful not to overdo this and finish up with perfectly reasonable code becoming ambiguous. However there are some conversions that we may want to suppress or replace.

If you consider implementation details you will need to watch out for code that should be factored out. Things like lowest common denominator will be needed by more than one member function.

Do we need a function to print out (or dispatch to an output stream) the primary data? If so, should it be a member function or a global one? What reasons do you have for your choice?

One function we certainly need is a comparison function. Given that we can easily implement the various logical operators.

What else? Think carefully because there are still several bits that I have left out.

Conclusion

The above is the starting point for the complete design of a `Rational` class. I hope you will spend a little time completing it. This should include an explanation of any choices you have made. Such explanation is much more valuable than the silly comments that some programmers litter their code with. It is the thing that helps others to see why you did things. A design is like a blue print and so should contain all the information needed for someone to check the design as well as for someone to implement it.

When you have an extension added to your house, the blue prints are required by those validating the proposal against local building regulations. They are also needed by the various craftsmen (bricklayers, plumbers, electricians etc.) who must implement the plans. You will not get authority to go ahead until all major features have been designed and documented.

Just the same should apply to provision of a new class in C++. Of course design refines analysis and sometimes you might have to revisit that first stage, and implementation may sometimes require a design modification but those should be infrequent. In the case of a well documented design revisiting the design should be easy if necessary because the principles are already clearly stated. You really should not be cutting implementation code till your design is pretty solid.

Well it is over to you.

The Harpist

I have a copy of the C++ Report CD (1991-95) for the best documented complete design for a `Rational` class. You need not follow the same plan as the Harpist. I will leave it to the editorial board of Overload to determine the winner. Francis.

News, Views, & Reviews

The C&C++ European Developers Forum

Conferences can be tricky to evaluate depending on your balance of expectations and outcomes. My expectations were grouped around the promises in the brochure about learning more about C/C++ and meeting like-minded people. Organisation by Parkway Research was good, Oxford Town Hall easy to get to and commodious, and the weather was wonderful (OK that's not due to organisation but it was a real bonus). The principal bug was that the untested acoustics of the main hall turned out to be appalling.

Day 1 (Friday 18 July) consisted of a variety of parallel sessions and was attended by about 200 people. Session 1 had six groups with sign-up varying from 72 (STL) down to 5 (Lotus Domino). Session 2 was more skewed, from 7 for Perl5 to 100 for Patterns. The final session had 168 mainstreaming to hear Bjarne Stroustrop and only 16 for Delphi for C/C++ programmers. I have no idea what the out-turn was but there was a shortage of seats for the patterns session, so some of the smaller sessions may have been somewhat intimate. Day 2 was all in the main hall, with what appeared to be a slightly larger take-up than on Friday. I hope these numbers will convince the powers that be (aka Francis?) that this sort of thing is well worth mounting.

I was with the majority in all 3 choices on Friday, and look forward to feedback from the other sessions. The STL session was given by Leen Ammeral (Hogeschool van Utrecht), Patterns and Implementations by Kevlin Henney (QA Training) and on Saturday we had Dan Saks (Saks & Associates) on Const, Bjarne again, Tom Plum (Plum Hall Inc.) on standards and compiler testing, and P J (Bill) Plauger on embedded C++.

Given that the major speakers had come hot-foot from the WG21 standards meeting, there was much mention of the Final Committee Draft which had resolved practically all the issues about which we have read in recent years. This is good news since we might now expect an ISO standard in mid 98. It is also relevant to the Forum because it was the newer features which dominated presentation and discussion. Since time and space prevent a blow-by-blow account of individual sessions the following is a more generalised account of my impressions. The standards issues, coupled with my Friday choices put more emphasis on generalities than I had expected, though this is by no means a criticism.

Probably the nitty-grittiest session was Dan Saks who almost frightened the pants off me as I realised how counter-intuitive was my appreciation of `const`. For the most part this did not rely on new features, except for `mutable`, which he deployed as part of a campaign to get constants out of headers and to stop casting the constness away. This had many detailed examples and it would be good to see some of these in Overload or CVu some day.

To be fully buzzword-compliant we now need to know about Patterns. When I first came across this (in 1989, would you believe) there was an emerging idea that a book about buildings published in 1977 by Christopher Alexander might have some relevance to software construction. Now there are books and articles all over the place and Alexander's book has become a best-seller. Kevlin showed a range of patterns with code fragments implementing them. This went down well, though I would have preferred fewer examples more fully detailed. Since the notes were substantially worked up perhaps we can persuade him to make some of them available here.

Bjarne Stroustrop has an engagingly informal presentation style but I would guess that it is thoroughly rehearsed. In fact, much of what he said is available in the third edition of *The C++ Programming Language* which sold like hot cakes after his Friday presentation (representing a large proportion of the 100 copies available in the UK at that time - run and get one if you can!). Logically his Saturday session comes first, describing as it did, how C++ evolved. The early history should be well known here, but it was interesting to hear him coming back repeatedly to the issues of abstraction and localisation which led to the namespace feature. In the Friday session he had expanded into generic programming and the ways in which container templates, iterators and generic algorithms can lead to simpler (and thus more robust) code. Since about 80% of the third edition is new, *go and read the book!*

Leen Ammeraal's session *STL for the less experienced* would have made more sense (to me) had it come after the above instead of first thing on Friday, but the notes were detailed and, going through them afterwards, it fits in well. This was a thorough look, with code, at containers and iterators. As with some other presentations it depended on the view that the new standard would soon be reflected in available compilers, and one valuable feature was the annotations of how VC++5.0, BC++5.2 and one or two of their relatives, deal with the code fragments. All this and more is available in his *STL for C++ programmers* (Wiley 97).

The standards process, as such, was described by Tom Plumb in the first half of his session and he went on to describe some of the issues in compiler testing as they try to follow emerging standards. Perhaps because it was the after lunch slot, I thought that his audience found this a bit dry. The final "now for something different" session saw Bill Plauger producing *Embedded C++* as an unofficial (and mainly Japanese) response to the issue that C++ has now become rather large. The proper sub-set that is proposed approximates to earlier versions of C++, especially the library (as described in his *The Draft Standard C++ Library*, Prentice-Hall 1995). It appears that code-bloat comes from exceptions (adding about 50% to sample programs in all configurations) and from multiple inheritance, templates (and the new STL) and new-style casts. Details for those interested at http\\www.caravan.net/ec2plus.

There was a small but perfectly formed exhibition around the circulation space on Friday though all except Blackwells had given up for Saturday. If speakers and presentations live up to those described above in a reprise, then Parkway may have more success in selling space. I certainly hope there is a reprise and recommend that more of you get to it.

Ray Hall
Ray@ashworth.demon.co.uk

They pursued it with forks and hope.
By Alan Griffiths

The End of the Road for C++?

I've been using C++ since the first port of *cfront* (by Glockenspiel) appeared on the MS-Windows platform about a decade ago now. During that time the language has changed enormously, most of the changes, when considered in isolation, have been improvements, but overall the effect has been disconcerting. With other programming languages I feel confident that I've mastered it after a few months. With C++ I felt that I was starting again every few months. New features made my existing knowledge obsolete or invalid.

I'm not a typical developer - I've enjoyed the ride despite the frustrations. But, in my experience the typical developer rarely, if ever, opens a book or magazine to update their skills. It may be apocryphal, but I've heard of one company that spent a fortune employing consultants to fix Y2K problems in their application code but failed to change the habits of their own staff. After a year or so the exercise needed to be repeated. Obviously, few ACCU members fit this description of "typical developer" but most of us have to work in organisations in which we are the minority.

Two years ago the '95 "Committee draft" [CD1] was issued for public comment. This led to an exchange of articles in Overload 7, by myself and the then editor Sean Corfield. My contention was that the language had become too hard to use, and that too many legacy coding practices had been broken.

Since then I've seen the botched attempts that compiler implementers have made at implementing the language, and observed the problems the committee have had in ensuring that the standard was clear and self-consistent. For example, the standard

library relied heavily on a language feature that didn't exist in the language definition.

The '97 "Committee draft" [CD2] has been publicly available for a few months and is much clearer than the previous version. However, as I was aware of some problems with it I was convinced we wouldn't be seeing a "Draft International Standard" before the ACCU conference. Since the conference had been subtly arranged to abut the standards committee meeting in London, there were a number of members there. They were uniformly of the opinion that the major problems had been addressed, and that the remaining problems could be dealt with without significant delays.

I doubt that anyone on the C++ ANSI/ISO committees expected the standardisation process to take so long, or to lead to such problems. The obvious comparison is with the C standardisation process, which largely restricted itself to documenting existing practice, and by comparison went smoothly. In comparison the C++ standardisation process greatly extended "existing practice" (as described by the "Annotated Reference Manual" which is now of historic interest only) and added support for generic programming, exception handling and namespaces. Each of these is welcome and sustainable as an individual addition to the ARM language. However, the interactions between namespaces and templates contributed significantly to the delay between CD1 and CD2, and the interactions between exceptions and the generic programming library (STL) were not resolved as of CD2.

A lot of very bright individuals have contributed to the development of C++, and it is a tribute to their skills and enthusiasm that the standardisation process didn't collapse under the weight of these difficulties. We are about to have a standard definition of the language, but before we all breathe a collective sigh of relief there is one question to answer: "how long before the compiler and library implementers catch up?"

And now: Java!

When I started using C++ it was because it was a better applications programming language than C, and there were no sane alternatives for MS-Windows development at the time. However, it is far more suited to "systems programming" than to "application programming", and as it has required an increasing level of skill to use correctly has become less and less suited to use by the typical "applications programmer".

Java shows great promise as an application programming language. If you don't require the same degree of control and responsibility that C++ supplies, it is far easier to use. Given the hysterical level of support that it has in the industry I'd expect the tools to be in place for me to switch to using Java for application development early in '98. However, I don't expect to see it replace C++ in its chosen domain. For instance, I have some components that could not meet their performance envelopes running on the JVM - some optimisations are just not possible in that environment.

Assuming that Java fulfils my expectations, I predict a mixed language development model for application development with the majority of code in Java and heavily optimised or environment specific modules coded in C++. However, which particular glue holds this together is an interesting question - COM, CORBA, JNI, and native compilation of Java all have proponents - that more than one solution will be available is certain.

The C++ SIG

The "C++ Special Interest Group" was set up to cover C++ development topics that would not be of relevance to the general membership of the ACCU. During Sean Corfield's term as editor *Overload* has been of great benefit in keeping abreast of the

changes to the C++ language, but these changes are coming to an end and I feel that the need for such a role will diminish over the coming years.

At the same time we have Java, another language in the "C" family, which I anticipate many of the current C++ developers will (or should) be using in the next year or two. One possible reaction to this is to create a new "Java" SIG, but given that the people most likely to contribute to this are those already contributing to the C++ SIG, it would probably spread our efforts too thinly.

The responses I've had to my editorial in Overload 19 divide into two camps:

- "I paid for C++, that's what I expect." and
- "My interest in programming is more general than C++, don't be afraid to branch out."

As far as I can see, provided that we don't fail to publish the C++ material that is submitted then any other material is a bonus for the latter camp and may be ignored by the former.

Alan Griffiths
AGriffiths@ma.ccngroup.com

Technical Sub Editor

EXE, the monthly magazine for software developers, is looking for a technical sub editor to join its busy team. The right candidate will have superlative English and a thorough knowledge of software development.

You will be able to rewrite technical features and produce headlines and standfirsts to tight deadlines. Programming experience a significant advantage, HTML a definite plus. Opportunity to write features and news stories for the magazine and the Web site.

Please send a full CV, examples of your work, and a 300-word critique of any UK computing title to David Mery at:

EXE Magazine, Centaur Communications, St Giles House, 50 Poland Street, London W1V 4AX

or email to dmery@dotexe.demon.co.uk

BCS OOP Patterns Day

The British Computer Society, Object-Oriented Programming and Systems Specialist Group, are having a 'Patterns Day' on Saturday 18th October 1997.

It's an all-day event at the IBM Centre South Bank and will include:

- Keynote presentations
- Pattern Writers workshops
- Patterns Readers' Groups
- Interactive workshops
- Public launch of the UK Patterns' Group

Speaking will be:

Jim Coplien, Bell Laboratories, US (author of "Advanced C++" and co-editor of the 2 "Pattern Languages of Program Design" books)

Neil Harrison, Lucent Technologies (co-author of an Organisational Patterns pattern language)

Franck Buschmann, Siemens AG (lead author of Patterns-Oriented Software Architecture)

Suzanne Robertson, Atlantic Guild (author of a Requirements Patterns pattern language)*

Booking details are available from the BCS OOPS SIG home page

(http://www.sis.port.ac.uk/bcs-oops.html) or through the OOPS Treasurer, Ray Warburton, email warburton@hvlc.demon.co.uk, or by snail mail to High View Development Ltd, 5 High View, Steep Street, Chepstow, Monmouth NP6 5 QB .

editor << letters;

Overload Future

John Merrells

Yes, a letter to myself.

Unfortunately, despite my general pleas, Overload 20 didn't generate any letters. So, Vox Pox ahoy, I beat some of you subscribers up with a reader's survey. I e-mailed 40 people to solicit some feedback on the future of Overload. Only 8 replied! Of the rest: 9 of the addresses bounced, 3 had resigned from the ACCU, and 2 no longer subscribed to Overload. So, when you get an email from us next week, I'll be expecting great things.

Below are some snippets from your comments. They are not attributed to authors as I didn't expressly state that I would be publishing anything from the mail exchanges.

Should Overload contain Java articles?

- I don't think so. The ACCU SIGs allow members to subscribe to groups which closely match their set of interests. I think that Java articles should appear in C Vu and, if there is enough interest, and we can find an organiser, we should start an ACCU Java SIG.

- Yes. I have minimal experience of Java, and will probably need to learn more, in the next year or so

- Yes, probably, but don't go overboard.

- Yes. It's a great platform. Just don't jump on the bandwagon with articles that everyone has done before. Most Java related text I see covers the same ground - simple 'wow' things. There must be some meat to it.

- Java articles would be of interest, but may be of sufficiently general interest that they belong in CVu? Rather than in Overload which goes out to a fraction of the ACCU membership.

- Two more agree without comments.

Should Overload contain OO articles?

- Yes. Most people I know in the industry are good(ish) programmers, but don't give enough thought to OOD and decisions which will affect future modifications. With some OOA/D skills they would be writing more maintainable and useful code.

- I'd prefer introductory and language-independent OO articles to appear in C Vu. But, given that analysis, design, and patterns are very important for writing non-trivial C++ programs, I'd welcome them in Overload. Notations such as UML are very useful for describing systems of related or collaborating objects - they should definitely be a part of Overload articles.

- I don't see any harm. I would lump Java and OO together. Throw in Eiffel and Smalltalk too. Basically I want to understand OO techniques better. Within the OO community there are still arguments raging about single inheritance vs. multiple inheritance, garbage collection vs. non-garbage collection. Of course different problems lend themselves to different languages.

- OO articles would be good. Something along the lines of the articles Paul Field did in CAUGers, but with (say) a UML spin and more general interest.

- Yes, particularly patterns.

- Three more agree without comments.

Is there anything else Overload should contain?

- Basically I get Overload to improve my C++ and learn more. The majority of articles are way over my head. I'd like to see more explanation of syntax, especially for templates.

- Less pedantry. I would like to see fewer articles dealing with obscure points of syntax and more articles dealing with the application of OOL to real problems.

- Comparatives against other languages. CASE & CAST tools experiences.

- Less 'language' articles, more on good design techniques.

- Some decent compiler reviews. The CVu book reviews really help cut through a lot of the dross; something which gave me some equally unbiased help when looking at compilers would be great.

General comments

- I am a very very 'early learner' with C++, many of the articles are interesting but way beyond my comprehension.

- I do like the mixture of opinions, introductions to new features, techniques, things to avoid, and so on. I also particularly like Overload for being platform-independent.

Conclusion

This, admittedly small, sample of subscribers seem to feel positively towards Overload covering general Object Oriented topics, and even articles about languages other than C++.

But, of course, and as ever, we can only publish what we receive. If you want Overload to broaden its horizons, then you must generate some 'alternative' material. In Overload 22 I'd like to print an introductory article about UML, and how to implement a common pattern in Java.

John Merrells
john.merrells@octel.com

ACCU and the 'net

ACCU.general

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

Demon FTP site

The contents of *CVu* disks, and hence the code from *Overload* articles, eventually ends up on Demon's main FTP site:
ftp://ftp.demon.co.uk/accu
Files are organised by *CVu* issue.

ACCU web page

Thanks to Net Access and DeMontfort University we now have a machine permanently connected to the Internet. The official ACCU web pages have moved to a new home.
http://www.accu.org/

C++ – The UK information site

This site is maintained by Steve Rumsby, long-serving member of the UK delegation to WG21 and nearly always head of delegation.
http://www.maths.warwick.ac.uk/c++

C++ – Beyond the ARM

Sean Corfield maintains a set of pages about recently added C++ features. He welcomes feedback on their content.
http://www.ocsltd.com/c++

Contacting the ACCU committee

Individual committee members can be contacted at the addresses given above. In addition, the following generic email addresses exist:
caugers@accu.org
chair@accu.org
cvu@accu.org
info@accu.org
info.deutschland@accu.org
membership@accu.org
overload@accu.org
publicity@accu.org
secretary@accu.org
standards@accu.org
treasurer@accu.org
webmaster@accu.org
There are actually a few others but I think you'll find the list above fairly exhaustive!

Credits

Founding Editor

Mike Toms

Editor

John Merrells
4 Park Mount, Harpenden, Herts, AL5 3AR.
john.merrells@octel.com

Readers

Ray Hall
Ray@ashworth.demon.co.uk

Ian Bruntlett

Einar Nilsen-Nygaard
EinarNN@atl.co.uk
einar@rhuagh.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

David Hodge
31 Egerton Road
Bexhill-on-Sea, East Sussex. TN39 3HJ
davidhodge@compuserve.com

Copyrights and Trademarks

Some articles and other contributions use terms which are either registered trademarks or claimed as such. The use of such terms is intended neither to support nor disparage any trademark claim. On request, we will withdraw all references to a specific trademark and its owner.

By default the copyright of all material published by ACCU is the exclusive property of ACCU. An author of an article or column (not a letter or review of software or book) may explicitly offer single (first serial) publication rights and thereby retain all other rights. Except for licences granted to (1) Corporate Members to copy solely for internal distribution (2) members to copy source code for use on their own computers, no material can be copied from *Overload* without written permission of the copyright holder.

Copy deadline

All articles intended for inclusion in *Overload 22* should be submitted to the editor, John Merrells <john.merrells@octel.com>, by September 15th.