contents

credits & contacts

Overload Editor:
John Merrells
overload@accu.org

Contributing Editor:
Alan Griffiths
alan@octopull.demon.co.uk

Readers:

Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Pete Goodliffe,
Chris Lowe
ads@accu.org

Overload is a publication of the ACCU. For details of the ACCU and other ACCU publications and activities, see the ACCU website.

ACCU Website:
http://www.accu.org/

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial – How much is a good thing?

"How much what is a good thing?" you may ask – the answer is "many things". In design it is often "how much abstraction is a good thing". In explaining it is often "how much simplification is a good thing?" In editing Overload it is "how much change is a good thing". Change is double edged: that which isn't prepared to change must be prepared to design, but inappropriate changes can lead to a quick death. The other problem with change is that it can catch the unwary. And, if you refer to my last editorial you will see that I was unaware of a change. (If you don't know what I'm talking about don't be concerned – I'm not being mysterious – all will be explained after a short digression into the role that Overload plays in ACCU.)

Sometimes queries arise as to why ACCU produces two publications – especially as it is sometimes unclear why material appears in one rather than the other. There are many answers, but, to me at least, the thing that distinguishes Overload from C Vu is that Overload covers design issues at a range of levels of interest. In comparison C Vu is much more focused on programming, and where design appears in C Vu it is mostly at the level of idioms. I'm happy with this division of scope, but in writing the last editorial had failed to realise quite how far away from its origins as a specialist C++ publication Overload has evolved. I wrote that I was unsure if a pattern language about team working would be appropriate. The editor and "readers" soon put me right about this! Of course, I'm happy about this – where else would I get such an article published? – but I do wonder if the readership is being left behind by the pace being set. I hope not, but unless some of you make your views known to us I can't be confident.

Whenever I meet up with ACCU members the conversation sooner or later turns to complaints about the way the industry operates. You know the sort of thing: why do managers make unreasonable demands? Why are (other) developers so incompetent? ... You have been there; you know how it goes. These issues are often raised as rhetorical questions; but, they are genuine problems and deserve answers; and, it is only by seeking answers that we can lay the groundwork for an effective solution. This type of issue should be appropriate to ACCU publications – after all it is clearly of interest to the members. And it was the view of the rest of the Overload team that it was appropriate for Overload to publish material that presents an analysis of these questions.

On the basis of my experience and beliefs I'm certain that in looking for solutions we should apply the lessons that we've learnt in our work. In particular, we know that some ways of communicating a solution are more effective than others. A particular lesson I draw from the "design patterns" movement is that it is helpful to include details of the motivating problem, the solution, and any trade-offs in the discussion of a "solution". However, when dealing with our less enlightened colleagues we often find orphaned solutions cut off from the original problems or rationale. And it is often these orphaned solutions that appear unreasonable. But is dismissing them as unreasonable a reasonable reaction? Is it just that we fail to recognise the motivating problem they successfully solve or is it that they are being used where they are not applicable?

One such "solution" that cropped up in discussion at a recent get-together of ACCU members in Nottingham – they know who they were – comes under the banner of "don't do as I do; do as I tell you". That is: a widespread tendency to make a statement about how things should be done and then to ignore it. A developer might tell you that maintainability is the most important concern in writing their code, but may be found trying the latest "cool trick" they have found in the language. Or an author may write, "error handling is important" and then omit mention of it throughout the book. Or a methodologist may say, "team building is important" and then only discuss the work processes and the work products.

We all know that these strategies can have undesirable consequences. The developer is unlikely to produce maintainable code from their experiment (although they may well learn something that can improve subsequent work). The author will fail to inform the reader how code should be written in a production environment (although they may successfully teach the principles). And the methodologist won't convey successful strategies for running a successful project (but may provide some useful milestones). But, instead of just complaining about the consequences, it is important to recognise that all of these behaviours reflect solutions to problems – and to consider what these problems might be.

It is easy to assume that these characters are idiots, but if we make the effort to think about what they are trying to achieve we see that there is a common thread running

amongst them: "how much scope is a good thing". Without understanding the problem they face it is impossible to decide whether their approach improves the situation. The developer might need to master a new language feature in order to express the solution to a problem effectively. The author may need to simplify the subject matter to communicate the ideas or because of space constraints. The methodologist may be addressing the more serious problems. (Admittedly they might not – just don't jump to this conclusion.)

Whatever the case, if you don't take the time to understand why someone is acting the way they are then you will fail to engage them in a discussion of the merits of that behaviour. I have a lot of sympathy with the developer, author and methodologist (because I've done these things). I also have a lot of sympathy with confronting them (done that too). This editorial isn't really about solving these problems (and I don't have all the answers) but rather than leave them unresolved this is what I do in their positions:

- As a developer experimenting with a new coding technique I try to leave it a while and to write it up before using it in production. (I'll be submitting an article about template metaprogramming for the next issue.) And if I can't explain it then I don't understand it well enough to use it. (So, if you don't see the article next time...) Occasionally I feel tempted to succumb to the "haven't got time for it" argument – but every time I do that I am reminded that "if you haven't got time to do it right then you haven't got time to do it wrong first and then fix it".
- As an author I try to structure my code so that the error handling is as painless as possible and doesn't need to be omitted in articles. More often I'll omit whole functions or sections of code whose implementation is unsurprising. Sometimes I have to omit error handling during the initial presentation of ideas and then cover errors in a second pass. (And sometimes that second pass gets cut for space or time – to my subsequent regret.) But messy error handling usually indicates bad design: mine, or the API, or the programming language that I'm working with. (And that leads to another interesting discussion: how to work within the constraints of an imposed bad design.)
- In talking about development methods I find that it is problematic to get ideas relating to team formation and environmental factors across. I don't yet know if this is a problem with me or the audience. (I have my suspicions – one senior manager dismissed my observation that improved morale on a project indicated that things were getting better with "they feel better because they think they can meet the new delivery dates, but that doesn't prove they will".) Seriously, though: it is my task to get the message through and I'm still working on it. (If you have ideas that can help, I, and a certain journal, will be interested.)

The moral of this editorial is to ensure that we understand the impact that our actions will have on others and the motives of others whose actions affect us. If only everyone in the industry would write code others will understand, would explain (and apply) techniques for handling error conditions, and would promote teamwork and improve (not denigrate) people's ways of working. If only everyone could be like us...

The team that produces Overload is dealing with a problem: the world is changing and we change with it. ACCU is changing – what was once a C specialist user group has now expanded its interests. Not only to other languages (C++, Java, Python, C#, etc.) but to software design, working practices and organisational issues. I think that most of us are programmers because we delight in novelty – I would certainly prefer to solve a new problem each day than to repeat a tired old solution endlessly. This is unusual in the population at large, but the ACCU members I speak to have that same attitude. Overload is changing – because the editorial team is solving the problems they think matter. These problems are reflected in the articles submitted and the experience of the team members. If you don't like our solution, try to understand why we are doing this – and if you have a better solution then we'll be glad of your help.

Alan Griffiths
alan@octopull.demon.co.uk

Copy Deadlines

All articles intended for publication in *Overload 57* should be submitted to the editor by September 1st, and for *Overload 58* by November 1st

Reshaping an Old Piece of Design

by Mark Radford

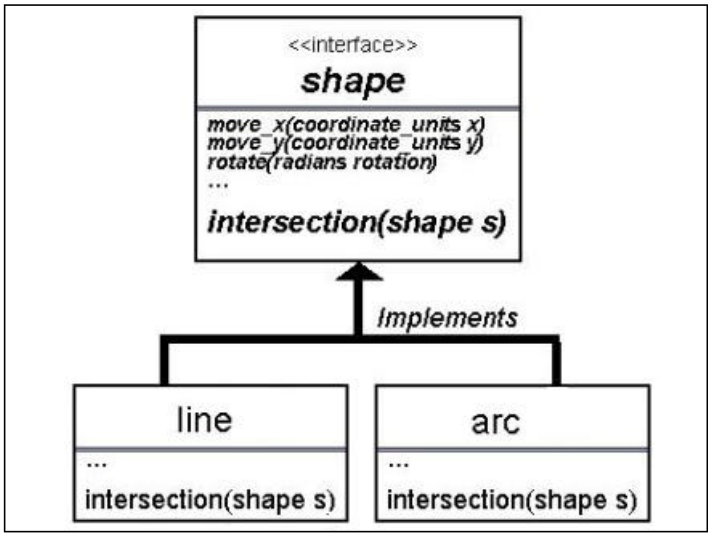
In C++, virtual functions are fundamental in supporting the capability to implement an object-oriented design. They allow a call to a member function made on a pointer/reference to a base class to result in a member function of the object's concrete class being called. In doing so, they are the language's fundamental mechanism of run time polymorphism – the function actually called depends on the type of the object pointed to, as determined at *run time*.

Sometimes being able to select a function to call based on the run time type of *one* object is not enough. Sometimes there is a need to create the effect of a function being virtual with respect to two or more objects. Some languages (e.g. CLOS) have such a mechanism, and such functions are known as *multi-methods*. However C++ has no such feature, and where multi-methods are required in C++ the effect must be achieved using design and programming techniques.

In this article I will first describe a problem I once faced, that motivated me to take an interest in these techniques. I will describe the solution I chose (which unfortunately was not a good one) and the alternatives I considered, examining the tradeoffs they offer. Then I will go on to look at the solution I would choose if I faced the problem now, and explain why I would prefer it.

The Problem

A few years ago I was involved in the development of a package for producing two-dimensional technical drawings. The drawing program supported two basic shapes: straight lines, and semi-circular arcs, and it is easy to understand how the hierarchy in the figure below was fundamental to the design.



It is obvious that these shapes would need an interface capable of supporting the operations the user is certain to expect, such as being able to move the shapes around and rotate them. However, because the program was for producing drawings of a technical nature – essentially 2D CAD – an operation to calculate the *intersection* with another shape was also necessary. Unfortunately having available a shape abstraction is not good enough: the `intersection()` methods need to implement the intersection calculation formula, and implementing the formula requires the concrete type of both shapes. In passing, it was to my delight that I found Bjarne Stroustrup cites almost this very problem as an example (in [D&E]) of where multi-methods would be useful.

The solution I came up with at the time was not very good and the irritating thing was that I knew I knew this – I just didn't know what else to do. I could think of other approaches, but they all seemed worse than the one I used. For example, some sources (e.g. [More Effective C++]) use the brute force approach of downcasting in conjunction with RTTI. In hindsight though, the RTTI approach offered a better set of tradeoffs.

This problem has been in my mind (on and off) ever since, and years later, I have come up with what I think is a satisfactory approach.

Two Alternative Solutions

I considered two solutions at the time. One of them worked by finding out the run time types of the shapes using run time type information (RTTI); this could be described as a "brute force" approach. The alternative used an object-oriented approach, and I consider it to be a classic example of a solution being flawed while being unquestionably object-oriented.

Solution 1: The RTTI Approach

First consider what a fragment of the code to implement this approach would look like. Here, `dynamic_cast` is used to check for each possible type, and to provide the necessary downward conversion (or downcast):

```
void intersection(const line& l, const shape& s,
                  intersection_points& where) {
  if (const line* lp
          = dynamic_cast<const line*>(&s)) {
    lines_intersection(l, *lp, where);
  }
  else if (const arc* ap
          = dynamic_cast<const arc*>(&s)) {
    line_arc_intersection(l, *ap, where);
  }
  else
    //..
}

void intersection(const arc& a, const shape& s,
                  intersection_points& where)
{ /* .. */ }
```

Now consider the consequences of adding a new specialisation of `shape`, e.g. an elliptical arc. This would mean two things:

1 Adding a new `intersection()` function overload.

2 Adding more code to the existing `intersection` functions.

In passing, note there is a historical twist to my rejection of this solution: neither `dynamic_cast`, nor any other form of RTTI for that matter (remember I said it was a few years ago), were implemented in the compiler used on the 2D CAD project! Therefore, this approach would have required the manual implementation of some kind of an RTTI substitute (e.g. each class having an integer constant to identify it).

Solution 2: A Flawed Object Oriented Approach

This is the solution I implemented at the time. It employs an object-oriented mechanism of type recovery using virtual functions. The mechanism takes advantage of the fact that an object's concrete type is known within the member functions of the object's class.

Let's look at a C++ fragment showing relevant parts of the `shape` hierarchy's class definitions:

```
class shape {
public:
  virtual ~shape();
  virtual void intersection(const shape& s,
        intersection_points& where) const = 0;
  virtual void intersection(const arc& s,
        intersection_points& where) const = 0;
  virtual void intersection(const line& s,
        intersection_points& where) const = 0;
//...
};
class arc : public shape {
private:
  virtual void intersection(const shape& s,
        intersection_points& where) const;
  virtual void intersection(const arc& s,
        intersection_points& where) const;
  virtual void intersection(const line& s,
        intersection_points& where) const;
// ...
};
class line : public shape {
private:
  virtual void intersection(const shape& s,
        intersection_points& where) const;
  virtual void intersection(const arc& s,
        intersection_points& where) const;
  virtual void intersection(const line& s,
        intersection_points& where) const;
// ...
};
```

The `shape` class provides the interface class heading up the hierarchy. Note that it has a virtual function overload taking `shape` as a parameter, as well as one for each of `line` and `arc`; if another type of shape (e.g. an elliptical arc) were ever to be added to the hierarchy, `shape` would need a further virtual function taking the new type as a parameter, and derived classes would need to implement it. Therefore, this design is awkward to extend because it would require a change to code in many of the files participating in the implementation of the `shape` hierarchy.

The next code fragment shows what happens during an attempt to find the intersection (if any) of objects of type `line` and `arc`:

```
shape* shape1 = new line(..);
shape* shape2 = new arc(..);

shape1->intersection(*shape2, where);
                // Calls line::intersection()

void line::intersection(const shape& s,
          intersection_points& where) const {
  s.intersection(*this, where);
                // Call is re-dispatched...
}
void arc::intersection(const line& s,
         intersection_points& where) const {
  line_arc_intersection(s, *this, where);
          // ...and handled by the
          // arc::intersection()
          // overload that handles lines
}
```

The first call is made on an object of concrete type `line`, so the first virtual function implementation entered is that of the overload `line::intersection(const shape&, ..)`. Note: the type of the pointer returned by this is `line*` (rather than `shape*`).

Next, a call to `s.intersection(*this, ..)` is made, and results in a call to the `intersection()` overload taking a `line` as a parameter. Given that the pointer passed in (i.e. `shape2`) points to an object of concrete type `arc`, the result is a call to `arc::intersection(const line&, ..)`. Now the concrete types of both objects is known.

Sadly this solution is flawed because, in a nutshell, it renders derived classes intrusive not only on each other, but also on the base class. It must be remembered that calculating intersection points is only *one* aspect of shape functionality, yet providing it needs three virtual functions in the interface of each class in the hierarchy.

Towards A Better Solution (?)

In seeking a better solution, I'm going to start by asserting that the flawed object oriented solution would actually have been quite reasonable but for one thing: classes are *intrusive* on each other. My point is that this intrusiveness would not be such a problem if it could be compartmentalised and therefore its impact limited. To this end I will recruit the help of the EXTENSION OBJECT design pattern (originally documented by Erich Gamma – see [PLoPD3] for the full write-up). What follows is only a brief and slightly C++ centric summary of the pattern, but the description (below) of how it is used to implement a better solution should complete the picture.

Pattern

EXTENSION OBJECT.

Context, problem and forces

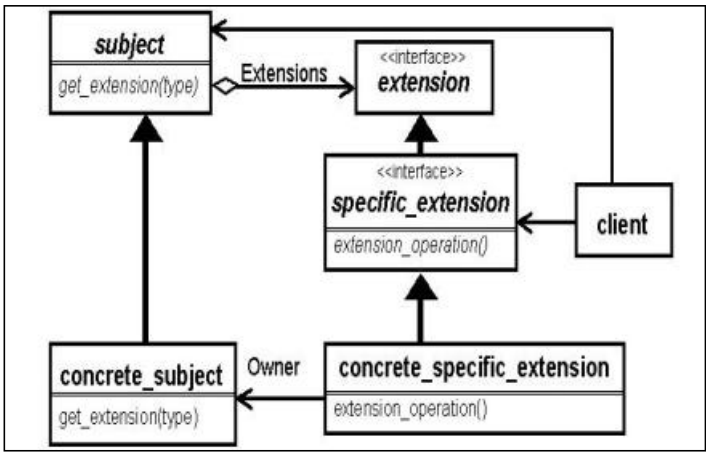
Different clients will have different requirements of an object's interface. The precise interface that will be required by each client cannot always be anticipated at design time. Also, it is often unacceptable to trade provision for them against the interface bloat that would result. In C++ this problem can be addressed to some extent by an approach using freestanding functions. However this does not solve all the (potential) problems (for example, freestanding functions cannot be virtual).

Solution

Support the additional interfaces using separate objects and give the *Subject* an interface for returning *Extension Objects*.

Configuration

The extensions hierarchy (see figure below) is headed up by the `extension` interface, while the facilities the extension offers to



clients are made available through the interface `specific_extension`.

The `extension` interface does not support the operations required by the client, because different extensions will offer different operations. Therefore client obtains access to extensions via `get_extension()`, to which it passes *type*, where *type* is simply some kind of indication of the extension type being requested.

Consequences

It can be seen that this pattern offers benefits in terms of flexible extensibility, but there are some drawbacks, for example:

1 Some of the behaviour of subject is moved out of it, so subject no longer expresses all the behaviour that clients can perceive it as having (whether this is a good or bad thing depends on the actual behaviour).

2 The client code will need to recover the `specific_extension` type. A typical method of doing so in C++ is by using `dynamic_cast`. Therefore, clients become more complex in the face of the "machinery" needed to use the extensions. This machinery can be encapsulated, but the issue still needs to be kept in mind.

Solution Using Extension Objects

The solution presented as a flawed object-oriented solution was in some ways an attractive one, exhibiting the benefits of object-oriented design, keeping code performing a function together and separate from code performing other functions. It was only flawed as a consequence of making classes within the shape hierarchy intrusive on each other, and the interface clutter caused (three virtual functions were needed in each class's interface). Introducing EXTENSION OBJECT allows the same mechanisms to be deployed while keeping the intrusiveness and interface clutter out of the shape hierarchy. The design now looks as shown in the figure below.

In this design, the following mappings from the EXTENSION OBJECT configuration are used:

- `shape`'s `create()` method takes over from subject's `get_extension()` method. This is because of a C++ object lifecycle issue that will soon become clear.
- `shape_extension` and `shape_intersector` assume the roles of `extension` and `specific_extension`, respectively.
- `line_intersector` and `arc_intersector` are the `concrete_specific_extensions`.

As an aid to understanding these mappings, the names from the configuration are used as *stereotypes* in the exposition in UML (see figure below).

Implementation

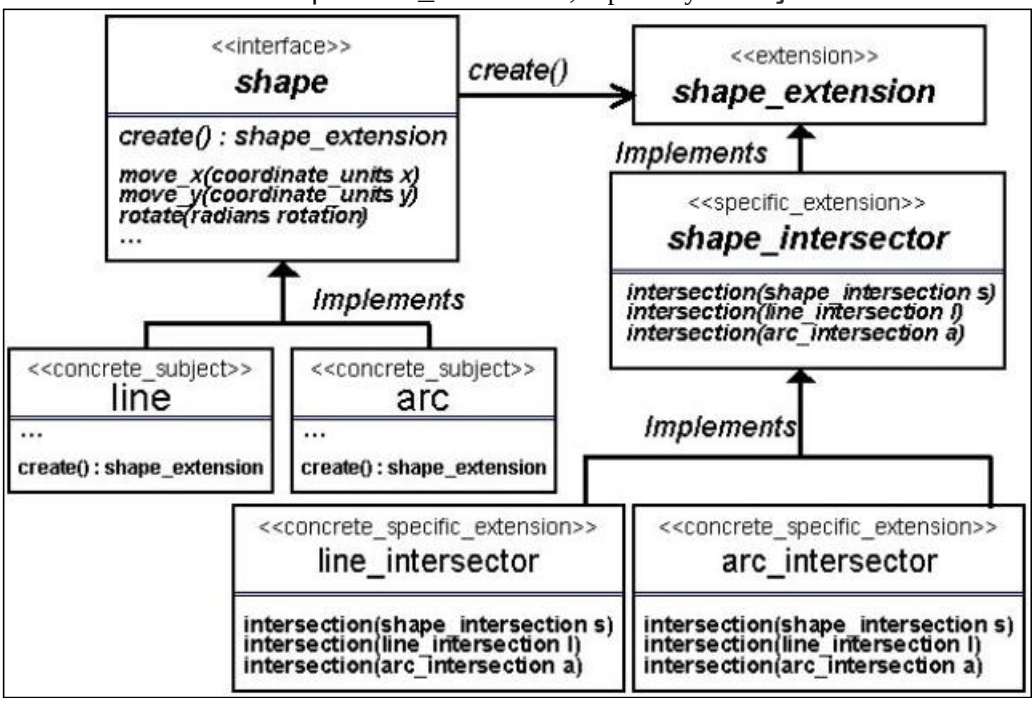
The mechanics of recovering the types and working out the intersection points are the same as in the flawed solution – the only difference is that this time the participants are `shape_intersector`, `arc_intersector`, `line_intersector` and the additional `shape_extension`.

The class definition contains very little:

```
class shape_extension {
public:
  virtual ~shape_extension();
// ...
};
```

It has a virtual destructor, but that needs no explanation:

```
namespace intersections {
class shape_intersector {
public:
  virtual ~shape_intersector();
  virtual void intersection(
       const shape_intersector& obj,
       intersection_points& where) const = 0;
  virtual void intersection(
       const line_intersector& obj,
       intersection_points& where) const = 0;
  virtual void intersection(
       const arc_intersector& obj,
       intersection_points& where) const = 0;
  // ...
  };
boost::shared_ptr<shape_intersector>
   down_cast(
      boost::shared_ptr<shape_extension> obj);
}
```



The `shape_intersector` class is the first one in the hierarchy to have an interface of any substance. It declares `intersection()` member function overloads in much the same way as `shape` did in the flawed object oriented solution – the difference here being that these overloads take `line_intersector` and `arc_intersector` parameters, in place of `line` and `arc` parameters respectively.

Another declaration of interest is that of the `down_cast()` function: not a member of `shape_intersector` but provided within the `intersections` namespace. To understand its role, first we need to look at `shape`:

```
class shape {
public:
  virtual void move_x(coordinate_units x) = 0;
  virtual void move_y(coordinate_units y) = 0;
  virtual void rotate(radians rotation) = 0;
// ...
  virtual boost::shared_ptr<shape_extension>
   create(const std::type_info& type) const=0;
};
```

The `shape` interface class provides (besides the functional interface supporting user operations) a virtual member factory function `create()` that returns a `shape_extension` instance. Here there is a deviation from the canonical EXTENSION OBJECT configuration, because `concrete_subject` (`line` or `arc`, omitted from the UML diagram) is designated as its owner, which is not quite the case here. The design in this example uses the C++ idiom of using a smart pointer to manage memory acquisition and release, to avoid running into problems with object lifetimes.

Returning to `down_cast()`: in order to use the `shape_intersector` interface, the `shared_ptr<shape_extension>` instance returned from `shape::create()` must be converted to type `shared_ptr<shape_intersector>` (remember this was listed as a consequence of the EXTENSION OBJECT design pattern). A custom mechanism in the form of `down_cast()` is provided to achieve this, because unfortunately the use of a smart pointer cuts across the natural approach of using `dynamic_cast`.

The definitions of classes `line` and `arc` are self-explanatory: they just provide implementations of `shape`'s virtual member functions `move_x()`, `move_y()`, `rotate()` etc. I'm not going to list them here because I don't believe they will actually add anything to the illustration. Instead I'm going to move on to `intersection()`, another freestanding function declared within the `intersections` namespace:

```
namespace intersections {
intersection_points intersection(
      const shape& s1, const shape& s2) { // 1
  intersection_points where;
  boost::shared_ptr<shape_intersector> first
     = down_cast(s1.create(
           typeid(shape_intersector)));  // 2
  boost::shared_ptr<shape_intersector> second
     = down_cast(s2.create(
           typeid(shape_intersector)));  // 3
  first->intersection(*second, where);    // 4
  return where;
} }
```

Before looking at `intersection()`'s implementation, I feel it is worth digressing briefly to look at a trade-off that has been made. It was observed that as a *consequence* of the EXTENSION OBJECT design pattern, the machinery for obtaining `extension` (`shape_extension`) instances and down casting them to `specific_extension` (`shape_intersector`) adds complexity to clients. It was also observed that one way to address this complexity is to *encapsulate* it, and this is the approach taken here: i.e. it's all wrapped up in the `intersection()` function. This encapsulation introduces a tension with the design decision to create `shape_extension` instances on the heap (instead of the originating object owning them): there is no way to preserve these instances between calls to `intersection()`. Thus efficiency is traded for simplicity of usage (and tidiness of exposition in an article :-)).

Getting back to `intersection()`'s implementation...

The function takes two `shape` instances (by reference so they exhibit run time polymorphism), `s1` and `s2`, as its parameters (statement 1). Statements 2 and 3 create `first` and `second`, these being the `shape_intersector` instances, and here two things should be observed:

- The `shape::create()` function is called within the call to `down_cast()` so the instances, although present, never appear explicitly as type `shape_extension`.
- In the calls to `shape::create()`, the arguments are in both cases `typeid(shape_intersector)`, i.e. not the typeid of the most derived classes. This is because the concrete classes `line` and `arc` know they must create `line_intersector` and `arc_intersector` respectively – they only need to be told they are creating extensions to a type `shape_intersector`, as opposed to any other type of extension.

Statement 4 is where the intersections (if any) are calculated. The rest of how this works is very similar to the way in which the flawed object-oriented solution worked:

```
shape* shape1 = new line();
shape* shape2 = new arc();
```

And their intersections calculated:

```
intersection_points where =
intersection(*shape1, *shape2);
```

The workings of the `intersection()` function were explained above, so we now need to look at how `line_intersector::intersection()` and `arc_intersector::intersection()` work. When `intersection()` is called with `shape1` and `shape2` as arguments, statement 4 in its implementation results in a call to the `line_intersector::intersection()` overload taking a `shape_intersector` parameter:

```
void line_intersector::intersection(
         const shape_intersector& s,
         intersection_points& where) const {
  s.intersection(*this, where);
              // Call is re-dispatched...
}
```

Remember `shape2` has concrete type `arc`, so re-dispatching the call results in a call to `arc_intersector::intersection()` – specifically, the overload that takes a `line_intersector` as a parameter:

```
void arc_intersector::intersection(
        const line_intersector& s,
        intersection_points& where) const {
  line_arc_intersection(s, *this, where);
}
```

That's it. At this point the concrete types of both `shape_intersector`s are known, and the calculation (details of which we are not concerned with here) can be performed.

Phew!

Tradeoffs – In Favour

Intersection logic is non-intrusive with respect to the `shape` hierarchy. In the case of the flawed object oriented solution, the

[concluded at foot of next page]

Stream-Based Parsing in C++

by Frank Antonsen

This paper shows how to implement general parsers as a family of streams. This allows for very readable, maintainable and flexible parsers. The method is illustrated with a parser for simple arithmetic expressions, but can easily be extended to a parser for a full-fledged programming language. Moreover, the same technique can be applied to the entire process from lexing to execution, since actions can be associated with each sub-parser.

Introduction

The parsing of input is a very important problem appearing in many different parts of software development – parsing user input in the form of command-line options, the parsing of arithmetic expressions in a calculator, parsing values in a user-defined configuration file or compiling some programming language.

This makes it important to have different approaches. What we will present here in this paper, is a method of parsing inspired by what is done in functional programming (FP). The paper is *not* about functional programming in C++[1], rather it is about how to implement a particularly elegant idea from FP in an object oriented context.

The entire process, from lexical analysis to actual execution of a program can be divided into a number of individual steps:

```
source -> lexer -> parser -> optimiser
       -> execution -> output
```

In FP, this could be represented by a family of functions:

```
output = execution ° optimiser ° parser
         ° lexer(source)
```

The advantage of this approach is *flexibility*: it is easy to omit or modify individual steps, which is important for playing around with different approaches in language design or compiler construction.

One could attempt to implement the different sub-parsers as functions or (better) function objects in C++. The disadvantage of doing so, however, is the proliferation of parentheses this would entail. In C++ there is no operator for function (or functor) composition corresponding to ° above (which may be called many different things in a functional language – e.g. a period or simply the letter 'o'). Nor can such an operator be defined in a natural way – none of the overloadable operators are well suited for becoming composition operators.[2]

Moreover, the presence of too many parentheses makes the code harder to read and is hence also more error prone.

One can consider a function as a stream, however. Instead of writing `x=f(y)` one could try to write `x << f << y`. Here, we do have a natural operator for composition, namely the stream operator `<<`. This suggests considering the entire process as a collection of streams:

```
output << execution << optimiser
       << parser << lexer << source
```

1 Although, C++ being a multi-paradigm language this would be a worthwhile topic.

2 That being said, it ought to be mentioned too that the Standard Template Library (STL) has introduced many FP features, among these a limited support for partial binding and the ability to write function composition operators, [7], although this still cannot be done by overloading a natural operator.

[continued from previous page]

problem was that derived classes were intrusive on the base class, and on each other. In the case of the solution that uses EXTENSION OBJECTS, classes derived from `shape_intersector` are also intrusive on each other, but there is a very important difference: there is no intrusiveness on the `shape` hierarchy. For example: if another shape is added, only the classes in the multi-methods hierarchy are affected.

Note that, in the case of the example of adding another type of shape (an elliptical arc for example), the bodies of existing `shape_intersector` member functions will not need their implementations changing. This is a consequence of virtual functions being used to automate the control flow by placing it in the hands of the C++ language. By contrast, in the case of the RTTI solution, the control flow is implemented directly in the code, and as a consequence adding the code for a new type of shape means modifying existing code. In the former case, the absence of a need to change existing code means that the chance of introducing an error into it is reduced.

Tradeoffs – Against

The `shape` and `shape_intersector` hierarchies have parallel corresponding classes. Working with and maintaining such parallel hierarchies always creates a balancing act of design.

The most obvious burden is the extra types that now inhabit the design, and these must be managed – not just in physical terms but also in addressing the communication issues that arise (more documentation will be needed).

More subtle is the lack of any direct mention of intersections in the `shape` interface, and in the interfaces of classes derived from it. Here, a consequence associated with applying the EXTENSION OBJECT design pattern haunts the design.

In Conclusion

Using the object-oriented paradigm does not automatically make a design superior. In the past object orientation has been adopted in the hope that it would be the silver bullet that would solve all software development problems. Of course, history now records that nothing was further from the truth. There were many factors involved, one being the lack of understanding of object orientation itself. Another critical factor however, was the assumption that being object oriented automatically made a design a good one. The flawed object oriented solution presented earlier is an excellent counter example.

An important lesson is that even good OOD has its costs. It comes back to the fact that when solving problems with any level of complexity, there is no such thing as a solution per-se – there are options and tradeoffs.

Finally, the BSI C++ panel are currently discussing a proposal by Julian Smith to add multi-methods to the language – therefore this feature may or may not be present in the language when the next edition of the standard appears. Full details of the proposal can be found at Julian's web site (see [Multi-Methods Proposal]).

Mark Radford

References

[Boost] The Boost library (see `www.boost.org`)

[D&E] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994

[More Effective C++] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.

[Multi-Methods Proposal] Julian Smith's proposal for adding multi-methods to C++ (`www.op59.net/cmm/readme.html`)

[PLoPD3] Robert Martin, Dirk Riehle and Frank Buschmann (Editors), *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

This won't work, however, since `<<` is left-associative. Hence, either one needs to introduce parentheses once more, e.g., write `x << (f << y)` and so on, or one will need to use another, right-associative operator. The former case will automatically suffer from the abovementioned problems with using function objects.

Consequently, we will have to use a right-associative operator instead. Unfortunately, C++ does not allow one to declare an operator to have a user-defined associativity, so instead we will have to replace `<<` by one of the right-associative operators defined by C++. There are very, very few of these. In fact, the only binary right-associative operators are the assignment operators `+=`, `*=`,.... Thus, the simplest possible change is to use `operator<<=`. This also has the added advantage of resembling an arrow, more showing the direction of the flow of data.

Now, even though strictly speaking the sub-parsers won't be stream objects, we will still refer to them as such by analogy to ordinary streams (I/O stream, file streams, string streams) present in C++. The reason being that the defining feature of a stream is its ability to process input and to be pipelined, which is precisely what our generalised stream will do.

In this paper, we will concentrate on the *parsing* step, but in such a way that the remaining steps of the process could be implemented in a similar fashion. In fact, we will see how to make a simple modification to our parser and turn it into an expression calculator. For concreteness, we will consider a particular example, which is simple enough not to introduce unnecessary complications yet complex enough to be non-trivial. The chosen example is the parsing of arithmetic expressions like `1+(2-3)*4`. We will only consider integers.[3] Furthermore, we will not worry about the precedence levels of the standard arithmetic operators – this could be done by slightly modifying the grammar as shown in for instance, [1]. It will be shown later how to accommodate this with very few changes to our framework.

Hence, the basic ingredients in our language are:

```
<digit>     ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                7 | 8 | 9
<operator>  ::= + | - | * | /
<lpar>      ::= (
<rpar>      ::= )
```

To this we add the definition of a number as a sequence of one or more digits:

```
<num> ::= <digit>+
```

Now, a simple expression is either a number on its own or it is two numbers separated by an operator:

```
<simexp> ::= <num> | <num> <operator> <num>
```

Such an expression is the smallest string composed from the symbols which makes sense in this simple language. A general expression can then be built like this:

```
<exp> ::= <simexp> | <lpar> <exp> <rpar>
                   | <exp> <operator> <exp>
```

Actually, the simple expression is redundant and could be omitted by replacing the first option in the production for `<exp>`. In any case, the set of rules above constitute the entire grammar.

The basic idea in the FP-style of parsing is to split-up the parser into a family of *sub-parsers* each corresponding to a term in the grammar, with special *combinators* corresponding to ways of combining these sub-parsers (there will be one for combining them – often just standard function composition in an FP-language – and one for representing options in the BNF grammar), see [2-4]. For a way to implement part of this in another multi-paradigm language, Perl, see [5].

In the functional language Miranda, for instance, one could write a parser for this simple language like this (following [2] and ignoring problems with left-recursive grammars for sake of illustration):

```
exp = (exp $then operator $xthen exp) $alt
      (literal '(' $then exp $xthen
       literal ')') $alt simexp
```

Here, `exp`, `then`, `xthen`, `alt`, `literal` and `simexp` are all sub-parsers (the prefix '`$`' turns any function into an infix operator in Miranda, and brackets around arguments are optional in Miranda as well as in Haskell and ML). The sub-parser `operator` simply parses operators, while the sub-parser `literal` parses the literal string given as the argument, and `alt` is a sub-parser representing alternatives as given by the symbol '`|`' in the BNF grammar. We won't be able to reach the same level of conciseness in this C++ implementation though, but we will try to get as close as possible.

Such parsers may not be as efficient as the ones generated by general high-quality parser-writing tools such as `lex` and `yacc` (and their various relatives such as `flex` and `bison` from GNU), but they have other advantages:

- **Readability**: By splitting the parser up into a number of sub-parsers each corresponding to a term in the BNF grammar, there will be a much closer relationship between the structure of the entire parser and the original BNF grammar.
- **Maintainability**: Since the syntax is fairly straightforward and closely mirrors the corresponding BNF grammar, such parsers are easy to maintain.
- **Flexibility**: The same splitting-up also implies that productions can be added or omitted very easily. Hence, such parsers are extendable. Furthermore, different versions of the various sub-parsers can be tried out.

For simplicity, we will assume that the lexing has been done and resulted in an array of single characters. This is of course a rather trivial lexing (which, moreover, is easy to implement) – most lexers would return not a list of characters but a list of tokens. In order to show the power of the technique, however, it is advantageous to consider such trivial lexers. Such a lexer, for instance, could be implemented by simply reading from a file, one character at a time, returning a list of characters read when done.

This paper will be structured as follows: First, the abstract base class is defined. This is a very basic class skeleton, but will form the foundation of all the richer sub-parsers to be defined later. At the same time we define the basic parse tree class and other related datatypes. Secondly, we introduce some simple general utilities (a Boolean function for testing for digits and two list processing functions found in all functional languages). Next, we define our first generic sub-parsers for numbers and operators. The fourth step is to define general parser combinators, allowing us to combine sub-parsers to generate new types of sub-parsers.

All of these steps are completely general and form a basic parsing library. We then turn to using these general tools to actually parse integer arithmetic expressions. This turns out to be very easy and there is a very, very close relationship between the BNF grammar and the actual implementation of the sub-parsers as promised.

Finally, we discuss various ways to refine the framework.

The basic stream class

We will begin by defining a general *parse stream* or *pstream*. The various sub-parsers will then all be derived from this base class.

3 Floating point numbers could be accommodated with small changes, but this will introduce an unnecessary level of complexity

The parser will need to keep track of a list which is passed on between consecutive parses. Even though, we will only consider lists of single characters here, it is worthwhile to work with a more general set-up, namely that of lists of strings.

Our pstreams will have a state, containing the parse tree constructed so far. In order to be able to pipe pstreams together, thereby building more complex parsers from simple sub-parsers, we will also need the pstream to keep track of the remaining tokens.

Hence, we define a new data type:

```
typedef std::pair<Ptree, List> Presult;
```

where `Ptree` is the class defining parse trees and where `List` is defined by:

```
typedef std::list<std::string> List;
```

which will be our basic data structure. Similarly, `Ptree` is a specialisation of a more general parse tree:

```
typedef ParseTree<std::string> Ptree;
```

The `ParseTree` class is a simple binary tree:

```
template <class T>
class ParseTree {
private:
  ParseTree *left;  // left sub tree
  ParseTree *right; // right sub tree
  T root;
public:
  ... // constructors & destructor
  bool isEmpty() const { return root==T(); }
  bool isLeaf() const
              { return left==0 && right==0; }
  T getRoot() const { return root; }
  ParseTree* getLeft() const { return left; }
  ParseTree* getRight() const
                          { return right; }
  void setLeft(ParseTree& lft) { left = &lft; }
  void setRight(ParseTree& rgh)
                          { right = &rgh; }
  void setRoot(T rt) { root = rt; }
  void update(T); // used for operators
  void insert(T); // used for numbers
  void insert(ParseTree&);
                // used for parentheses
};
```

The member functions `update` and `insert` are there to allow parsers to manipulate the parse tree. The former adds a node as the root, copying the old tree to the left sub-tree, this is to be used when parsing binary operators. The latter inserts a node at the first empty sub-tree it finds. This is used for parsing either numbers or parentheses.

Using just recursion these methods could be implemented simply like this:

```
template<class T>
void ParseTree<T>::update(T val) {
  if(isEmpty())
    throw std::logic_error(
            "Syntax error: Missing operand");
  ParseTree* newLeftSubtree
                      = new ParseTree(*this);
  root = val;
  left = newLeftSubtree;
  right = 0;
}
```

```
template<class T>
void ParseTree<T>::insert(T val) {
  if(isEmpty()) {
    setRoot(val);
    return;
  }
  if(getLeft() == 0) {
    setLeft(*new ParseTree(val));
    return;
  }
  else {
    if(getRight() == 0) {
      setRight(*new ParseTree(val));
      return;
    }
    getRight()->insert(val);
                      // use right recursion
  }
}

template<class T>
void ParseTree<T>::insert(ParseTree<T> &pt) {
  if(isEmpty()) {
    setRoot(pt.getRoot());
    setLeft(*pt.getLeft());
    setRight(*pt.getRight());
    return;
  }
  if(getLeft() == 0) {
    setLeft(pt);
    return;
  }
  else {
    if(getRight() == 0) {
      setRight(pt);
      return;
    }
    getRight()->insert(pt);
                      // use right recursion
  }
}
```

The `if`-statements in the above implementations are the C++ analogue of the argument pattern matching of functional languages[4], they are needed to control the recursive steps and to ensure termination. The code above can be shortened a little bit and most of the `return`-statements can be omitted if one instead uses nested `if`-clauses. While this will make the code a few lines shorter, I think the present coding style has the advantage of clearly showing the reader that the function returns after having handled each special case.

Both methods first handle the case of an empty tree. For `insert`, the value passed simply becomes the new root, whereas for `update` an exception is thrown. Next comes the case of a leaf, i.e., a node without children. Here, `update` inserts the passed value as a new root moving the leaf to the left subtree, while `insert` merely inserts its value as the left subtree rooted at the original leaf node. If the left subtree is occupied, `insert` first tries the right

4 This "pattern matching" is really an advanced dispatch mechanism allowing the user to "overload" functions not just on basis of different *types* of arguments but also on different *values*.

one, if this is also non-empty it uses recursion to find the first empty subtree.

The abstract base class, `pstream` is now:

```
class pstream {
protected:
  Presult pres;  // contents so far
public:
  ... // constructors & virtual destructor
  List const& getList() const
    { return pres.second; } // accessor method
  Presult const& getPres() const
    { return pres; }        // accessor method
  virtual Presult operator<<=
                          (const List &) = 0;
  virtual Presult operator<<=
                          (const Presult&) = 0;
};
```

Notice that only the accessor methods, `getList` and `getPres`, are non-trivial and that the streaming operator, `operator<<=` is a pure virtual function.

Writing the parser

We will now turn to the question of actually writing the parser by splitting it up into a number of sub-parsers as stated in the introduction.

Utility functions

It is advantageous to define a number of utility functions representing the basic ingredients of the grammar. This can be done by defining a few boolean functions like this:

```
bool isDigit(std::string c) {
  return c == "0" || c == "1" || c == "2"
         || c == "3" || c == "4" || c == "5"
         || c == "6" || c == "7" || c == "8"
         || c == "9";
}
```

with a similar function `isOperator` for testing whether a character is an operator. In general, one should write one such Boolean utility function for each type of symbol in the language. Hence, we define two such functions, `isDigit` and `isOperator`. It is, of course, also possible to define a Boolean function testing for brackets but since we only have one type of these it is much easier to simply insert a test `c == "("` or `c == ")"` directly in the code than define special functions `isLPar` and `isRPar` respectively.

Of course, a similar result could be obtained by using built-in functions such as `isdigit` but the advantage of writing them ourselves is the ability to illustrate the general principle.

We will also need a pair of basic list processing functions available in all FP languages. The first is for extracting the *head* of a list, i.e., its first element. This function is called `first`, `car` or `hd` in various functional languages (Common Lisp, old Lisp, and ML respectively). Here we will settle for the name used in Haskell:

```
string head(List ls) {
  return ls.front();
}
```

Similarly, for extracting the *tail* of a list, i.e., the remaining elements, we will write a wrapper around one of the STL list member functions:

```
List tail(List ls) {
  ls2.pop_front();  // remove head
  return ls2;
}
```

The tail function also goes under various names in the FP-community, e.g. `rest`, `cdr` or `tl` in Common Lisp, old Lisp and ML respectively. Once more, we have settled for the Haskell name. Note, by the way, that none of these actually change the list passed to them – we could have declared the arguments `const`, but we would then have to use `const_cast<>` all the time, whenever we want to call them, and this would be rather tedious.

These are the only two list utility functions we will be needing.

The sub-parser for numbers

The entire code for the sub-parser for numbers is very simple, illustrating the ease of the functional-style parsing. It is simply:

```
class pNum : public pstream {
public:
  pNum() {};
  pNum(const List & ls)
       { pres = make_pair(Ptree(),ls); };
  pNum(const Presult &pt) { pres = pt; };
  pNum() {};
  inline Presult operator<<=(const List &);
  inline Presult operator<<=(const Presult &);
};

inline Presult pNum::operator<<=(
                              const List &ls) {
  return operator<<=(make_pair(Ptree(),ls));
}

inline Presult pNum::operator<<=(
                            const Presult &pr) {
  List ls = pr.second;
  if(ls.empty())
    return pr;    // nothing to parse
  std::string c = head(ls);
  if(!isDigit(c)) {
    return pr;    // not a number, do nothing
  }
  else {
    std::string val(c);
    ls = tail(ls);
    while(isDigit(c=head(ls)) && !ls.empty()) {
      val += c;        // construct number
      ls = tail(ls);
    }
    Ptree pt(pr.first);
    pt.insert(val);
    return make_pair(pt,ls);
  }
}
```

This sub-parser illustrates the basic idea of FP-style parsing using streams: To parse a particular expression-type, write a very simple parser (basically just a copy of the abstract base class) and implement the corresponding `operator<<=` for acting upon a reference to a `Presult`-object.

The sub-parser for operators

The sub-parser for operators is almost identical, with the only significant change taking place in `operator<<=`:

```
inline Presult pOp::operator<<=
                        (const Presult &pt) {
  List ls = pt.second;
  if(ls.empty())
    return pt;  // nothing to parse
  std::string c = head(ls);
  if(!isOperator(c))
    return pt;
  Ptree ptt(pt.first);
  ptt.update(c);
  return make_pair(ptt,tail(ls));
}
```

The main difference between parsing numbers and operators is the way the parse tree is updated. Parsing numbers simply inserts the number at the first available empty place, whereas operators have to identify their operands first and hence use the `update` method instead. Numbers are typically leaves while operators are roots of subtrees.

Parser combinators

It is convenient to be able to handle combinations of sub-parsers as well, e.g. to indicate that a particular token must always come after another (*sequencing*) or to indicate a choice between two tokens (*alternatives*). In the BNF grammar, such *combinators* are represented by concatenation and by `|`, respectively.

Combinators are higher-order parsers, taking parsers as arguments and returning new parsers. In typical FP-implementations, in Haskell, say, these would often be represented by currying, i.e., by partial binding of arguments.[5] In C++, however, it is more convenient to define new objects. As an illustration, we will define the following combinators `pAlt`, for handling alternatives, `pThen`, for handling sequencing, and finally `pMore` for handling one or more occurrences of a token. We will also define some syntactic sugar by overloading the operators `||`, && and ++, respectively, to be alternative methods to use these combinators.

The combinators are just parsestreams, although higher-order, hence they are defined by deriving from the abstract base class and by implementing `operator<<=`. Since they have to be able to deal with all kinds of parsers, they are defined using templates. In a sense, one can think of templates as C++'s analogue of such higher order constructs. One might call them "higher order objects".

For instance, the combinator for alternatives can be written like this:

```
template <class T, class S>
class pAlt : public pstream {
private:
  T p1;
  S p2;   // component pstreams
public:
  ... // constructors & destructor
      // & accessors & operator<<=
};
```

This class differs from the previous parsestreams in having two pstreams as internal data members (together with the appropriate accessor methods). These are just defined, for simplicity, as general classes of types `T` and `S` respectively, relying on compile-time polymorphism to ensure that these can be used as pstreams.

To actually be able to use this combinator, we must implement `operator<<=`. This can be done as follows:

```
template<class T, class S>
inline pAlt<T, S> operator||
                  (const T& p1, const S& p2) {
  return pAlt<T, S>(p1,p2);
   // alternative way of creating pAlt objects
}
// definition of how pAlt works is given
// by operator<<=
template<class T, class S>
inline Presult pAlt<T, S>::operator<<=
                       (const Presult& pr) {
  Presult lp1(p1 <<= pr), lp2(p2 <<= pr);
                    // use component parsers
  List l1=lp1.second, l2=lp2.second;
  return (l1.size() <= l2.size())? lp1 : lp2;
                    // return best parse
}
```

where we have also taken the opportunity to add some syntactic sugar by providing `p1 || p2` as an alternative syntax for constructing a `pAlt`-object out of two sub-parsers.

The implementation of `operator<<=` is a bit naive, but will suffice for the present case. In general, one may need a more complicated criterion for "best parse" than simply returning the parse resulting in the smallest list of remaining tokens, but in our case it will suffice.

Similarly, the code for `pThen`, the sequencing combinator, is:

```
template<class T, class S>
class pThen : public pstream {
private:
  T p1;
  S p2;   // component parsers
public:
  ... // usual stuff
};

template<class T, class S>
inline pThen<T, S> operator&&
                (const T& p1, const S& p2) {
  return pThen<T,S>(p1,p2);
                       // alternative syntax
}

template<class T, class S>
inline Presult pThen<T, S>::operator<<=
                       (const Presult& pr) {
  return p2 <<= p1 <<= pr;
}
```

Finally, the combinator for one or more occurrences of a token, which would be expressed by a regular expression in BNF, `(token)+`, can be written as:

```
template<class T>
class pMore : public pstream {
private:
  T p;   // component parser
public:
  ... // usual stuff
};
```

5 C++ can also do this in a limited way using `bind1st` and `bind2nd` on function objects. These are available in the `<functional>` header file and provide one more example of FP-concepts being introduced into C++. See e.g. [7]

```
template<class T>
inline Presult pMore<T>::operator<<=
                        (const Presult& pr) {
  Presult pr2(p <<= pr);
  List ls  = pr.second;
  List ls2 = pr2.second;
  while(!ls2.empty() && (ls2.size()
       < ls.size())) { // something was parsed
    pr2 = (p <<= pr2);   // continue parsing
    ls  = ls2;
    ls2 = pr2.second;
  }
  return pr2;
}
template<class T>
inline pMore<T> operator++(const T pp) {
  return pMore<T>(pp);
}
```

Here, we have settled for a straightforward "procedural" implementation, one could also have used recursion (as one would almost certainly have done in FP) in the definition of `operator<<=`.

In a sense, these combinators together with a parser for literal substrings, `pLit`, which we haven't given but which is almost identical to `pNum` and `pOp` earlier, are all we need to parse anything. Parsing numbers, for instance, could symbolically be written as:

```
pNum = ++(pLit("0") || pLit("1") || pLit("2")
        || pLit("3") || pLit("4") || pLit("5")
        || pLit("6") || pLit("7") || pLit("8")
        || pLit("9"));
```

which is more or less what one would have written in a functional language such as Haskell or Miranda. C++, however, does not allow one to define new classes from old ones in this manner, hence one would have to put this definition of `pNum` into the implementation of `operator<<=`. In the next section we will see that this is precisely what we will do to parse general expressions.

There is one more reason, however, one cannot simply use the above trick to write `pNum`. Parsing numbers, as well as operators, one needs to perform certain *actions* (inserting or updating the parse tree). We will later see how to associate actions to parsers, but for now we will have to make do with custom-written parsers `pNum` and `pOp` explicitly taking care of the necessary parse tree manipulations.

The sub-parser for bracketed expressions

This is slightly more complicated. It turns out to be advantageous to consider bracketing as a kind of parser combinator. Hence, given a parse stream `p` we will define a new pstream type just like we did for `pAlt` and the other combinators. This, by the way, is one way in which our implementation differs from the normal functional approach.

The definition of the bracketing combinator, `pBrack`, is very similar to the previous combinators:

```
template<class T>
class pBrack : public pstream {
private:
  T p;  // internal sub-parser
  Presult pr2;
public:
  ... // usual stuff
};
```

As usual, all of the work is done in the overloaded `operator<<=`:

```
template<class T>
inline Presult pBrack<T>::operator<<=
                        (const Presult &pr) {
  if( (pr.second).empty() )
    throw logic_error("Syntax error.
                Closing bracket expected!");
  std::string c = head(pr.second);
  if(c == "(") {
    Presult prr( p <<= make_pair
                (pr.first, tail(pr.second)) );
                // apply internal parser
    (pr2.first).insert(prr.first);
                // update internal parse tree
    return operator<<=
          (make_pair(pr.first, prr.second));
  }
  if(c == ")") {
    Ptree pt(pr.first);
    pt.insert(pr2.first);
                  // insert parsed subtree
    return make_pair(pt,tail(pr.second));
        //skip closing bracket & terminate recursion
  }
  return pr;       // do nothing
}
```

The only subtlety is in maintaining an internal, temporary parse tree. Upon returning, this internal parse tree will hold the parse tree for the entire bracketed expression, which can then be inserted into the full parse tree. This pstream mimics the way human beings often read parenthesised expressions – one sees an opening bracket and immediately one begins to parse the internal expression until one sees a matching end bracket. If nested brackets are found, one resorts to recursion.

Putting it all together

We now have all the ingredients needed for parsing integer arithmetic expressions. One could put the previous classes and functions into general header files to be used for all parsers, and then proceed to write parsers for the specific language at hand, which is what we will turn to now.

Now, the grammar is recursive and while C++ is certainly capable of handling recursion (in fact, we have used it frequently in this paper), the language is not really capable of handling recursive definitions such as the grammar in its present state. Hence, we will have to introduce an extra layer of indirection. Rewrite the grammar as:

```
<aexp> ::= <num>  | <num> <op> <num>
<sexp> ::= <aexp> | <lpar> <aexp> <rpar>
<exp>  ::= <sexp> | <lpar> <sexp> <rpar>
           | <sexp> <op> <sexp>
```

The idea is now simply to write three parsers `pAExp`, `pSExp` and `pExp`. These classes are completely trivial, with all the work being done in `operator<<=` as usual.

The first sub-parser is:

```
inline Presult pAExp::operator<<=
                        (const Presult &pr) {
  pOp po;
  pNum pn;
  return pn || (pn && po && pn) <<= pr;
}
```

Notice the simple expression in the last line. It is a simple, faithful reflection of the definition of the corresponding term in the BNF grammar.

The remaining sub-parser, `pSExp`, and final parser, `pExp`, are very simple and only their non-trivial `operator<<=` function will be given.

For `pSExp` the code is simply:

```
inline Presult pSExp::operator<<=
                          (const Presult &pr) {
  pAExp pa;
  return pa || pBrack<pAExp>(pa) <<= pr;
}
```

Which, once more, is a faithful representation of the corresponding definition in the BNF grammar.

Finally, the full parser, parsing general integer arithmetic expressions is simply:

```
inline Presult pExp::operator<<=
                          (const Presult &pr) {
  pSExp ps;
  pOp po;
  return ps || pBrack<pSExp>(ps)
            || (ps && po && ps) <<= pr;
}
```

Hence, with the generic sub-parser tools defined, writing a functional style parser using pstreams is an easy task, which makes it well suited for language experiments and for rapid prototyping.

With these definitions in place, to parse a list of characters, `ls`, using this parser, one simply writes:

```
pExp pe;
result = pe <<= ls;
```

where `result` is of type `Presult`. For a complete parse, this would contain a parse tree as first component and an empty list as second.

Refinements

The previous sections have shown one way to implement functional-style parsing in C++ using a generalisation of streams together with operator overloading. Incidentally, the heavy usage of operator overloading shows one advantage of C++ over a language like Java that does not support the overloading of operators. Most of the above could also be carried out in Java, but one would then have to use functions instead of operators resulting in harder to read code. Java also lacks C++'s features for generic programming (the templates), although a library providing some of this support is available. Some of the same effects could be mimicked in Java, however, by judicious use of the universal base class `Object` and frequent casting. Such an approach will not be as easy to read and will, moreover, be more error prone than the one presented here. Although a direct translation of the above C++ code into Java would not be satisfactory, one could instead use Java Beans – in some sense, these are able to model a behaviour similar to that of functions in a functional language.

On the other hand, C++ is not perfect in this respect either, since it lacks the possibility of user defined operators as well as a method to specify the associativity of an operator and whether it is to be applied as infix, postfix or prefix. Such possibilities are often available in FP languages, e.g. in ML and Haskell, and they are also likely to be available in future versions of existing languages such as Perl6.

The necessity of writing the execution from right to left lies in the standard OO-convention that `y<<=f` is really short for `y.operator<<=(f)` and hence treats the left hand operand differently from the right hand one. This is a consequence of OOP's dispatching rules. If one wanted to make left-to-right parsing work instead, one would have to define, inside *all* classes, methods for handling the different parsestreams. For instance, one would have to define `List::operator>>=(T &p)`, where the typename `T` can be any valid parsestream subclass. This could of course be done with templates much the same way as was done for the combinators, but it would also necessitate the writing of a wrapper class for the List-object in order to be able to extend it this way. A much cleaner solution would be to use *multi-methods*. However, unlike Lisp's CLOS, these are not directly available in C++. There are ways around this, of course, C++ being after all a very flexible language, [6].

A problem not addressed at all in this paper is the problem of optimisation. Clearly, the parsestreams as developed here are optimised neither for speed of execution nor for space, but rather for *speed of definition*. By this phrase, it is to be understood that the method is intended for rapid prototyping and for experimenting with language features. Although the program isn't slow, it would certainly be advantageous to have the parsestreams run faster in real-life applications. One simple way of doing this is to pass a tree-iterator around keeping track of where the last insert/update of the parsetree occurred, for instance by pointing to the first empty subtree. This would speed up the `insert` and `update` operations.

Handling ambiguity and precedence

For simplicity, we did not consider operator precedence in the example above, i.e., `1+2*3` will be parsed as `(1+2)*3` and not `1+(2*3)`. Of course, this could always be enforced by appropriate use of parentheses, but it would clearly be advantageous to conform closer to the likely expectations of the end-user.

It is well-known that operator precedence can be handled by slightly modifying the grammar (see e.g. [1]):

```
<expr>   ::= <term> + <term> | <term> - <term>
                             | <term>
<term>   ::= <factor> * <factor>
           | <factor> / <factor> | <factor>
<factor> ::= <number> | ( <expr> )
```

Thus, at the cost of adding new terms to the grammar, one can handle operator precedence in the expected way. It is trivial to change our family of sub-parsers to accommodate this, in particular since the recursive definitions have been removed in the same step.

Our parser stream framework worked well for the simple case of basic arithmetic expressions, but a general grammar is likely to be ambiguous with the parser effectively having to make certain choices at various stages of the parsing. Clearly, it would be of great interest to be able to handle this case too.

The standard way this is dealt with in FP is to replace the `Presult` data type with another one, [2-4].

In the face of ambiguity, what we have to deal with is not a single, unique parse, but rather a family of *possible parses*. Hence, the proper way of dealing with ambiguity is to introduce a new data type, let's call it `LPresult` (for List of Parse results). This is defined as:

```
typedef std::list< Presult > LPresult;
```

All the parsers must then return this data type instead. This, of course, necessitates some modifications. Each sub-parser must now act on *all* the elements of the list of possible parses. Consequently, the individual parse trees will have different sizes, in general with the largest one corresponding to an empty list of remaining tokens, and thus to a complete parse.

With these quite simple changes, our parse stream framework will be able to deal with ambiguous grammars as well.

Adding actions

The FP-style of parsing discussed in this paper opens up further modifications. In the arithmetic expression example, we saw how one of the types (the operators) had to perform some non-trivial operations on the parse tree.

This can be generalised. Instead of just allowing the simple manipulations involved in parsing binary operators and parentheses, one could allow more general actions to be associated with steps in a parse.

Various examples could be:

a) a pretty printer, printing out the code in a nicely formatted manner as the parse goes along;
b) cross-translation, translating each element of a parse into some code in another language, in the same way as `yacc` and similar tools do;
c) execution of the result as it goes along, in the case of arithmetic expressions this would be one very natural action to add, effectively turning the parser into a calculator.

To associate an action with a step in a parse, i.e., with a particular sub-parser, we need two things. First of all, we need a function to perform. This is done by using a function object, as this is the best way of passing around function definitions in C++. Secondly, we need an operator associating an action with a given sub-parser. This latter step will be done by overloading the `>>=` operator. Hence, given a sub-parser `p` and a functor `f`, we will define `p>>=f` to be the sub-parser `p` with the actions given by `f` added to it.[6] This is consistent with interpreting the sub-parsers as streams: the output of one pstream is sent to the associated function object for further processing.

The definition of the corresponding sub-parser type, `pUse`, is just like the definitions of all the other combinators. Explicitly[7]:

```
template<class T, class S>
class pUse : public pstream {
private:
  T p;     // component parser
  S f;     // function to apply
public:
  ... // usual stuff
};
template<class T, class S>
inline Presult pUse<T,S>::operator<<=(const
Presult& pr) {
  Presult pr2(p <<= pr);  // apply parser
  return f(pr2);  // apply function – MUST
return Presult
}
```

```
template<class T, class S>
inline pUse<T,S> operator>>=
                (const T &pp, const S &ff) {
  return pUse<T,S>(pp,ff);
                        // alternative syntax
}
```

To use this, one must then define a function object. For instance:

```
class printer {
public:
  Presult operator() (Presult &pr) const {
    std::cout << "Parse tree: ";
    (pr.first).print();
    std::cout << std::endl << " and ";
    prList(pr.second);
    return pr;
  }
};
```

assuming that a `print` method has been added to the `ParseTree` class and that the function `prList` prints a list in some appropriate format.

Let `ls` be a list of characters, one can then write:

```
pUse<pExp,printer> pu(pe,prn);
pu <<= ls;
```

where `pe`, `prn` are instances of `pExp` and `printer` respectively. This would then print the parse tree together with the list of unparsed characters. In fact, precisely such a construction was used during the test phase of developing the programs presented here.

Alternatively, one could write:

```
(pe >>= prn) <<= ls;
```

leaving the creation of the proper classes to C++. This latter syntax is probably as close as one will be able to get to the FP-syntax in C++.

Similarly, one could define a function object, `executor`, which executes the code found in the parse tree.

Finally, once pUse has been defined, one could redefine `pNum` and `pOp` in terms of `it` and `pLit`. The former would have to extract the inserted individual digits, combine them to form a number and re-insert this at the proper place in the parse tree, while the latter would have to extract the operator and call `update`.

Conclusion

We were able to extend the functional-style parsing using sub-parsers to C++ by defining a generalised class of streams, called *pstreams*. With this, we could define *combinators* allowing us to build new sub-parsers by combining old ones. With these general tools out of the way, it turned out to be very easy, once recursion had been handled properly, to implement the BNF grammar for integer arithmetic expressions. Moreover, similarly to the situation in FP, there was a very intimate relationship, in fact more or less an isomorphism, between the actual BNF production rule and the corresponding sub-parser, making it very easy to write such sub-parsers – it would even be feasible to write a general sub-parser generator to do so automatically.

It was also shown how to handle operator precedence and, perhaps even more importantly, ambiguous grammars. None of this involved major changes to the framework, thus the proposed framework scales well to more complicated grammars such as those of typical programming languages.

[concluded at foot of next page]

6 In monadic parsing in Haskell one often uses the sequencing operator `>>=` for precisely this purpose, hence we will use that here too.

7 In most FP languages one would say that `f` must be of type `Presult -> Presult`.

EuroPLoP 2003 Conference Report

by Allan Kelly

Exciting. Tiring. Fun. Intoxicating. Mind stretching. Brilliant. Just a few words to describe EuroPLoP 2003, the annual European conference on Pattern Languages of Programming held in Germany during the last week of June. Although, I'm not completely sure 'conference' is the right word, nor does the gathering confine itself to programming patterns. I'll try and give you an idea of what happened somewhere in Bavaria

Where is it?

The venue, as always with EuroPLoP, is Kloster Isree, a former monastery turned hotel and conference venue about 100km west of Munich. The hotel is set in beautiful countryside outside a small town – not that you have much time to explore the town or the countryside, but it adds to the general feeling of calm. The conference is actually something of a retreat, albeit a retreat which involves a lot of hard work (and, erh, beer.)

Full price this year was €900, but this included the conference, accommodation, meals, refreshments and as much beer as you can drink – or any other liquid refreshment you may prefer. As if all this didn't make for enough of a bargain the organisers threw in a brightly coloured yoyo. Who could ask for more?

Who goes to EuroPLoP?

Since everything is included the conference really starts over breakfast where conversation quickly turns to the conference itself and all matters patterns. This continues through lunch and dinner down to the bar and into the small hours of the night. Only interrupted by traditional conference drinking songs. And this is a conference with traditions carried on by a core of regular attendees.

For the conference newbie this core of regulars could seem off putting. But everyone is very friendly and by the end of the first day this feeling is disappearing, is gone by the end of the second, and by the time the conference ends you feel like you're part of an extended family.

In total there were 65 people at this year's conference. Unsurprisingly the majority of attendees were German but the second largest group (14) people, were from the UK. Add to this a couple of New Zealanders, half a dozen Americans, 5-6 Scandinavians and another dozen from assorted other European countries.

These seem to be split in equal thirds between academics, independent consultants and regular employees. The academics have a particular problem to wrestle with when it comes to patterns, that is, academia places a particular emphasis on original work, yet the very essence of patterns is that they document known solutions to problems.

The small scale and interactive nature of the conference means that by the time it comes to leave you have 64 new friends. (Well, in my case a few less as about 8-9 people are also to be seen at ACCU conferences.)

At most conferences the speakers list is one of the attractions to pull in the crowds, providing the opportunity to hear known speakers address a topic. EuroPLoP doesn't have any speakers, no big names, instead there are authors, and most of the attendees are themselves pattern authors. And rather than receiving a speakers list before the conference, you leave with a participants list. This leaves me thinking the conference is kind of upside-down.

What do you do?

Like any conference EuroPLoP is split into sessions, punctuated by meals and coffee breaks. The sessions though are split into Workshops and Focus Groups – the former in the morning and the latter in the afternoon.

Workshops are an opportunity for workshop members to review the patterns presented by the members of the group. These have been submitted and shepherded in advance so are already of high quality. The author introduces their paper then steps back, becoming a fly on the wall while the group discuss the paper. Only at the end is the author allowed to return and talk, and then he/she is only allowed to ask for clarification, they do not engage in defence of the paper.

The objective is to improve the paper. Over the months following the conference the author incorporates those suggestions they wish to into their paper. The revised paper is then resubmitted to be included in the conference proceedings. This is another way in which the conference is upside-down, the papers which make it into the proceedings have been changed from the papers presented.

I'm glad to say that my paper, a pattern entitled Encapsulate Execution Context, was well received by my group. However, when they turned their attention to improvements it can be most frustrating as the group discusses changes you have already wrestled with, or different group members contradict one another. Receiving feedback, even that meant positively, can be a bruising experience. Only later when I had a chance to write up my notes and reflect on the experience could I honestly say it was all positive.

Most of the afternoon is taken up with focus groups. The format of each group differs depending on what the workshop leader(s) wish to achieve. For example, one of this year's groups worked with Lego Mindstorms to build robots. The group leader's intention

---

[continued from previous page]

Moreover, we showed how one could associate actions to individual sub-parsers thereby dramatically extending the possibilities of stream-based parsing. The associated actions could be used to create a pretty printer, or even to translate or execute the code.

Frank Antonsen
frankantonsen@netscape.net

References

[1] J. C. Martin, *Introduction to languages and the theory of computation, 2nd ed*, McGraw-Hill (1997).

[2] G. Hutton, "Higher-order functions for parsing", *J. Funct. Prog. 2 (3)*, p323 (1992).

[3] J. Fokker, "Functional parsers", *Lect. Notes of the Baastad Spring School on Funct. Prog.* (1995).

[4] S. L. Peyton-Jones, D. Lester, *Implementing functional languages. A tutorial*, Prentice-Hall (1992).

[5] F. Antonsen, "Functional programming in Perl", to appear in *The Perl Review.*

[6] A. Alexandrescu *Modern C++ Design : Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

[7] N. M. Josuttis, *The C++ Standard Library. A tutorial and reference*, Addison-Wesley, 1999.

was to investigate the pattern discovery process by looking for patterns in robot construction.

Another group discussed team working and practices for human interaction in groups. Again the objective was pattern discovery. This meant working in small teams and discussing what we do in the work environment and looking for documentable practices.

In addition to workshops and focus groups there are a variety of other activities such as writers' groups and birds-of-a-feather sessions. The third night sees a grand banquet during which prizes are awarded, some serious, some humorous.

I've heard you play games at EuroPLoP?

Something which marks out EuroPLoP from your average conference, and even your not so average conference, is the presence of games and art. The conference has a resident artist who runs an art studio and organises games at several breaks during the day. The games are non-competitive and not necessarily physically demanding – although some attendees made a decision not to offer themselves for stage diving!

Apart from having some good fun there is a serious intent behind the games. Giving feedback to people can be difficult, and it can be more difficult to hear people talk openly about one's own work. However, it is hard to take any of this personally when the person giving it was sitting on your knees last night.

There is also a lot of humour at EuroPLoP. This occurs inside meetings where there is a very relaxed, upbeat atmosphere, in the drinking songs and in the conference's own daily magazine. (Although Overload readers may have felt strangely at home with a magazine edited and largely written by two regular Overload contributors.)

The games also add to the sense of "getting to know you" that breeds trust and creates a sense of community between the people there. In a sense, the conference didn't end when everyone went home, it goes on, each of us is part of something bigger than ourselves that will continue to evolve.

Where are patterns going?

If anyone still thinks Patterns equals the Gang of Four (Gamma et al, 1994) book and "Pattern Languages of Programming" (PLoP) means the conference concerns itself with just programming or IT matters, now is the time to wake up and smell the Bavarian Beer.

In fact, I think the pattern I presented was one of the most GoF like as it concerned itself with a common high level programming technique. Other patterns in my workshop dealt with embedded systems – giving their example in assembler code – or techniques for using Aspect Oriented Programming. Other workshops looked at pedagogical patterns, use case patterns, pattern writing, leadership patterns, and even patterns of shepherding patterns.

I detected three trends in patterns in the papers presented and the conversations about patterns:

- **Pushing the boundaries:** pattern writers are starting to explore the boundaries of what patterns can do and where they can be used. As already mentioned academia has problems with patterns, however, this is not stopping some academics from trying to use them and research with them. A recurring theme was the use of patterns as a form of knowledge management.
- **Application to new domains:** software people may have adopted patterns from architects but they have been more successful than architects in spreading the word. Fields with immediate relevance to software are now starting to experiment with patterns, educators are starting to write pedagogical patterns, while the IT security community is attempting to frame much of their work as patterns.
- **Division of patterns:** another recurring theme is the correct "level" or "audience" for patterns. Some people are exploring how we may group patterns into hierarchies, so we may have abstract patterns at the top, with other implementation patterns forming a second layer of concrete patterns. For example, an abstract pattern may describe how to implement a scheduler, while a concrete pattern would extend this to techniques on DOS based computers. There is also a debate as to whether this kind of information is best presented as a concrete pattern or as a case study.

  Others are interested in how to present patterns to different audiences. A format and content that is great for an inexperienced developer may not work so well for a battle hardened veteran. Even one's own demands on a pattern may change when the document moves from being an introduction to being a reference. What is the solution? Multiple styles? Hypertext?

What is increasingly clear is that patterns can lend a more human dimension to technical literature. This may occur directly, through patterns about human behaviour, or through the presentation of highly technical information in more accessible formats. Either way, the greater emphasis on people makes patterns a useful knowledge management tool.

(Now I come to think about it, I seem to recall Jim Coplien, either in print or more likely at an ACCU conference talking about Christopher Alexander's reaction to software patterns. If I recall correctly, he thought that the software community wasn't paying enough attention to the human aspects of patterns. Maybe the software patterns community is now addressing this, or maybe Alexander didn't realise that by computer industry norms, the patterns community does appreciate people more.)

What else?

What else can I say about EuroPLoP? I got home and felt as if I had been running for three days, physically it was very tiring. This was not just from the games and beer, the conference maintains a very high level of intellectual activity. My body may have been exhausted but my mind felt like it had been given a workout in a mental gym.

Much of the credit must rest with the conference regulars who form the pattern community, and in particular the European patterns community centred on Hillside Europe. This is a community with a noticeable ACCU overlap, I counted about 10 participants had been to one or other of the ACCU conferences – most notably the programme was chaired by our own Kevlin Henney.

Different conferences fulfil different roles. Academic conferences may be little more than presenting papers. Commercial conferences may be glorified training sessions. For me EuroPLoP was about two things. Firstly it was about contributing to the growing body, and secondly it was about growing as a person and opening myself to some new ideas.

Would I recommend it? Yes with one reservation: if you are going, be prepared to be open, this is not a conference for those with fixed ideas, fears or a point to prove. It is a conference where you give and you receive, and like Christmas, much of the pleasure comes from the giving.

Allan Kelly

Three Phantastic Tales

by Alan Griffiths

When people work together (and most software development involves people working together) they are often not pulling in the same direction. When you notice that others are pulling in a different direction it is natural to assume that they are the cause of the problem. After all you know that you are not doing anything stupid. But in reality the behaviour of your colleagues isn't stupid, it is just strange, because you don't understand what they are trying to achieve. And you can't fix what you don't understand.

Strange behaviour requires explanation, and the form the explanation takes reflects the prevailing cultural context. Behaviourists will talk of "conditioned responses", psychologists of "archetypes", evolutionists of "memes", and I'm going to talk of *ghosts*. On one level these particular ghosts are a narrative device but, on another, they are very real and pose a danger to any project that is visited by them. Anyone experienced in software development will recognise the spirits in the stories that follow. I have met them many times with many names but, to protect the people who have been possessed by them, I have chosen to use names that reflect their essence.

These shades are Mr Deadline, Seymour Checks and Noah Shortcut; the first of these is a project manager and the others are developers. Each of them contributes to the failure of a project, although each is working in a way that is a rational response to the way they see events unfolding.

Mr Deadline's tale

Like most project managers Mr Deadline has a lot of demands on his time. He keeps the customer and management informed and contented with progress of the project. He ensures that the equipment, software and developers required for the project are available when they are needed. And he ensures that work is allocated to and completed by developers.

With so many demands upon his time he seeks simple strategies for satisfying them: create a plan against which he marks off progress, predicts when resources will be needed, and records the allocation and completion of development tasks.

At the beginning of the project he gives out the first tasks to Seymour Checks and Noah Shortcut and, happily, both tasks are completed on time. But, as the project progresses, he finds that although Noah continues to complete his work on time Seymour takes longer and longer to complete his work and the project falls behind the planned schedule.

Adding developers to the project helps a little, but none of them are as productive as Noah Shortcut (or as slow as Seymour). Mr Deadline ensures that all the critical elements of the system are completed on time by giving them to Noah, while anything less urgent (or unplanned) is passed to the less reliable developers (Seymour and the others).

The project continues to fall further behind schedule and, additionally, some serious bugs are detected during testing and acceptance testing leading to significant delays through rework. Eventually, the planned delivery date is reached without all the work being completed to an acceptable standard. The project has failed to deliver (and, because of the extra staff, is also over budget).

Sometimes a project will be cancelled at this point, but in this case the project continues. Mr Deadline is required to fix the problems as soon as possible but also comes under pressure to release staff to the next project. In the hope of avoiding a repeat of these problems the next project is staffed with the most productive developers. After a while Seymour is left as the sole developer on the project to fix the remaining problems.

Eventually, all the problems are fixed and the project brought to an (unsuccessful) conclusion.

Noah Shortcut's tale

Noah is bright, eager and understands the need to minimise the amount of time and effort spent on the project. From the moment he receives his first piece of work he is trying to avoid any activity that would delay the delivery of that work. When he looks through the documentation for that work there may be a few things that he doesn't fully understand, but he can see what classes and functions he needs to write – which is all he needs to start cutting code.

Once the code is done the job is almost over – he just needs to integrate it and check that it is working. He spends some time exercising his code through the debugger "to make sure it works", fixing any problems he encounters, and he can soon announce that he's finished.

As time passes his confidence grows: he always finishes on time and is trusted with all the important new stuff to write. He's also very aware that the project is behind schedule – and he does his best to catch up by finding new shortcuts through the project processes. In particular, he finds that he can reduce the time spent checking his work: if the testers find problems he can fix them easy enough; and, since they don't find many, this approach avoids a duplication of effort.

When the last piece of functionality is handed over Noah feels a sense of triumph – there are probably a few bugs to fix, but the hard part is over. And, in recognition of this achievement, Noah and the other great programmers are moved onto the next project to work their magic there!

Seymour Checks' tale

Despite the impression one may gain from Mr Deadline's story, Seymour writes reliable code quickly. Why then does he take so long completing his tasks?

When Seymour receives his first piece of work he reads through the documentation and makes notes on anything that isn't clear and on how he will prove the code that he writes (to be specific, he does this by writing automated tests). Then he seeks clarification on all the issues, writes the code and checks it works (by running his tests) before announcing he's finished.

As the project progresses he finds that more and more of his work relies on existing code. Where this is code Seymour wrote himself it is clear what the code should be doing and there are tests that demonstrate that this is indeed what it does. Where another developer wrote the code this is not the case and it is frequently unclear whether the code achieves its intent. At first Seymour assumes that his colleagues have validated their code. But after repeatedly finding that his code is failing because of errors in the existing codebase Seymour becomes disillusioned with the slapdash work of his colleagues.

Because in addition to the work assigned to him, Seymour is fixing problems in existing code, he begins to fall behind Mr Deadline's schedule. Seymour is conscious of these delays and especially of the length of time it takes to prove that the problem isn't in his new code and to locate it. So, in an effort to find and correct these problems effectively, he takes to writing tests for any

existing code he uses that is missing tests and fixing the problems he discovers. However, as more and more code is added to the project (and as changes are made to the production code without updating the tests) the effort of doing this leads to an even greater overhead to Seymour's activities.

Only when the codebase in the project begins to stabilise (because no more features are being added and developers are leaving the project) does it become possible for Seymour to make progress in addressing the many bugs hidden in the codebase.

Seymour is the last developer on the project: the hero tracking down and fixing the problems that others left unresolved. Eventually he succeeds: the system reaches an acceptable standard and work on the project is brought to a close.

Why does the project fail?

Clearly the above tales are different views of the same failing project, and each of the tales describes someone who is doing their best to ensure that the project succeeds. There is no evil villain plotting to prevent the success of the project, nor anyone doing anything that is obviously stupid at an individual level. The problem lies in the interaction between individuals – our spirits do not consider the effect that their actions have on other project members or the project as a whole:

- Mr Deadline attempts to speed up the project by getting each piece of work done as fast as possible. But the pressure that he places on Noah and Seymour promotes hidden rework and this slows down the project. He cannot see this without understanding the dynamics of the project as a whole. Unless he realises that he is part of the problem he will resist changing his behaviour.
- Similar comments apply to Noah Shortcut who is going as quickly as he can but, in doing so, produces careless work that (sooner or later) needs rework and so delays the progress of the project. Once again, unless the connection is made apparent then he will continue to focus on speed to the detriment of progress.
- By now you should see the pattern: Seymour Checks' effort to remove the bugs conceals the level of rework and prevents it being recognised as a problem. But, without an appreciation that this is happening, he won't change his approach.

One reason that I've discussed these stories together is that these spirits travel together. Once a team member succumbs to one of the spirits they (unintentionally) encourage the other behaviours. This point is important when effecting an exorcism, so we will examine this mechanism more closely now.

By not enforcing an adequate quality check on the work done Mr Deadline creates an environment that encourages cutting corners. The spirit of Noah Shortcut will soon possess anyone who looks for the simplest way to complete a task. The opportunity to cut corners also affects Seymour Checks – although his self-discipline is sufficient to keep him from shortcuts it also tempts him to the opposite excess (introducing redundant tests). Mr Deadline also fails to ensure that rework is recognised as a continuation of the original task; this creates an environment that encourages Seymour to "just go ahead and fix it" and keeps from Noah an awareness of the cost of his carelessness.

Noah Shortcut's need for speed will lead him to skimp on any quality checks included in the project process (review meetings, tests or whatever) and to discourage any such "time wasting" procedures. Mr Deadline is eager to speed things up and will listen sympathetically to ideas that will "save time". At the same time Noah is leaving a trail of careless mistakes through the codebase – while each time Seymour is tripped up by one of them he becomes more determined to root them all out.

Seymour Checks is keeping key information to himself: the cost of the time spent fixing other people's mistakes. To him it is reasonable: by the time he's found the bug, it is quicker and easier to fix it than to explain it to someone else (who is probably in the middle of something important). But if Mr Deadline isn't aware that rework is happening (and that it is mainly in work produced by Noah) then he will assume that the code is of an adequate quality. Equally, if Noah isn't made aware that he is making mistakes he will not try to rectify them.

How to fix it

The first step is to be sure that what is going on really matches the tales I've told. Not every project manager is Mr Deadline, not every quick programmer is Noah Shortcut and not every slow programmer is Seymour Checks. But they are easy to recognise once you know their characteristic behaviours.

Now, it is pretty well known that if people are to be changed they must first want to change. And unless your colleagues (or you) realise that they are possessed by one of these three spirits then they won't take any steps to exorcise them. Accordingly the next step is to explain to them just what is going on. This isn't easy because, as the tales illustrate, each of them is already doing his best according to his understanding. The stories are a useful device to sidestep the tar pit of telling people they are doing something wrong. It is human nature to react defensively to such a confrontational approach (which allows the ghosts to entrench themselves). The stories are much less threatening – so feel free to use them to present the case for change.

Until I came up with these stories I struggled to get the necessary ideas across. Naturally the stories are not enough: you must tell them at the right time and have evidence to show that they apply to the current project. For example: you might find occasion to tell your project manager how substandard work products can impact downstream tasks and use the story of Noah Shortcut to illustrate that his fastest programmer could be a liability. He won't know if this is really what is happening on his project, so you also need evidence that Noah's shortcuts are costing other people on the project time. In one recent case I'd just told this tale to the project manager when the integration test team complained that it had been stalled for two hours trying to compile the system. A little investigation showed that Noah had checked in a change without first building the system properly. (This led to a tightening of procedures and a willingness to consider the other tales.)

There are no guarantees: the battle isn't over once there is agreement that there is a problem – habits do not change easily. The ghosts will still put up a fight! You will need to have an answer to the argument that "there isn't anything we can do about it". The different spirits have different ways of putting it:

- Mr Deadline will tell you that "I can't tell if a piece of code works until it is tested. And, after a bug is found in testing and someone has tracked it down to the right piece of code, it would take too long to give it back to the original developer."
- Noah will see most suggestions as a waste of time: "there are always bugs regardless of any checking process – so why knock yourself out trying to eliminate them?"
- Seymour will tell you that "I've already done most of the work tracking down the problem, people don't want to do anything if I do tell them and I don't want to make a big deal of it."

[concluded at foot of next page]

A Unified Singleton Framework

by Jeff Daudel

Introduction

Software systems often contain objects that exist for the duration of the program. A relatively small system may have only a handful of these objects, where a large system could have hundreds. At first glance, maintaining these objects, including their creation and destruction order, may not seem too difficult. However, closer inspection reveals the true magnitude of the complexity involved. Only ten such objects have over 3.6 million different orderings; one hundred objects have a number of orderings that is represented with 157 zeros. Now consider periodically adding or modifying the relationship of these objects. In a system where core systems and even pieces in those systems are built with these objects, it would not be unreasonable for a system to have many ongoing changes, even if it were well-designed. Accounting for all these factors, maintenance of both the proper orderings and all the dependencies can quickly become a nightmare.

Many developers attempt to handle these objects by hand and encounter difficulties. Others conclude that large C++ systems cannot be written without automated garbage collection, and I would say that the belief has a lot of credit given the numbers they are up against. But garbage collection can be costly. The run-time and memory overhead, coupled with its complexity, make it unacceptable for many applications. And I would have to ask, "Is this the best we can do to handle such a critical problem?"

Additionally, let me add that this may not even be an isolated problem, but only one of several unaddressed issues of a common design pattern. Similar problems exist and are more prevalent as a system grows. This design pattern is a powerful tool to build complex C++ systems, but it also might have a few gaping holes in its support.

The pattern I am referring to is the singleton. I believe it is one of the most fundamental patterns for a software foundation. If a system consisted purely of objects, singletons might be the only reason to have the word "the" in a programmer's vocabulary. Anyone that ever said, "the manager", "the renderer", or "the startup state" would be referring to a specific singleton object. But just because something has a lot of potential, does not mean a gloomier side doesn't exist. Unfortunately in software, the gloomier side is usually in the implementation details. The singleton pattern is no exception.

In this paper, I will outline the classical singleton pattern in more detail. I will present a generic system that will not only provide the basic capabilities of the singleton pattern, but will provide several other useful abilities. It will extend the notion beyond the current scope of a singleton to provide answers in a larger context of problems. I will discuss the common problems that occur when implementing the pattern in the C++ language. I will then describe how the system attempts to solve all the issues, even on massive scales.

More specifically, the new system will deal with:

1 Integrating two different families of singletons – static and dynamic
2 Handling massive numbers of dynamic dependencies
3 Deducing a valid destruction order for all singletons
4 Providing robustness by detecting cyclic dependencies and invalid uses of the system

These benefits, along with several others, will be provided to the programmer at a cost of usually two lines per object.

These capabilities form "A Unified Singleton Framework". It attempts to unify common problems of all such objects. It unifies different classifications of singletons. It unifies their solutions, relieving the usual hand crafting on an object by object basis. Once I have demonstrated this system to you, I hope you will agree that you would not want to write another system without using the unified singleton approach.

The Classic Singleton

The first book to formally document the singleton was the Design Patterns book by Gamma et al. The book describes singletons as objects that have two properties – global access and single instance. Let's examine each in turn.

Global access could be another term for "convenient access". The benefit of easily accessing an object, such as not having to pass a parameter through a multitude of functions, is less code and less up-front planning. This together means simpler. One downside to a singleton being globally accessible, is that it conflicts with the principle of encapsulation. If there were a way to make an object conveniently accessible but only selectively available to its intended audience, there would be room for improvement on the pattern. But current methods of restricting access encroach on convenience, which would void the primary reason in the first place.

The second principle assures the use of the same object. Many assumptions and synchronization requirements rely on this singularity, and the singleton pattern delivers this well.

Singleton Usage

Consumers of singletons really care about one usage feature – getting a singleton. They need not concern themselves when singletons are created or destroyed, or the dependent relationship between one singleton and another. Nor do they care about where they came from, where they live, or what parameters were needed to create them. They proclaim, "Give me! No Details Please!" I will demonstrate this in code shortly.

It is this "getting" that provides the simple interface into the singleton abstraction. The following is an interface for singletons which covers all possible singleton classes. It is a templated function that takes the singleton class as a type parameter.

[continued from previous page]

You need answers that fit your organisation (like using pair programming, test harnesses or code reviews to ensure the standard of work). These are all techniques that focus on quality – but the truth is that is it quicker to develop things right than to develop them wrong and then fix them. This is often a hard cultural change for an organisation because the connection between quality and progress isn't easy to make explicit.

Whatever working practices you try to introduce you also must lead each of the affected individuals to realise that their habits are causing problems on the project. Unless the majority of these individuals are prepared to change at the same time the individuals concerned will return to the "comfort zone" of their habitual behaviour and the project will slip back into failure mode.

Alan Griffiths
alan@octopull.demon.co.uk

```
template<class TSingleton>
TSingleton* GetSingleton();
```

Using the interface, an example of singleton usage would be:

```
GetSingleton<Printer>()->
       Print(GetSingleton<DocManager>()->
       GetMyDocument());
```

This shows two singletons, `Printer` and `DocManager`, being accessed in the same line. No previous preparations were made. No objects had to be passed in. This is especially useful for the main function, since no objects are ever passed in.

Making a Singleton

How difficult is it to make a class a singleton? This is a good question since many highly regarded books have differing opinions. Effective C++ by Scott Meyers, and the Design Patterns book present slightly modified versions based on what appear to be a similar theme. You might have to write a half-dozen or more lines of cookie-cutter code using a static variable of some sort:

```
class Singleton {
public:
  static Singleton& GetSingleton() {
    static Singleton instance;
    return(instance);
   }
private:
  Singleton() {}
};
Singleton Singleton::instance;
```

Modern C++ Design by Andrei Alexandrescu takes a step forward in providing some templated classes to alleviate most of this work, but there are still several template decisions and instantiations to make. He also presents several usage modifications that can be applied to individual singletons. This is important given there are probably innumerable variations of problems. But what is equally important is to allow a system to deal with these pieces in a unified manner. This will be demonstrated later on.

If the previous code seems like a lot, that's because it is. There is a simpler way, and it can be done in two additional lines:

```
class log {
public:
  API_SINGLETON(log);
};
// In a .cpp file
DEFINE_SINGLETON(log_impl);
```

That's it. `log` is now a singleton. Any function can retrieve the singleton by making the call:

```
GetSingleton<log>();
```

The function will retrieve the one and only instance of `log`. The `log_impl` class can be the `log` class itself or any derived type. This would allow for polymorphic singletons and their implementations to be encapsulated from the user. The singleton will be properly created by the time it is needed and destroyed before the program exits. No further work is needed and no singleton memory leaks are assured.

The following is a simplified version of the `API_SINGLETON` macro. I've temporarily stripped out the notion of multiple types of singletons, linkages and private data:

```
#define API_SINGLETON(T)\
class SingletonTraits {\
public:\
  SingletonTraits();\
  ~SingletonTraits();\
  static T* mpInstance;\
};\
SingletonTraits SingletonTraitsInstance
```

This macro declares an internal class traits type and then makes an instance of that trait.

You might wonder why not use a templated traits class instead of a macro. It is a technical issue about what can be templated and what cannot. I will point out later in the article that a unified singleton can be shared across component boundaries. On some platforms, such as Microsoft's Windows, certain linkage specifiers cannot be templated. One component is required to export the definition while another component will need to import it. There is some leeway for member functions, but not for member data. Although not my preference, macros seem like the only way around this issue. But that's ok. The macro/template combination provides a powerful idiom. The `DEFINE_SINGLETON()` macro has requirements that cannot be templated either.

This trait instance allows the system to detect whenever the class is constructed or destroyed, by monitoring the respective methods. Because of this, the system is always aware if more than one singleton is created, or if one is improperly destroyed. It will immediately notify the user of any invalid usage, making it difficult to accidentally subvert the system. It will be shown later that some singletons need to be made by the user at least once, so we cannot protect the constructors or destructors for their implementation type.

For example, if a user attempts to make a singleton class on the stack, the system will report an assertion. If a user attempts to delete the singleton by calling delete, an assertion is reported. There isn't a lot of room for misuse, which means less time tracking down errors.

The `DEFINE_SINGLETON()` macro is a bit more involved. It implements several features that are not present in previous singleton frameworks. I would like to outline their functionality first before discussing it further.

With these traits, a simplified version of the `GetSingleton` template looks like this:

```
template<class T>
inline T* GetSingleton() {
  typedef typename T::singleton_traits traits;
  if(traits::mpInstance == NULL) {
    Create< T >();
  }
  return(traits::mpInstance);
}
```

Most of the difficult work is deferred to the `Create()` method which is defined by the `DEFINE_SINGLETON` macro and explained later.

Dynamic Singletons

Let's take another look at a different singleton case. There is one case in particular that usually influences a programmer to reject singleton as a design for managing an object's lifetime.

Let's assume that a program uses a global object called the "Renderer". Let's also say that we want two different versions of the renderer – one for OpenGL and one for DirectX. They both utilize the same interface. We want to choose which renderer to

use when the program starts. This means the selection of which object cannot be made until run-time. We also want to create the renderer with some run-time arguments.

This is where a programmer may now say, "Oh well. The renderer is sort-of a singleton, but just doesn't quite fit the pattern. I'll have do everything from scratch on this one and manage it by hand." Let's pause there.

This is the case where if something doesn't exactly fit one's conception of singleton, then it might not fit at all. But if the programmer were willing to slightly extend his notion of a singleton, there would be a lot more room to work with.

If you recall, the original usage of a singleton is quite simple and has only one requirement – "GetSingleton". If both renderers have the same interface, then the consumer does not care which version of the singleton he is getting. He wants the one and only "renderer". Somebody else should have already taken care of selecting which version the renderer represents underneath.

This is an example of a dynamic singleton. As far as the consumer can tell, it is no different from a static singleton, which is bound during compile-time. Consumers just say "get" and they expect a singleton. Who bound it, created it, or anything else is remains unnecessary detail to them.

A dynamic singleton is just as easy to make:

```
class Renderer {
public:
  API_DYNAMIC_SINGLETON(Renderer);
};
// in the .cpp file
DEFINE_DYNAMIC_SINGLETON(Renderer);
```

Functions can get the singleton by making the same call:

```
GetSingleton<Renderer>();
```

The one instance of the renderer will be returned. It will either be the OpenGL or the DirectX version, depending on the setting code.

The dynamic singleton leads to other benefits as well. It can be set by modules that are loaded later in the program, also called late-binding. They can also be swapped out for another singleton. For example, it is possible to swap the OpenGL renderer for the DirectX version during the middle of the program execution. If all the consumers use the Renderer interface, they will never be aware of the difference, allowing for a lot of flexibility. If a singleton must be created with run-time evaluated arguments, dynamic singletons should be your choice. In contrast, static singletons are always ready, so there really is no appropriate time to provide constructor arguments.

You might think that this violates the principle of single instance. From a user's point of view, there must be only one instance of a singleton at any point in time. And the unified system assures this. But deleting a current singleton and then immediately creating a new one to replace it, would not imply multiple instances at any frame in time. Getting the singleton will never be ambiguous, which as pointed out previously, is the fundamental requirement of any singleton framework.

There is a slight extension to the interface of a dynamic singleton. We need to allow the singleton-producer code to select which singleton to use, and we do this through `SetSingleton`:

```
template<typename TInterface,
         typename TImplentation>
SetSingleton(TImplementation* pSingleton);
```

This method will set the singleton pointer. Any function that calls `GetSingleton()` on a dynamic singleton that has not been set would be undefined. Notice there is a second templated argument – `TImplementation`. This type is used as the derived type when deleting – which means the interface need not expose a public destructor, and the implementation is kept encapsulated from the consumer. For trivial classes, the implementation could be the same type as the interface. When a dynamic singleton is destroyed, it will be deleted properly.

Contrast this with a static singleton. Static singletons are always set, and do not support the `SetSingleton()` interface. If you were to call `SetSingleton()` on one, the compiler would report a syntax error. One big advantage of static singletons is that you can get them during global initialization as well, which occurs before `main` is invoked.

Destruction and Dependencies

Now that all superficial interface discussion has been touched upon, I can discuss the real bread-and-butter of a unified singleton framework. It solves a problem that is more devious than most engineers may at first imagine, or at least until they actually want their program to shutdown without making a mess.

As a project's lifetime progresses, developers are usually capable of getting a program to boot up and eventually load up all the objects they will need. But, getting the program to shutdown and actually delete all those objects is usually another matter. The complexities of preventing one object being destroyed too early (before another that it depends upon), or of avoiding an unanticipated cyclic dependency during the development, are often realized too late. A program may not be able to shut itself down cleanly without crashing and might have to rely on the operating system to abruptly kill the process.

The solution to this problem lies in well-defined dependencies. If the singleton framework knows the proper dependencies, then this problem is easier to solve.

Let's say there are a `keyboard`, a `log` and a `disk`. The `keyboard` writes to the `log`. The `log` in turn writes to the `disk`. The `keyboard` depends on the `log`, and the `log` depends on the `disk`. It would look like this:

```
keyboard -> log
log -> disk
```

There is only one proper destruction order if a dependee were to always be available to its dependent:

1 Destroy `keyboard` (no object depends on it)
2 Destroy `log` (`keyboard` is already destroyed)
3 Destroy `disk` (`keyboard` and `log` are already destroyed)

In Modern C++ Design, Andrei Alexandrescu suggests that a programmer could manually assign longevity numbers to singletons to convey these dependencies. This is certainly plausible if there were only a few objects, but what about a dozen? Longevity numbers themselves have no inherent meaning; they are only relative to other singletons. As a system develops, some of those singletons will change. And if you recall, ten objects already have 3.6 million different longevity orders. If a programmer were able to continually consider 1,000 of those, there would still a few million he was omitting. This is an intractable problem. One thing I learned in my computer science classes, I could spend my time more wisely than trying to solve an intractable problem.

You might be tempted to think that finding an order for ten really isn't that hard. If there are only a couple dependencies between

singletons, finding one usually isn't. But every time a new dependency is introduced, the entire order needs to be reevaluated and could change. As the system grows and more code needs to get shared, the dependencies will get complex. That is when the magnitude of this problem usually shows up.

Andrei does suggest an alternative by alluding to a dependency manager for singletons. But he also brings up a separate optimization issue in that singletons that are not explicitly used should not be created. The overhead of unused singletons could be inefficient. Because of this, references must be placed in the dependees that refer to the dependents. Using the above example, the `disk` would have to refer to the `log`. The relationships would then look like this:

```
log->disk
disk->log
```

This would cause a circular dependency. Due to this, Andrei dismisses the idea of singleton dependencies altogether.

I would first have to examine the reasoning for such an optimization. Most singletons that are at the system-level will eventually be required anyway and must be allowed for. Another difficulty is that one dependent may have multiple other dependents, forming a large dependency tree. Temporarily optimizing out a root singleton may prevent many other dependencies from being specified. It may not be possible to recover these at a later point in the program.

If a singleton's creation overhead is really significant, there are alternative solutions available. The overhead could be moved out of the singleton's constructor and into the first usage of the object, such as in its member methods. For example, in a `log` class, a file could be opened when the "logging" method is first executed. I would describe the benefit of this removal optimization as minimal compared to the enormous difficulty of obtaining a proper destruction order out of a million possibilities, or even a number represented with 157 zeros. To put this in different perspective, I wouldn't want to be distracted away from a taming a man-eating beast because a small mouse crossed my path.

The reason why correct dependencies will find a proper destruction order is that the unified singleton framework has a "Singleton Stack". Whenever an object is fully constructed, it is pushed on to the singleton stack. When the program exits, it will destroy the singleton stack in the reverse order it was created.

To specify a dependency, there is an additional line of code in the dependent's constructor:

```
disk::disk() {}
log::log() {
   GetSingleton<disk>();
}
keyboard::keyboard() {
   GetSingleton<log>();
}
```

A rule of thumb that has been helpful to me is that if you access another singleton in any member function (or the destructor), then you must get the singleton in the constructor as well. I call this the principle of "symmetric getting". The result will be correct dependencies.

A couple of others things to note. Static singletons are created during their first get. Singletons are not added to the singleton stack until their constructors have completed. So no matter which singleton is requested first – `disk`, `log`, `keyboard`, the singleton stack will always be the same – `disk`, `keyboard`, then `log`.

If `log` is asked for first, `log` will create `disk` and put it on the stack first. If `keyboard` is asked for first, it will create `disk` and then `log` and then `keyboard`. This is very similar to how constructors in derived objects work in C++. It represents the essence of dependencies.

Cyclic Dependencies

There is also one inherently invalid sequence of dependencies – cyclic dependencies. When cyclic dependencies are introduced, they eventually lead to an infinite loop. Thus they always need to be avoided. For example, if we had these dependencies:

```
log::log() {
   GetSingleton<disk>();
}
disk::disk() {
   GetSingleton<log>();
}
```

Which should be created first? Which should be destroyed first? There is no correct answer. But that's ok, since the unified singleton framework will detect cyclic singleton dependencies during run-time. An assertion will be encountered if one exists. This means that if no assertion is encountered, you are guaranteed to have a non-cyclic singleton design, which could otherwise be very difficult to prove in a large system. You can then rest easy as a maintainer of such a system. This also guarantees that there exists a correct destruction order, and the system will be able to invoke that order – another comforting thought.

Putting it all together

Now that I've touched upon the main issues that the unified singleton framework addresses, we can return to the `Create` method which is instanced by `DEFINE_SINGLETON()` macro.

At a high level, it implements the following:

1 Creation code for the polymorphic implementation type.
2 Registering the singleton onto the "singleton stack"
3 Tracking all possible states a singleton may be in.
4 Detecting misuse – cyclic dependencies, multiple runtime-bound singletons

Here is the `Create` method from the static singleton implementation:

```
template<typename T, typename TCreatePolicy,
   typename TThreadPolicy = BasicThreadPolicy>
class static_singleton_impl
    : public singleton_impl<T> {
  static void Create() {
    T*& instance = Instance();
    TState& state = State();
    if(state == Constructing) {
      // Cyclic construction was found
      ASSERT_MSG(0, "Cyclic
              singleton dependency detected");
    }
    else if(state == Destroyed) {
      // Accessing a dead singleton
      ASSERT_MSG(0, "Accessing
                        dead singleton");
    }
```

```
      else {
        // must be in uninitialized state
        ASSERT(state == Uninitialized);
        // set constructing state
        state = Constructing;
        instance = TCreatePolicy::Create();
        // after instance has completed
        // construction, register it on the
        // singleton stack
        state = Constructed;
        GetSingleton<SingletonStack>()->
                              Register(Destroy);
        // register an atexit call
        atexit( AtExit );
      }
    }
  }
```

The method is primarily responsible for dealing with two variables – the instance and its state. It treats the singleton as a mini-state machine.

The first "if check" looks to see if the singleton is already being constructed. If so, then this singleton has a construction cycle. This can often be difficult to visually inspect, since singleton recursion may occur several layers deep.

The second "if check" looks to see if the singleton was previously destroyed. This would be a "dead singleton" access. The framework is responsible for ensuring that this does not happen; that is the benefit of the framework in the first place. It is more of an internal check to make sure the framework is working properly.

If the singleton passes all those tests, then the creation begins. It sets the state to `Constructing` and then invokes the creation policy, which may be customized to each singleton. A provided simple creation policy will call `new` on the implementation type, which might be a derived type or the type itself. After it is fully constructed, the state is set to `Constructed` and then registered with the `SingletonStack`. The `SingletonStack` is a singleton itself and will always be on the bottom of its own stack. The `Destroy` parameter is a static class member that ensures smooth destruction of the singleton in a similar fashion to how it was created.

An `AtExit` handler is then registered with the C run-time system. It will detect when a module is being shutdown. It is worth noting that the `atexit` must be called in singleton template implementation and not in the `SingletonStack` implementation. There can be several `atexit` stacks when there are multiple run-time modules. We want to ensure that a singleton's `atexit` is registered in the module that was responsible for creating the singleton.

A dynamic singleton implementation does not have a related `Create` method, since it never actually creates a singleton. But it does deal with custom destruction methods that are passed in when a dynamic singleton is set. At that time, it registers the destruction with the `SingletonStack`. There is a helper template that makes it simple for users to pass in these destruction methods easily.

It is these details, and integration of the subtle variations of singletons, that prove to be a bit of work. If done by hand for each singleton, I would ask if an individually written singleton is the best choice for a robust system.

Other Benefits

Singletons in a unified framework can be accessed across component and dll boundaries. In fact, singletons could serve as the only interface between components. One component could get a singleton that is provided by another. Users would get this for free and do nothing special when retrieving a foreign singleton. Gets are resolved at linking time so there is no run-time performance penalty.

Developers also can have full access to all static singletons before `main` is executed. There could be a complex initialization routine during program static initialization. They need not worry whether a singleton is ready since the system ensures it will be, and by the way, destroyed properly. The system works equally well before or during execution of `main`. This is a difficult task to do outside such a system, and I had personally never seen anything do it correctly, much less on a large scale.

This framework also allows for singletons to be template template parameters. In modern generic programming, this provides a powerful mechanism for many advanced techniques.

Future Directions

The unified singleton framework can serve as the foundation for more complex systems. To date, factory systems, state machines, and resource managers have all been built onto using this architecture. Since no requirements are placed on any of their constituent, i.e. forced base classes or required implementation, it is easy to build on top of.

Conclusion

Singletons can provide a basis for other components in a modular system. They can come in two flavours: static and dynamic. Each has a trade-off of capabilities. The unified system maintains a live singleton stack. Singletons are destroyed in the reverse order to that in which they were created. Cyclic dependencies represent an invalid design and minimally can be detected by the singleton framework at run-time. Best of all, it provides a technique to solve the large-scale dependencies and life cycles of all singleton objects.

Jeff Daudel
jeffdaudel@yahoo.com

References

[1] *Design Patterns* by Gamma et al., 1995
[2] *Modern C++ Design* by Andrei Alexandrescu, 2001
[3] *Effective C++: 50 Specific Ways to Improve Your Programs and Design (2nd Edition)* by Scott Meyers, 1997

Full code of a unified singleton framework and usage examples can be found at: `http://daudel.org/code/Singleton.zip`

EDITORIAL COMMENT
This article prompted considerable debate amongst the members of the editorial team. Some feel it should not be published in this form because it encourages excessive use of the Singleton pattern, while others feel that it deserves publication because it describes a valid technique for implementing Singleton classes.

Singleton does two things: 1) it ensures that no more than one instance of a class can exist in a program and 2) it makes it easy to access the single object from anywhere in the program. The first is sometimes (although rarely) useful. The second is an invitation to excessive coupling which needs to be resisted.

This is an unusual situation for the editorial board, as most issues are fully resolved before publication, and I rarely (if ever) comment on articles directly in these pages. We are not here to censor though, just to guide authors through the process of creating technically correct, well written, and interesting articles. We would certainly welcome any comments from the readership about this article for publication in Overload 57.

John Merrells, Editor