contents

credits & contacts

Overload Editor:
Alan Griffiths
overload@accu.org
alan@octopull.demon.co.uk

Contributing Editor:
Mark Radford
mark@twonine.co.uk

Advisors:

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@gmail.com

Pippa Hennessy
pip@oldbat.co.uk

Advertising:
Thaddaeus Frogley
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

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: Does all this help make better software?

I'm writing this in the second week after the ACCU Spring conference where I met many of the contributors whose articles have appeared in these pages. I enjoyed putting faces to names and the many conversations that ensued. I also enjoyed meeting all the others of you that attended and, judging from all of the feedback that has come my way, all of you enjoyed the conference too.

But was the purpose of the conference to have fun? (We did!) Or were we there to learn to make better software? (Did we?) Judging from the topics of conversation there, people went to learn about producing software, and the fun was a byproduct of an effective learning strategy.

In a talk by Allan Kelly (on learning) I learnt a name for a concept I've sometimes struggled to convey: a "community of practice". This, at least the way I interpret it, is a collection of individuals that share experiences, ideas and techniques in order to develop their collective knowledge. What better description of the group of people that read each other's articles, attend each other's presentations and provide feedback in the pub (or on mailing lists) afterwards?

Does it help to have a name for this? To be able to say "the ACCU is a software development community of practice"? Yes, it does. There is power in names: we can ask, "do you know other software development communities of practice?" far more effectively than "do you know other organisations like the ACCU?"

And recognising that the ACCU is about mutual learning makes it a lot easier to explain the value of participating in its activities. Why go to the conference? To learn what others have discovered! Why write for the journals? To get feedback on what you have discovered! With a community one gets back what one puts into it.

Does Being Part of a Learning Community Help Us Make Better Software?

One of the characteristics of group learning is the way that ideas build upon one another. Some years ago a discussion of optimising assignment operators led to Francis Glassborow writing an article "The Problem of Self-Assignment" (Overload 19). An editorial comment about the exception-safety of one approach to writing assignment operators led to articles by Kevlin Henney "Self Assignment? No Problem!" (Overload 20 & 21). One could trace the development of this idea further. For example, in "Here be Dragons" (Overload 36) I incorporated a generalisation of the principal idea into an exception-safety pattern language as the "Copy-Before-Release" idiom (and current discussions of "move semantics" are also related).

Over the course of a few articles the canonical form of the assignment operator was rewritten and the C++ community (or part of it) has adjusted its practice accordingly. But where did the idea come from? Francis was describing a method of writing an assignment operator that didn't require a branch instruction to deal with self-assignment - he wasn't writing about exception safety and, as far as I am aware, took no account of this aspect of the form he described. The editor's remark recognised the exception-safety aspect, but didn't seek to revise the "right way" to write assignment operators. It was left for Kevlin's articles to address the issues with earlier practice directly and make the case for change.

Naturally, this is just one example of the way in which participation in a community of practice helps advance the frontiers of knowledge. The exchange of ideas within the community was what led to the recognition of the importance of this idiom - no one person deserves all the credit. As a result, the programs being written by members of this community became just a little bit more reliable and easier to maintain.

The Business Value of Better Software

Do the ideas developed within the community help us deliver more valuable software? Or to be specific, how much difference does knowing how to write an exception-safe assignment operator make to the average program?

If one can judge by the quantity of code that, on successful projects, reaches production with exception-unsafe assignment operators, not a lot! On the other hand, not having exception-safe assignment presents a demonstrable risk of obscure, hard to reproduce problems. I've worked on several systems whose delivery schedule has been seriously disrupted by avoidable problems that required significant effort to diagnose.

One system I worked on lost a number of days when an intermittent problem was found that only occurred on Win2K (or maybe it was NT4 - one appeared to work reliably, the other failed at random intervals of up to a couple of hours). After the project missed a milestone I was asked to take over the investigation, as both the developer that identified the problem and the project lead had run out of ideas. I spent two days isolating an "undefined behaviour" problem that, once diagnosed, was clearly the result of a coding style that was not as robust as one might expect from ACCU members. (OK, in this example it wasn't actually an assignment operator, it was a "one definition rule" violation, but an exception-unsafe assignment could easily have the same effect on a project.)

There is an obvious cost to the business of developers spending time learning new techniques, not just the time spent learning, but also ensuring that "shiny new hammers" are kept in the toolbox and not the hand. The savings are not always so obvious. In the above example I know how much the client paid for the time I spent diagnosing the problem, and roughly how much time I spent explaining to team members what had gone wrong (and how I'd figured out a problem that had baffled them). What I don't know is the opportunity cost of delivering the software late, or the value of the work that would otherwise have been produced by the developers that tried to solve it. Whatever the cost was, it would have paid for a lot of learning!

Managers (at least the good ones) do understand the concept of insurance, paying small premiums in terms of learning activities to avoid the risk of the occasional, but very expensive, search for obscure problems that completely invalidate the planned delivery. Indeed, many of the practices being promoted by the “Agile” development community (frequent deliverables, continuous integration, automated acceptance test suites, ...) can be viewed as ways to minimise the risk of the project drifting off course.

Of course, there is a balance to be found - a point at which the additional cost of further improving the developers, or development process, or tools would exceed the added value delivered. This is clearly not an easy assessment to make. This process of assessment is frequently hindered by developers failing to demonstrate that they understand the need to deliver business value - which makes it appear that they are simply “seeking new toys”.

Demonstrating Business Value

It is hard to relate improved design idioms or more effective tools to the value they bring to the project or organisation. And herein lies part of the difficulty in assessing the value of the ideas we exchange and develop in such forums as the conference. Many of them seem to merely reduce the size of that vast pool of infrequently manifesting bugs that seems to exist in most codebases. (The existence of this pool of bugs and the appropriate way to manage it was another discussion at the conference - is it worth putting any effort into diagnosing and correcting a problem that may only manifest once a century?) A very substantial proportion of the infrequently manifesting bugs needs to be removed before any benefit can be perceived - and, as we don’t know the size of this pool, it is hard to know in advance what impact reducing it can have.

Demonstrating that one approach to developing software is better than another is hard, there are lots of variables and people will interpret the evidence in accordance with their preconceptions. I once ran one of three teams producing parts of a project. My team put into practice all the best ideas I knew from my reading: daily builds, automated test suites, reviews of designs and code. Another team divided up the functionality and each team member coded their own bit. The process the third team adopted revolved around a self appointed expert programmer (who rewrote any code that wasn’t “up to standard”). My team didn’t suffer the time pressure seen on the other two (no need for overtime) and delivered software that had an order of magnitude fewer defects discovered in integration testing than either of the other two. I thought I’d done a good job and demonstrated the effectiveness of my approach. However, rather than my approach being adopted by the other teams, my “idle” developers were moved to the “more dedicated” teams that were working overtime to fix bugs and were soon as dedicated as the rest.

I learnt from the experience (even if the organisation that was employing me didn’t) that it is not enough to succeed, it is more important to be seen to succeed. While my team had always had a clear idea of what functionality was completed (and was confident that it worked) there was very little visibility elsewhere. What was visible to everyone was my team was going home on time, sitting around chatting in front of PCs and having no difficulty in producing its deliverables. We were having fun while the other groups were making heroic efforts to deliver! I should have made sure that there was much more visibility of the progress we were making. It doesn’t take much effort to present a list of the intended functionality on a whiteboard, pin-board or similar and to check off the items that have been verified as having been completed. A visible indicator that progress was being made would have done a lot to reassure other parts of the business that we knew what we were doing and to justify our apparently relaxed attitude. (There were other things to learn too: from inexperience in introducing ideas into a team or organisation, I kept too much control of the team practices - the team members didn’t feel that they “owned” them and, in consequence, didn’t promote them in the teams they were moved to.)

Having Fun

“Having fun” is an indicator that things are going well. Any time you’re not having fun writing software, or learning, (or anything else) then it is time to assess the reason for it. That isn’t to say that there are not occasions where rote learning and repetitive practice are effective: most software developers I know would benefit from learning to touch-type, which involves a lot of repetition to (re)form habits. (And even in these cases, it helps to introduce a fun element.)

There is nothing wrong with having fun: just consider how to show that you’re delivering value at the same time!

Alan Griffiths

overload@accu.org

Copy Deadlines

All articles intended for publication in *Overload 68* should be submitted to the editor by July 1st 2005, and for *Overload 69* by September 1st 2005.

Letters to the Editor

PHP-Related Letter to the Editor – Overload 66

Terje,

I have recently found time to learn PHP, which is something I had been meaning to do for a while, and my response to the dynamic typing is the complete opposite of yours; I found it quite refreshing.

Provided that you have sufficient tests, such as those you get from doing TDD, the dynamic typing can be liberating rather than a cause for concern. After all, it's not as if it's something new – Smalltalk is dynamically typed, for example. In fact, I often find myself using templates in C++ precisely because the implicit interfaces give me some of the freedom of dynamic typing.

I found that the lack of explicit types made refactoring easier, as I did not have to change variable declarations or function signatures, and I didn't need to explicitly declare interfaces.

You only have to look over on comp.object to find a horde of people who swear by dynamic typing, including respectable authors such as Ron Jeffries and Robert (Uncle Bob) Martin.

I'm not yet willing to say I prefer dynamic typing to static typing, but I can certainly see the benefits. I had a discussion with someone recently about naming conventions, arguing that including information about the type in variable names (such as with Hungarian Notation) was a waste of bandwidth, as the name and usage of the variable should provide the necessary information. Dynamic typing goes a step further, and says that the actual type is really quite irrelevant, it's what you do with it that counts. Of course, it does mean that good naming and clean design are all the more important.

That's not to say that PHP is perfect, though – you can't overload functions, for starters, though I guess that's a consequence of them not having a fixed signature. The limiting factor here is that it can make it hard to extend existing code; I can't provide alternate implementations of functions that work on new classes.

Following this line of thought to its natural conclusion drives me to the "everything is an object" stance of Smalltalk – if everything is an object, then you can extend behaviour just by writing a new class which derives from, or even just wraps, an existing class (after all, we don't need inheritance for the interface), and passing instances of the new class instead of instances of the old one.

I guess that this actually demonstrates that C++ is evolving in a direction that works – if you're going to have non-member functions, then for maximum extensibility, you need static typing and overloading, including operator overloading. You also need generics (templates), and the ability to deduce types from expressions (the auto and typeof proposals). It's just a different way of providing freedom of expression, and the ability to write clean abstractions.

PHP therefore lives in a sort of never-never-land, where you can write code with everything being an object, but you can write free functions too, and the language ships with rather a lot of them. It is neither one thing, nor the other, which makes for plenty of opportunities for poor coding. That just makes it all the more important to strive for clean design, using refactoring and plenty of tests to help along the way. I welcome the addition of exceptions to PHP 5; having to work with error codes in PHP 4 reminded me how hard that can be. PHP 5 also adds destructors, which fire when the last reference to an object is destroyed; much better than Java's finalization mechanism.

That's my thoughts for now,

Anthony Williams
anthony.ajw@gmail.com

And Terje's Reply:

Anthony,

Thanks for your thoughtful reply. Yes, I know that quite a few proponents of agile development like dynamic typing. Before going further, perhaps establishing some terminology might be in order. Often, *dynamic* and *implicit* typing is lumped together in discussions, while they are really orthogonal. One classification I've read and found useful is: *implicit* vs *explicit* typing (whether or not the type is declared), and *static* vs *dynamic* typing (whether or not types of values are determined and checked at compile time or run time). With this, we can classify some languages:

- C++: Explicit and statically typed
- Haskell: Implicit (with optional explicit) and statically typed
- PHP: Implicit (with optional explicit for user-defined types in signatures) and dynamically typed

In languages like Haskell, if you don't specify the types, they are inferred from the expressions at compile-time. Something similar happens with C++ templates, as you mention. However, Haskell has type classes, giving a form of constrained genericity (as in Java 5.0 and C# generics, and the concept proposals for C++ [1]), whereas parameters to C++ templates, as they are today, are pretty unconstrained. This tends to lead to hard-to-understand error messages, when the program crashes deep inside a function (rather than giving an error at the call site about parameter mismatch). The same kind of problem, only worse, may happen in dynamically typed languages. Worse, because:

1. Being dynamically type-checked, you don't get any errors before that particular piece of code is executed with those particular parameters, and
2. Every variable and parameter is basically a variant - being able to take on a value of any type, including null

I find it interesting that in C++, unlike the current state of scripting languages (although there's some movement in that direction for them as well), there's a tendency towards getting a language where you can be more explicit about your intent [1]. Proposals such as "Explicit Class" (N1702 - being able to decide which, if any, default member functions are provided), "Explicit Namespace" (N1691 - being able to turn ADL off, and selectively enable it), "Explicit Conversion Operators" (N1592 - having to explicitly request a conversion), as well as the mentioned concept proposals (N1758, N1782, et. al. - enabling you to state the concepts that the template parameters have to model), "Design by Contract" (N1773), etc. all facilitate expressing our intent more

[concluded at foot of next page]

Can C++ Learn from Generics in Ada?

by Peter Hammond

Introduction

Although C++ templates afford great flexibility and power in the language, it is widely recognised that the widespread adoption of modern template techniques is hampered by "features" of the implementation of templates [Heinzmann, 2004a, b]. At the recent ACCU, conference Stroustrup [2005] described two proposals currently before the C++ standards committee for implementing Concepts in C++. Concepts seek to alleviate the difficulties of template programming by separating the definition of the template from its instantiation to a certain extent, in as much as the contract placed on the types used to instantiate the contract is explicitly stated, so it can be checked and more meaningful diagnostics given.

In a question at the end of this presentation, Coplien suggested that other languages may have models that C++ templates could learn from, and specifically mentioned Ada generics. This paper will give an overview of the features of the Ada generics model, from a personal perspective of a programmer with some experience of Ada, rather than a language designer. It will show why the author believes C++ has little to gain from Ada generics.

Rationale of Ada

The design of the Ada language puts a very strong emphasis on safety and integrity. It is a language that is particularly tractable to static analysis by abstract interpretation [Chapman, 2001]. The design of the generic model carries this on, aiming to provide "*the security that is present for ordinary, unparameterized program units; in particular the degree of compilation-time error checking*" [Ada83].

The main forces in the design of C++ are significantly different. While no language deliberately courts run time errors, C++ puts the emphasis more on flexibility and expressiveness [Stroustrup, 2000]. As a consequence of these differences, the Ada generic model is very different from the template model of C++.

The Ada Generic Model

The model of generic units in Ada allows a high degree of static type checking to be applied to the generic body alone, independent of its instantiation. It imposes a contract on the types that can be used to instantiate the unit. This is a very attractive idea, and is the reasoning behind the proposals for Concepts in C++. However, the use of generics in Ada will not fit well with the kind of ad-hoc specialisation and template reuse that is common in modern C++.

Two features of the Ada generics mechanism would be particularly unsuitable for C++, and they are also key to the way Ada achieves the goal of separating definition and instantiation:

1. The need for explicit instantiation;
2. The limited set of formal types that can be used.

Furthermore, Ada's very strict type model leads to additional generic parameters being needed that at first sight seem redundant.

Explicit Instantiation

The first of these is a tiresome burden for the programmer but probably not inherent in the generic model. Given the overheads implied by the syntax, this can be as much typing (and therefore reading) as writing out the body in full. For example, whereas in C++ we might do this:

```
int x, y;
some_big_struct a, b;
. . .
swap (x, y);
swap (a, b);
```

[continued on next page]

[continued from previous page]

clearly in the code (and being able to be checked by the compiler/runtime).

By expressing our intent explicitly in the code (using type declarations, concepts, contracts, or whatever), the compiler is better able to sanity-check the code, and in the case of static type checking, even before the program is run. Moreover, it provides documentation for the users and maintainers of the code, as you can see what is expected, and what is guaranteed. In contrast, languages like PHP hardly lets you specify *anything* up front, and thus it defers any error detection to runtime (if even then). A couple of common arguments for dynamic typing are that a) static typing doesn't necessarily find all errors, and b) with unit tests, you can perform similar checks, and more. I feel that a) is a kind of straw man (as it can be said about most good practices), and it doesn't mean that static type checking isn't useful. As for b), yes, you can do it with tests (or using asserts in the function definition, checking the types manually), but why would you? By using static type checking (whether or not the types are declared) you're assured that the program makes at least some kind of sense. Why anyone would want to do manually what the compiler can do automatically is beyond me. Yes, unit tests are useful, and they complement type checking (and DbC, etc.) nicely, rather than displacing it.

Static checking (any kind of checking) enables you to detect any bugs early, and that's a good thing, as it gives rapid feedback. As mentioned, with a dynamically typed language, you risk getting type-related errors in code that is only executed rarely, perhaps a long time after it's written, or worse, you get silent errors, unless you're sure you've covered all the possible cases with tests, including values coming from outside.

If you want to be able to avoid specifying types, unless you explicitly want to, then, as you mentioned, type inference with templates (as well as the auto/decltype proposal (N1607)) enables this, and good generic code typically contains few explicit type declarations. This way, you get the "agility" of dynamic typing, with the type safety of statically checked types.

Regards,

Terje Slettebø
tslettebo@broadpark.no

References

[1]http://www.open-std.org/JTC1/SC22/WG21/docs/papers/

The Ada equivalent would be:

```
X, Y : integer;
A, B : Some_Big_Struct;
procedure swap is new swap (integer);
procedure swap is new swap (Some_Big_Struct);
. . .
swap (a, b);
swap (x, y);
```

This is generally a minor hassle in the idioms of Ada. However, it clearly makes the idioms of modern C++ impractical.

Limitations on Formal Types

The most significant drawback of the Ada generics mechanism is the restrictions on what the generic body can assume from the type provided to it. The declaration and instantiation of a generic in Ada takes the following form:

```
generic
   Type T is X;
Package Foo;

Package MyFoo is new Foo (Y)
```

Where `X` is one of the keywords listed in the following table and `Y` is an actual type argument as listed:

Keyword (X)	**Actual parameter (Y)**
Digits<>	Any floating point type
Range<>	Any integer type
Delta<>	Any fixed point type
[tagged] Private	Any type, but objects can only be copied and compared
[tagged] limited private	Any type, but objects can only be compared
New subtype_mark	Any type derived from subtype_mark

The only other permitted types are access (pointer) to one of those types, or an array of one of those types. As well as packages, functions and procedures can be declared using the same syntax. For example, the swap procedure might be declared in Ada as:

```
generic
   type T is private;
procedure swap (X, Y : in out T);
```

There is no way to access arbitrary components of a record (struct), nor to call overloaded subprograms[1] by name matching. The "new subtype_mark" form does allow access to arbitrary members and subprograms of the base subtype; this achieves approximately the same effect as passing an object by base-class reference in C++, which of course does not require a template at all.

As an example of the limitations this imposes, consider two record types:

```
Type rec1 is record
  X : Integer;
  Y : Integer;
  Some_Other_Stuff : Foo;
End record;

Type rec2 is record
  Y : Integer;
  X : Integer;
  More_Stuff : Bar;
End record;
```

This is roughly equivalent to the C++:

```
struct Rec1 {
  int X;
  int Y;
  Foo some_other_stuff;
};

struct rec2 {
  int y;
  int x;
  bar more_stuff;
};
```

Now imagine that the `X` and `Y` members are semantically equivalent in both structs, and some duplication is discovered which can be refactored. In C++, the following is possible:

```
template <typename XandY>
void do_common_thing (XandY& obj) {
  obj.X = getX();
  obj.Y = getY();
}
```

This is not possible in Ada; "something with members called X and Y" is not in the list above, so when you try to write:

```
Generic
  Type XandY is ???
Procedure do_common_thing (obj : in out
XandY);
```

There is nothing that can be put in the `???`. The same problem extends to calling subprograms using arguments of the parameterised type (the Ada equivalent of calling member functions). The only way for the generic body to call a subprogram with the parameter type as an argument is to declare each subprogram required as an additional generic parameter. Clearly this can become very long-winded if there are several subprograms called from one generic.

1 Ada distinguishes between Functions, which return a value and may take only input parameters, and Procedures, which do not return a value and may take input, output and in-out parameters. Together they are termed subprograms.

[concluded at foot of next page]

Microsoft Symbol Engine

by Roger Orr

Introduction

Last year, I wrote an article detailing some code to provide a stack trace with symbols in Microsoft Windows. [Orr2004]

On reflection, I think the Microsoft symbol engine deserves greater explanation so this article discusses more about the symbol engine, what it does and where to get it from. The ultimate aim is to provide useful information which helps you diagnose problems in your code more easily.

What Are Symbols?

When a program is compiled and linked into executable code, a large part of the process is turning human readable symbols into machine instructions and addresses. The CPU, after all, does not care about symbolic names but operates on a sequence of bytes. In systems that support dynamic loading of code, some symbols may have to remain in the linked image in order for functions to be resolved into addresses when the module is loaded. Typically though, even this is only a subset of the names appearing in the source code.

When everything works perfectly this is usually fine; the difficulties occur when the program contains a bug as we would like to be able to work back from the failing location to the relevant source code, and identify where we are, how we got there and what are the names and locations of any local variables. These pieces of information can all be held as symbol data and interrogated, usually by a debugger, to give human readable information in the event of a problem.

[continued on next page]

[continued from previous page]

The Ada standard [Ada95] sec 12.1 gives the following example:

```
generic
type Item is private;
with function "*"(U, V : Item) return Item
                                       is <>;
function Squaring(X : Item) return Item;
```

which has to be instantiated like this, assuming that the package contains an overloaded operator for its user-defined type:

```
function square is new squaring
         (My_Package.My_Type, My_Package."*");
```

Apparently Redundant Parameters

Another source of syntactic tedium is the lack of inference of parameterized types. This is not actually a result of the generic model, but arises from the strict type model in Ada. As an illustration, this code fragment is taken from the Ada 95 reference manual [Ada95], section 12.5.4:

```
generic
     type Node is private;
     type Link is access Node;
package P is
     ...
end P;
```

Access is the Ada keyword for pointer; the programmer is forced to tell the compiler what synonym for “pointer to Node” is going to be used. Compare this with what might be used in C++:

```
template <typename Node>
class P {
public:
  typedef Node* Link;
  ...
};
```

To a C++ programmer’s eyes, the **Link** parameter in Ada looks redundant. However, it is not; a new type created as a synonym for “access Node” is a distinct type and cannot be implicitly converted from any other similar type. The programmer thus has to tell the generic in case there is already a new type name in use. In C++ of course, **typedef** does not have this effect; it merely introduces a new spelling for the same type. While there are pros and cons for this approach, it does impose some lack of flexibility on the use of generics.

Conclusions

The Ada generics model has always been contract based, as befitting the origins and idioms of the language. This provides for a high degree of early checking, at the expense of flexibility. While there may be some general lessons to be drawn from this approach, the specifics of the mechanism make it far too restrictive to be of any direct use in C++.

Peter Hammond
pdhammond@waitrose.com

Acknowledgements

The author wishes to thank the reviewers for their helpful comments.

References

[Ada83] “Rationale for the Design of the Ada Programming Language”, http://archive.adaic.com/standards/83rat/html/ratl-12-01.html
[Ada95] http://www.adaic.com/standards/ada95.html
[Chapman, R 2001] “SPARK and Abstract Interpretation - a white paper”, http://praxis-his.com/pdfs/spark_abstract.pdf
[Heinzmann, S 2004a] “The Tale of a Struggling Template Programmer”, Stefan Heinzmann, *Overload 61* June 2004.
[Heinzmann, S 2004b] “A Template Programmer’s Struggles Resolved”, Stefan Heinzmann and Phil Bass, *Overload 61* June 2004.
[Stroustrup, B 2000] *The C++ Programming Language*, Special Edition, Bjarne Stroustrup, pub Addison Wesley.
[Stroustrup, B 2005] “Better Support for Generic Programming”, http://www.accu.org/conference/accu2005/kd9fc73n3uj94krmnfcj383jhpaduyivby/Stroustrup%20-%20Better%20Support%20for%20Generic%20Programming.pdf

Most programmers using Microsoft C++ on Windows are familiar with the Microsoft Debug/Release paradigm (many other environments have a similar split). In this model of development, you begin by compiling a 'Debug' build of the code base in which there is no optimisation and a full set of symbols are emitted for each compiled binary. This generally gives the debugger the ability to work backwards from a logical address and stack pointer to give the source line, stack trace and contents of all variables. Later in the development process you switch over to building the 'Release' version of the code which typically has full optimisation and generates no symbolic information in the output binaries.

There are several pitfalls with this approach. In my experience the most serious is when you have problems which are only reproducible in the release build and not in the debug build. Since there are no symbols in the release build it can be very hard to resolve the problem.

Fortunately this is easily resolved. It is relatively easy to change the project settings to generate symbolic information for the release build as well as for the debug build. An alternative approach is to abandon (or at least modify) the Debug/Release split, perhaps material for another article!

For Microsoft.NET 2003 C++ you enable symbols in release build by setting options for the compile and link stages. First set 'Debug Information Format' to 'Program Database' in the C/C++ 'General' folder. Then set the linker settings Generate Debug Info to 'Yes' in the 'Debugging' folder, and specify a `.PDB` filename for the program database file name. Finally you must set 'References' to 'Eliminate Unreferenced Data' and 'Enable COMDAT Folding' to 'Remove redundant COMDATs' in the 'Optimization' folder because the Microsoft linker changes its default behaviour for these two options when debugging is enabled. (Settings exist in other versions of the Microsoft C++ compiler, and also for VB.NET and C#. See [Robbins] for more details.)

I also recommend removing other one optimisation setting, that of stack frame optimisation, to greatly improve the likelihood of being able to get a reliable stack trace in a release build. If performance is very important in your application, measure the effect of this optimisation to see whether it makes a sufficient difference to be worth retaining.

With these settings applied to a release build the compiler generates a PDB file for each built EXE or DLL, in a similar manner to the default behaviour for a debug build. The PDB file, also known as the symbol file, is referred to in the header records in the EXE/DLL but none of the symbols are loaded by default, so there is no impact on performance simply having a PDB file.

The Symbol Engine

Microsoft do not document the format of the PDB file and it often seems to change from release to release. However they do provide an API for accessing most of the information held in the PDB file and the key to this is a file `DbgHelp.dll`. This library contains functions to unpack symbol information for addresses, local variables, etc. A version of this DLL is present in Windows 2000, XP and 2003 but Microsoft make regular updates available via its website as 'Debugging tools for Windows' [DbgHelp]. Note that if you want to write code using the API you need to install the SDK (by using the 'Custom' installation).

However it is hard to update `DbgHelp.dll` in place in a running system (and attempts to do so can render some other Windows tools inoperable) so it is recommended that you either:

- ensure the correct version of the DLL is placed with the EXE which is going to use it , or
- load the DLL explicitly from a configured location.

Personally, I find both these solutions cause unnecessary complications so I simply copy the DLL to `DbgCopy.dll` and generate a corresponding `Dbgcopy.lib` file from this DLL, which is included at link time. The makefile included in the source code for this article has a target `dbgCopy` which builds this pair of files.

The debug help API usually expects to find the PDB file for a binary EXE/DLL by looking for the file in its original location, or along the path. However the Debugging Tools for Windows package also contains a DLL that can connect to a so-called 'Symbol Server' to get the PDB file. Microsoft provide a publicly accessible symbol server containing all the symbols for the retail versions of their operating system, which lets you get symbolic names (and improved stack walking) for addresses in their DLLs. This is invaluable when you get problems inside a system DLL; usually, but not always, caused by providing it with bad data!

This DLL, `SYMSRV.DLL`, is activated by setting the environment variable `_NT_SYMBOL_PATH` to tell DbgHelp to use the symbol server. Note that this only works correctly if the `DbgHelp.DLL` and `SymSrv.DLL` are both loaded from the same location and are from the same version of 'Debugging Tools'.

The environment variable can be set from the command line for the current windowed command prompt, or more typically set via the control panel for the current user or even for the current machine. An example setting to load symbols from the Microsoft site is using a local cache in `C:\Symbols` is:

```
set _NT_SYMBOL_PATH=SRV*C:\Symbols*
     http://msdl.microsoft.com/download/symbols
```

There are a couple of problems with this simple approach. Firstly, the Microsoft site may not be available (for example, a company firewall may not grant access to the location specified) so the symbols for system DLLs are inaccessible to the symbol engine. Secondly, the symbol engine tries to access the Microsoft site for every EXE or DLL that it loads for which it cannot find local symbols. This can take quite a long time if have many DLLs that do not have any debugging information.

As an alternative you can set up the path as above and use the `Symchk` program to load symbols for a number of common DLLS (for example `KERNEL32`, `MSVCRT`, `NTDLL`), and then remove the `http://...` portion of the environment variable to just access the local cache.

A more advanced technique which is also available is to set up a symbol server, running on your own network. You can then publish symbol files, built in-house or arriving with third party libraries, to this symbol server for use throughout your company without needing to explicitly install them on every machine.

Using the Symbol Engine

I present some basic code to use the symbol engine, show how to convert an address to a symbol and show a simple example of the stack walking API. Please refer to the help for the debugging DLL (provided with the Debugging Tools SDK - `DbgHelp.chm`) for more information and description of other methods that I am not covering in this introductory article.

The symbol engine needs initialising for each process you wish to access. Each call to the symbol engine includes a process handle as one of the arguments, this does not actually have to be an actual process handle in every case but I find it much easier to stick to that convention. Calls to initialise the symbol engine for a given process 'nest' and only when each initialisation call is matched with its corresponding clean up call does the symbol engine close down the data structures for the process.

Note: there are a small number of resource leaks in `DbgHelp.dll`, some of which are retained after a clean-up, so I would advise you to try and reduce the number of times you initialise and clean up the symbol engine. My simple example code uses the singleton pattern for this reason.

Here is a class definition for a simple symbol engine:

```
/** Symbol Engine wrapper to assist with
    processing PDB information
*/
class SimpleSymbolEngine
{
public:
    /** Get the symbol engine for this process
    */
    static SimpleSymbolEngine &instance();

    /** Convert an address to a string */
    std::string addressToString(
        void *address );

    /** Provide a stack trace for the
        specified stack frame
    */
    void StackTrace(
        PCONTEXT pContext,
        std::ostream & os );

private:
   // not shown
};
```

This class can be used to provide information about the calling process like this:

```
void *some_adress = ...;
std::string symbolInfo=
    SimpleSymbolEngine::instance().
        addressToString( some_address );
```

I've picked a simple format for the symbolic information for an address - here is an example:

```
    0x00401158 fred+0x56 at
                testSimpleSymbolEngine.cpp(13)
```

The first field is the address, then the closest symbol found and the offset of the address from that symbol and finally, if available, the file name and line number for the address.

Using the stack trace is more difficult as you must provide a context for the thread you wish to stack trace.

The context structure is architecture-specific and can be obtained using the `GetThreadContext()` API when you are trying to debug another thread.

```
CONTEXT context = {CONTEXT_FULL};
::GetThreadContext( hOtherThread, &context );
SimpleSymbolEngine::instance().
    StackTrace ( &context, std::cout );
```

You have to be slightly more devious to trace the stack of the calling thread since the `GetThreadContext()` API will return the context at the point when the API was called, which will no longer be valid by the time the stack trace function is executed.

One approach is to start another thread to print the stack trace. Another approach, which is architecture-specific, is to use a small number of assembler instructions to set up the instruction pointer and stack addresses in the context registers. You have to be careful if you wish to provide this as a callable method to ensure the return address of the function is correctly obtained, for this article I simply use some assembler inline.

Here is a simple way (for Win32) to use the symbol engine to print the call stack at the current location:

```
CONTEXT context = {CONTEXT_FULL};
::GetThreadContext(
    GetCurrentThread(), &context );
_asm call $+5
_asm pop eax
_asm mov context.Eip, eax
_asm mov eax, esp
_asm mov context.Esp, eax
_asm mov context.Ebp, ebp

SimpleSymbolEngine::instance().
    StackTrace( &context, std::cout );
```

In this case the tip of the call stack will be the `pop eax` instruction since this is the target of the `call $+5` which I use to get the instruction pointer.

Implementation Details

The constructor initialises the symbol engine for the current process and the destructor cleans up.

```
SimpleSymbolEngine::SimpleSymbolEngine()
{
   hProcess = GetCurrentProcess();
   DWORD dwOpts = SymGetOptions();
   dwOpts |=
       SYMOPT_LOAD_LINES |
       SYMOPT_DEFERRED_LOADS;
   SymSetOptions ( dwOpts );

   ::SymInitialize( hProcess, 0, true );
   }
   SimpleSymbolEngine::~SimpleSymbolEngine()
   {
       ::SymCleanup( hProcess );
   }
```

I am setting the flag to defer loads which delays loading symbols until they are required. Typically symbols are only used from a small fraction of the DLLs loaded when the process executes.

The code to get symbolic information from an address uses two APIs: `SymGetSymFromAddr` and `SymGetLineFromAddr`. Between them these APIs get the nearest symbol and the closest available line number/source file information for the supplied address.

```
std::string SimpleSymbolEngine::addressToString( void *address )
{
    std::ostringstream oss;

    // First the raw address
    oss << "0x" << address;

    // Then any name for the symbol
    struct tagSymInfo
    {
        IMAGEHLP_SYMBOL symInfo;
        char nameBuffer[ 4 * 256 ];
    } SymInfo = { { sizeof( IMAGEHLP_SYMBOL ) } };

    IMAGEHLP_SYMBOL * pSym = &SymInfo.symInfo;
    pSym->MaxNameLength = sizeof( SymInfo ) - offsetof( tagSymInfo, symInfo.Name );

    DWORD dwDisplacement;
    if ( SymGetSymFromAddr( hProcess, (DWORD)address, &dwDisplacement,  pSym) )
    {
        oss << " " << pSym->Name;
        if ( dwDisplacement != 0 )
            oss << "+0x" << std::hex << dwDisplacement << std::dec;
    }

    // Finally any file/line number
    IMAGEHLP_LINE lineInfo = { sizeof( IMAGEHLP_LINE ) };
    if ( SymGetLineFromAddr( hProcess, (DWORD)address, &dwDisplacement, &lineInfo ) )
    {
        char const *pDelim = strrchr( lineInfo.FileName, '\\' );
        oss << " at " << ( pDelim ? pDelim + 1 : lineInfo.FileName ) << "(" <<
                                                              lineInfo.LineNumber << ")";
    }
    return oss.str();
}
```

The main complication with the two APIs used is that both need the size of the data structures to be set up correctly before the call is made.

Failure to do this leads to rather inconsistent results. Particular care is needed for the **`IMAGEHLP_SYMBOL`** since the structure is variable size.

Note too that the documentation for DbgHelp refers to some newer APIs (`SymFromAddr`, `SymGetLineFromAddr64`) which do the same thing as these two.

I have used the older calls here since they are available on a much wider range of versions of the DbgHelp API.

The stack walking code sets up the structure used to hold the current stack location and then uses the stack walking API to obtain each stack frame in turn.

All the source code for this article is available at:
`http://www.howzatt.demon.co.uk/articles/SimpleSymbolEngine.zip`

```
void SimpleSymbolEngine::StackTrace( PCONTEXT pContext, std::ostream & os )
{
    os << "  Frame       Code address\n";

    STACKFRAME stackFrame = {0};

    stackFrame.AddrPC.Offset = pContext->Eip;
    stackFrame.AddrPC.Mode = AddrModeFlat;

    stackFrame.AddrFrame.Offset = pContext->Ebp;
    stackFrame.AddrFrame.Mode = AddrModeFlat;

    stackFrame.AddrStack.Offset = pContext->Esp;
    stackFrame.AddrStack.Mode = AddrModeFlat;

    while ( ::StackWalk(
       IMAGE_FILE_MACHINE_I386,
       hProcess,
       GetCurrentThread(), // this value doesn't matter much if previous one is a real handle
       &stackFrame,
       pContext,
       NULL,
       ::SymFunctionTableAccess,
       ::SymGetModuleBase,
       NULL ) )
    {
        os << "  0x" << (void*) stackFrame.AddrFrame.Offset << "  "
           << addressToString( (void*)stackFrame.AddrPC.Offset ) << "\n";
    }

    os.flush();
}
```

The code provided here is specific to the x86 architecture - stack walking is available for the other Microsoft platforms but the code to get the stack frame structure set up is slightly different.

The context record is used to assist with providing a stack trace in certain ‘corner cases’. Note that the stack walking API may modify this structure and so for a general solution you might take a copy of the supplied context record.

The stack walking API has a couple of problems. Firstly, it quite often fails to complete the stack walk for EXEs or DLLs compiled with full optimisation. The presence of the PDB files can enable the stack walker to continue successfully even in such cases, but this is not always successful. Secondly, the stack walker assumes the Intel stack frame layout used by Microsoft products and may not work with files compiled by tools from other vendors.

Conclusion

I hope that this article enables you to get better access to symbolic information when diagnosing problems in your code.

Various tools in the Windows programmer’s arsenal use the DbgHelp DLL. Examples are: the debugger ‘WinDbg’ from the Microsoft Debugging Tools, the pre-installed tool ‘Dr. Watson’, and Process Explorer, from `www.sysinternals.com`.

If you build symbol files for your own binaries, tools like these can then provide you with additional information with no additional programming effort.

You can also provide symbolic names for runtime diagnostic information in a similar manner to these tools with a small amount of programming effort. I have shown here a basic implementation of a symbol engine class you can use to map addresses to names or provide a call stack for the current process.

I intended it to be easy to understand both what the code does and how it works. This example can be used as a basis for more complicated solutions, which could also address the following issues:

- the code is currently not thread-safe since the DbgHelp APIs require synchronisation.
- the code only handles the current process, it can be generalised to cope with other processes. Incidentally this provides a good example of why the singleton is sometimes described as an anti-pattern!
- no use is made of the APIs giving access the local variables in each stack frame.

Happy debugging!

Roger Orr
`roger0@howzatt.demon.co.uk`

References

[Orr2004] ‘Microsoft Visual C++ and Win32 Structured Exception Handling’, *Overload 64*, Oct 2004

[DbgHelp] `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`

[Robbins] *Debugging Applications for Microsoft .NET and Microsoft Windows*, John Robbins, Microsoft Press

“The C++ Community” - Are We Divided by a Common Language?

by Alan Griffiths

Recent discussions on the WG2 “core” reflector[1] have involved considerable speculation about the proportions of the C++ community that employ different memory management strategies. While the comparative sizes of their constituencies are unclear it was apparent that both “smart pointers” and “garbage collection” have adherents and also that users of each approach had limited experiences of designs that deploy the other strategy.

Over the last month or so there has also been a newsgroup thread on c.l.c++.m - “Smart Pointers” - where there are clearly parts of the community on both sides of the question “are reference counted smart pointers a useful design option”? (And at the time of writing “garbage collection” isn’t one of the alternatives that has been presented.)

There is a difference between these discussions - the protaganists on the “core” reflector appear to be intent on learning from each other’s experience, those on the newsgroup in evangelising. (Guess which is the more interesting to follow.)

There are many problems that must be addressed by C++ developers and, while the issues faced will vary from developer to developer, memory management is an issue that must be addressed by almost all.

The example set by the “core” working group is too little and too late: that the C++ community is failing to share and to build upon the experience of employing different memory management strategies signifies that there is something wrong.

It Matters That Developers Can’t Talk to Each Other

A passage from “*The IT Team*” [Lees03], reproduced in Figure 1, illustrates why it matters that developers communicate ideas.

A project balkanised according to the dialects its codebase is being written in isn’t going to run as smoothly or efficiently as one that is unified in its approach. The costs are often hidden - developers may be reluctant to touch “foreign” code, reuse

“Tuckman (1965) identified four developmental stages for a team to which Tuckman and Jensen (1977) added a final stage of adjourn. The figure provides an adaptation of that model which attempts to illustrate that:-

- *A team may spend an undetermined and variable amount of time at each stage*
- *May not reach all stages*
- *May regress to previous stages.*

“It is estimated that three fifths of the length of any team project, from start to finish, is taken up with the first two stages, Forming and Storming.”

(Robbins, Finley 1996 p.191).

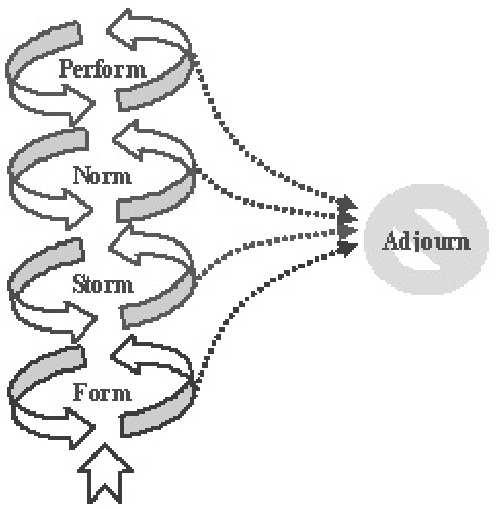
“As these stages are non-productive the ease with which a team moves through them will have significant impact on the team’s productivity.”

There is an ability to avoid or work through group conflict and an understanding of the strengths and weaknesses of team members. People adapt for the betterment of the team.

Conflict is avoided. There is a sense of cohesion. More realistic parameters are set for behaviour and performance.

There is a conflict amongst group members, choosing sides and bids for power. Goals may be unrealistic.

The team goes through an orientation phase in which people famiilarise themselves with the task and acceptable group behaviour is established.



Adaptation of the stage model of team development (Tuckman and Jensen 1977)

Figure 1 - A Passage from “The IT Team”

1 “Core” is the group working on the part of the C++ standard that defines the language.

opportunities may be missed, barriers are raised to refactoring - all things that can be attributed to other causes.

I've seen it - in a codebase of only 50KLOC I've seen four date+time classes, three classes for parsing XML and a mass of utilities for converting strings from one implementation to another.

C++ Dialects

The C++ language deliberately sets out to support a wide range of usage styles, and a number of usage styles are easily identifiable:

- Procedural programming ("a better C")
- Object based programming ("Data abstraction")
- Object-Oriented programming
- Generic programming
- Embedding of domain specific languages
- Template metaprogramming

Even aspects of functional or generative programming can be tackled, although these are generally deployed to support the above approaches. While this flexibility provides the opportunity to select the most appropriate and effective approach in any given context, it also provides plenty of scope for selecting an inappropriate approach for the problem in hand.

The downside to this wide ranging support for different styles of use is that the developer community has split into groups using distinct dialects. Does this matter? I think it does, because it impacts the ability of C++ users to communicate with one another. While dialects do develop in other languages, C++ seems to be unique in its propensity for them to wreck a project, dividing developers into groups that do not communicate well.

In "Is high-church C++ fair?"[Kelly01] Allan Kelly describes his experience of multiple dialects in the organisation he was working in. Allan describes four dialects (I'm sure we've all seen them) as follows:

"High-church – "the kind of C++ you see in C++ Report" : as I said above, templates, standard library, interface inheritance more important than implementation inheritance, full use of language features and new keywords

Low-church – "the kind of C++, C programmers write up to the kind of C++ in the MFC" : few templates, roll your own data structures, lots of void casts, implementation inheritance more important than interface inheritance

C – "the kind of C++ which is just C with classes" : we can probably all recognise this, and probably wrote a bit of it ourselves in the early days. However, after a short while we are either high-church C++, low-church C++ or, last but not least.....

Bad C++ - "where the developer writes C and has picked up a few bits of C++ and doesn't realise what they are doing" : no understanding of copy constructors, inheritance (of all kinds!) used without a clear reason, destructors releasing resources before they are finished with, no standard library, lots of void casts, unnecessary and inappropriate file dependencies."

Allan goes on to discuss the impact these have on his work.

"Once again it has come time for me to hand over a project and move on. This is a fairly big project, very important to the company, and, as I'm leaving the company it's important that the hand-over goes smoothly.

The code should be readable to most Overload readers (sorry, no examples), and most would not look out of place in C++ Report, in other words, it's what I call "high-church C++", it uses the language to the full, is littered with design pattern references (indeed, the main loop is a command pattern), a few templates, plenty of standard library, and inheritance is used mainly for interface not implementation inheritance.

…

I have deliberately used the terms high-church and low-church, not only because of well know religious programming wars but because both sides have their arguments (high-church claims to be more object oriented, while low-church claims to be more understandable to average programmers) and because it is easy for high-church to look down on low-church as less than true, while low-church regards high-church as fanatical.

…

Returning to my own project, if I where to take over a similar one, or someone of my experience where to take over this project I do not think they would have a problem. Indeed, they would probably pick up the code and understand it quiet quickly, the abstractions should mean you can understand the high-level side without understanding the details. Only when the details change do you need to understand them."

While these comments focus on only two of the dialects they do illustrate the divisions that can occur between the corresponding congregations Not only does each group have difficulty understanding the other, each also believes that theirs is the correct way to use the language. What is also interesting from this description is the implication that the team in which Allan was working does not seem to have resolved these differences and remains in the "storm" phase of team development.

The Origin of Dialects

I hope that I've demonstrated that C++ dialects matter - because they can dramatically reduce the effectiveness of the team. But where do these dialects come from? And why is C++ particularly prone to these problems?

Developers frequently move to C++ from another language and bring with them an approach that fits that language well. By design C++ allows them to do this - and, in doing so, fails to encourage an evaluation of the appropriateness of their approach in C++. As a result the developer works in a creole[2] formed of these two languages. In the early days of C++ the C/C++ creole was commonplace, but Java/C++ has gained recently.

Even developers coming first to C++ are not immune to this effect - they can inherit it from authors, trainers and colleagues. Kevlin Henney in "The miseducation of C++" [Henney01] identifies three families of C++ dialects and goes on to relate them to the way that C++ is taught.

"Early C++: The first public incarnation of the language formed part of a growing interest in the use of object-oriented development

2 A creole is a language descended from a pidgin that has become the native language of a group of people.
`http://en.wikipedia.org/wiki/Creole_Language`

in industry. Early C++ was a minimal language, extending C with a stronger type system, and the notion of classes with public and private access, single inheritance and operator overloading.

Classic C++: This is the C++ that many long-time C++ programmers know, and many new C++ programmers are still effectively being taught. It brought abstract classes, multiple inheritance and further refinements into the language. Classic C++ also introduced templates and exceptions, although they took longer to mature than other, more obviously evolutionary features. Where templates are used, it's normally to articulate container classes and smart pointers. Classic C++ saw the greatest proliferation in dialects as it was in this period that the language underwent standardisation. Changes ranged from the trivial to the major, from bool to namespace, from the proper incorporation of the standard C library to the internationalisation overhaul of I/O. Many projects still bear the scars of this portability interregnum.

Modern C++: This is where we are now. The language was standardised in 1998, providing a fixed target for vendors and a consolidation in thinking and practice. Exception safety is now a well understood topic and the Standard Template Library (STL) part of the library has introduced the full power of generic programming to the language, raising templates from being merely a way to express containers to an integral part of design expression in C++."

Despite the implication of a timeline in this classification, the dialects of Early C++ have not been assigned to the dustbin of history. They are still living languages: I've recently had reports of a heavily MACRO based style of C++ in one organisation that sounds as though it belongs to this group.

And "Modern C++" isn't the final, definitive way to use the language: there are also dialects that move beyond the scope of Modern C++ and into generative techniques. These have their place but, like anything else, when seen as a "shiny new hammer" the effect is to introduce complexity that serves only to postpone necessary design decisions and introduce uncertainty.

What the "timeline" does reflect is that there has been an evolution in the thinking regarding effective ways of using C++: the "later" dialects are the result of learning from the use of the earlier ones. As a consequence they generally avoid problems that arise in using the "earlier" dialects and, in consequence, are more capable of handling complexity. Thus resource management is problematic in Early C++ dialects, requiring constant attention, while Classic and Modern dialects incorporate idioms that free the developer from this constant distraction. (While this evolution of the idioms in contemporary usage provides for better solutions to harder problems it also creates a difficulty - a style that was considered "best practice" a few years ago might now be dismissed as misguided or sub-optimal.)

C++ Accents

There are also some more superficial elements that contribute to the separation of dialects in C++. Unlike the factors that separate Early, Classic and Modern C++ (or High Church/Low Church C++) these don't have a direct impact on the technical effectiveness of the language - and they can be found in dialects of all eras. But they are highly visible, and present a barrier to communication between their adherents:

- lower_case or CamelCase
- brace placement
- prefixes and suffixes
- …

I'm sure we all have preferences - mine is for consistency. (As I like to use the standard library and boost, I like to be consistent with the style these adopt.)

Solutions

The range of C++ dialects is vast - Allan describes four major groupings and Kevlin three, but a closer examination would probably find at least one for every C++ project in the world. If different developers in a team are comfortable with different dialects then these incompatibilities between them can make it very hard for the team to reach the performance levels it would otherwise be capable of.

Later in "Is high-church C++ fair?" Allan addresses one "solution":

"So may be my question is should really be turned onto management, the developers they hire, appoint to projects and training courses they use. But, most manages [sic] (well those I have encountered) gave up programming when they got the keys to the company car. If I actually asked them what I should write, I think I would be told to write low-church C++, but this would take me longer (more code would need to be written and more testing) and I do think it would be less maintainable in the long run. But without two individuals writing the same project in low and high C++ how can we quantify this?"

Allan returns to the subject in *Overload 65*: his article "The Developers New Work" [Kelly05] examines a number of non-solutions:

"We could "dumb down" our code, make it really simple. Trouble is, we have real problems and we need real solutions. To tackle the same problem with "low Church code" just moves the complexity from the context to an overly verbose code base.

We could just hire real top-gun programmers. This isn't really a solution; once again we're pushing the problem down in one place and seeing it come up in another. Since there aren't that many super-programmers in the world finding them is a problem, keeping them a problem, motivating them is a problem and even if we overcome these problems it's quite likely that within our group of super-programmers we would see an elite group emerge.

Hiring a group of super-programmers is in itself an admission of defeat, we're saying: We don't know how to create productive employees; we're going to poach people from companies who do. In doing so we move the problem from our code to recruitment.

So, maybe the solution is to get management to invest more in training. But this isn't always the solution. The mangers I had when I wrote High Church C++ tried to do the right thing. Is it not reasonable to assume that someone who has been on a C++ course can maintain a system written in C++? As Alan Griffiths pointed out, this is about as reasonable as expecting someone that has been on a car maintenance course to change the tires during an F1 pitstop.

The answer of course is: No, knowing C++ is a requirement for maintaining a C++ based system but it isn't sufficient of itself. One needs to understand the domain the system is in and the system architecture - this is why Coplien and Harrison say you need a whole year to come up to speed.

How do we communicate these things? The classical answer is "write it down" but written documentation has its own problems: accuracy, timeliness, readability, and memorability to name a few. In truth, understanding any modern software system is more about tacit knowledge than it is about explicit knowledge."

Well, that tells us a lot about what not to do, but in real projects we still need an answer.

Given my metaphor, dialects of spoken languages, I'm sure that my solution won't be a surprise to you. One can learn something of a language from dictionaries and grammars, but the only way to become fluent is to use it to communicate with others. That means reading and writing code - in the appropriate dialect - and discovering if it is understood by other developers.

Allan's answer [Kelly05] is similar:

"It is no longer enough to just cut-code. Sure you may need to do this too, but if you want to use modern C++ (or modern Java, Python, or what ever) it is your job to lead others in a change. And change doesn't happen without learning. Indeed, learning isn't really happening if we don't change, we may be able to recite some piece of information but unless we act on it we haven't really learnt anything.

So, when it comes to improving your code it isn't enough to sit your colleagues down and tell them that a template-template function is the thing they need here and expect them to make it so. You've imparted information, you may even have ordered them to do it, but they haven't been led, they haven't learnt and they won't have changed - they'll do the same thing all over again.

Simply informing people "This is a better way" doesn't cut it. You can't lecture, you can't tell, you can't enforce conformance. You need to help others find their own way to learn. Helping them find that way goes beyond simply giving them the book, they need to be motivated, people who are told aren't motivated, people who are ordered aren't motivated; motivating people requires leadership.

…

Hope lies not in code, not in machines but in people. If we believe that Modern C++ is best - and I truly, rationally, believe it is - then I have no choice other than to develop the people around me - and that belief is rational too."

I've more experience with this approach than Allan appears to have, and there is a problem that he appears unaware of: ten years (or so) ago I introduced my colleagues to elements of the latest thinking on how to use C++ - they were new to the language and I followed the literature. They adopted many of these idioms - things like "smart pointers" to managed object persistence, even some bits of STL. Then I left the organisation.

I'm still in touch with the development group and some of the developers there. Ten years ago "Classic C++" was the best way we knew to use the language, but C++ is a living breathing language and new idioms are being formed all the time. When I left these developers were no longer exposed to the "latest thinking" - like most of the industry they didn't "do books" (DeMarco and Lister - in "Peopleware" - report that the typical developer reads fewer than one technical book a year). They were not motivated to seek out new solutions in a subject where they were not experiencing problems. Nowdays a dialect of Classic/Low Church C++ is deeply entrenched and Modern/High Church C++ has passed them by. By failing to engage them directly in the wider community I left a situation where a local dialect was going to develop over time.

The lesson I take from this is that the goal should be more ambitious than ensuring a lingua franca within the team, more than introducing a contemporary dialect of C++. Developers must be exposed to developing ideas and engaged in the community developing them.

So What Works?

I don't have a magic bullet - I doubt that one exists. The different dialects of C++ have developed in response to genuine circumstances.

However, in the context of an individual project or team there is a range of things that, if employed sensibly, can make things better rather than worse:

- Coding guidelines? - These are safe if they deal with "accents", but can entrench obscure and obsolete dialects unless they are carefully reviewed, living documents that change as ideas are revised. See `http://groups.yahoo.com/group/boost/files/coding_guidelines.html` for a good example.
- Training courses? - These can expose those that attend to new idioms, but they cannot ensure they are either internalised or adopted by others.
- Code Reviews? - If properly conducted these can help the team settle on a common dialect, or at least become familiar with those in use. On the other hand they can be a cause of friction and entrenchment.
- Pair programming? - I've heard good reports of this - and bad. My experience does not include it working as a routine practice but I've been convinced that it can produce good results "when done properly".
- Coaching/mentoring? - Good coaches/mentors are hard to find. Bad ones are worse than useless.
- Reading and writing about C++? - People that read books and other publications are aware of wider community, but only those that participate (in newsgroups, at conferences, or writing books or for magazines) are truly involved in the community.

Conclusion

Divisions in the C++ community can cost in a number of ways, the most obvious being the inability of team members to communicate in the language. Another cost, that is easier to overlook, is that as techniques are developed to address the problems that developers face, they are not known outside the part of the community that develops them. This can lead to the "re-inventing of wheels" - I've lost count of the different smart pointers I've encountered (most of them with avoidable bugs). Even worse, it can lead to "living with" avoidable problems - the typical C++ program leaks resources despite the fact that RAII (and the techniques for implementing it) have been known to some parts of the community for a long time.

[concluded at foot of next page]

The Trial of the Reckless Coder

by Phil Bass

An Arrest Is Made

"Joe Coder, I'm arresting you on suspicion of coding without due care and attention, and with reckless disregard for the welfare of other code users." I had said this many times before, but this time I was uneasy. Something in Joe's eyes suggested there might be more to this case than I'd imagined.

I hesitated, suddenly unsure of the words of the standard caution. "You do not need to say anything, but anything you do say will be written down and may be used in evidence against you." It was the old caution, the one you heard in films and T.V. programmes. I blundered on, hoping no-one would notice. "Take him away, sergeant", I said a little too loudly. At least I'd resisted the temptation to say "Book 'im, Danno!"

Interrogation

My head was still a little fuzzy from the previous night. Not enough water in my nightcap, probably; or too much whisky. As we prepared to interrogate Joe I had forgotten my qualms about the `Bound_Callback<Pmf>` class template [1]. "Did you write this?" I asked, showing Joe the following code:

```
class Observer
{
public:
    Observer(Event<int>& event)
      : callback(bind_1st(memfun(&Observer::handler), this))
      , connection(event, &callback)
    {}

private:
    void handler(int)
    {
        clog << "Observer's event handler." << endl;
    }

    typedef
        Callback::Adapter<void (Observer::*)(int)>::type
        Callback_Type;

    Callback_Type             callback;
    Callback::Connection<int> connection;
};
```

Exhibit 1 - The evidence for reckless coding.

"What if I did?" said Joe. "Look, son", I said gently, "the copyright notice has your name on it and the version control log says you checked it in, so there's no use denying it." Joe shrugged. He wasn't going to make it easy for me, but the evidence was solid. "That's a lot of boilerplate code", I continued. "Is it?" Joe responded. His voice sounded innocent, but his eyes were defiant. I wasn't going to waste time in a verbal sparring match, so I showed him a shorter, simpler alternative:

[continued from previous page]

Naturally, everyone that reads this is making an effort to be part of the solution instead of being part of the problem. However, I urge you to make an effort to be more involved and to seek ways to involve your colleagues in the wider community.

Alan Griffiths
alan@octopull.demon.co.uk

References

[Kelly01] "Is high-church C++ fair?", Allan Kelly, http://www.allankelly.net/writing/WebOnly/HighChurch.htm

[Lees03] *The IT Team*, Sarah Lees 2002

[Henney01] "The miseducation of C++", Kevlin Henney, Application Development Advisor, April 2001, http://www.two-sdg.demon.co.uk/curbralan/papers/TheMiseducationOfC++.pdf

[Kelly05] "The Developers New Work", Allan Kelly, *Overload 65*:

Tuckman B (1965) in Rickards (2000). 'Development Sequence in Small Groups'. Psychological Bulletin Vol.63/6 pp.384-399.

Tuckman B and Jensen M (1977) in Rickards (2000). 'Stages of Small Group Development Revisited'. Group and Organizational Studies, Vol.2 pp.419-427.

```
class Observer
{
public:
    Observer(Event<int>& event)
      : callback(event, &Observer::handler, this)
    {}

private:
    void handler(int)
    {
        clog << "Observer's event handler." << endl;
    }

    Bound_Callback<void (Observer::*)(int)> callback;
};
```

Exhibit 2 - Showing consideration for other code users.

Joe humphed. He was obviously unimpressed. It was a puzzling reaction, but I pressed on. “That `Bound_Callback` template makes things much easier for the author of the `Observer` class, doesn’t it?” Joe said nothing, but the expression on his face challenged me to prove my point. “Doesn’t it?” I yelled, banging the desk with my fist. “Perhaps”, said Joe, unflustered.

“Perhaps?” I exclaimed. “There’s no ‘perhaps’ about it, my lad...” Joe interrupted me. “Look, granddad, you haven’t given me a specification for the `Bound_Callback` template, so how can I tell whether it’s useful?”

I had to admit he had a point and wrote this on the whiteboard:

Intent

A bound-callback is a callback that is bound to an event for the whole of its life. The `Bound_Callback<Pmf>` class template implements the bound-callback concept for the common case in which the callback invokes a member function.

Synopsis

```
template<typename Pmf>
struct Bound_Callback
  : Callback::Function<typename argument<Pmf>::type>
{
    typedef typename argument<Pmf>::type Arg;
    typedef typename   result<Pmf>::type Result;
    typedef typename   class_<Pmf>::type Class;

    Bound_Callback(Event<Arg>&, Pmf, Class*);
   ~Bound_Callback();

    Result operator()(Arg);
};
```

Note: Here, `argument<Pmf>`, `result<Pmf>` and `class_<Pmf>` are simple class templates with a nested `typedef`. They are known as meta-functions because they take a type parameter (the pointer-to-member-function type) and ‘calculate’ another type (the argument type, result type and class of the member function). Further information on meta-functions in general and these meta-functions in particular can be found in [1].

Types

`Pmf` is the type of a pointer to the member function to be invoked.
`Arg` is the type of the parameter of the member function to be invoked.
`Result` is the type of the result of the member function to be invoked.
`Class` is the class of which the function to be invoked is a member.

Constructor

```
Bound_Callback(Event<Arg>& event, Pmf pmf, Class* ptr);
```

Stores a reference to `event`; copies and stores the pointer-to-member-function `pmf` and the pointer `ptr`; connects `*this` to `event`.

Destructor

Disconnects `*this` from `event`.

Function Call Operator

```
Result operator()(Arg arg);
```

Invokes the member function pointed to by `pmf` on the object pointed to by `ptr` passing the value `arg` and returning any result.

I was quite pleased with myself. Joe, however, remained unimpressed. “Doesn’t seem very flexible”, he remarked. “What do you mean?” I said tetchily, “There’s nothing inflexible about that!”

It was a mistake. Joe stood up, his eyes sparkled and he attacked my whiteboard specification with vigour. “Well, for a start”, he said, “the callback function’s argument has to be exactly the same type as that published by the event.”

He scrawled the following example on the whiteboard:

```
// Try to connect handler(int) to Event<short>
class Observer
{
public:
    Observer(Event<short>&)
      : callback(event, &Observer::handler, this)
    {}  // error!

private:
    void handler(int);

    Bound_Callback<void (Observer::*)(int)> callback;
};
```

Exhibit 3 - Implicit conversions not supported.

“The event publishes short values, the handler function accepts `int` values and there’s an implicit conversion from `short` to `int`. You’d expect it to work, but it doesn’t.” There was a hint of triumph in Joe’s voice. He must have seen the puzzled look on my face because he carried on, patronisingly. “Look, granddad, the `Bound_Callback<Pmf>` constructor takes an `Event<int>&` as its first argument and we’re giving it an `Event<short>&`. There’s no implicit conversion from `Event<short>&` to `Event<int>&`, so you get a compilation error.”

He was right. I’d been drawn into an intellectual boxing match and I was losing on points. Instinctively, I began to defend the `Bound_Callback<Pmf>` code. “Yeah, but that’s outside the scope of the `Bound_Callback<Pmf>` specification”, I countered. Joe smiled smugly. “My point, precisely”, he sneered.

The Case for the Defence

I remembered Joe’s smugness when the case came to trial and his defence counsel addressed the jury. “You have heard the prosecution allege that the defendant has been coding with reckless disregard for the welfare of other code users. By their own admission, their case rests almost entirely on the evidence presented in Exhibit 1 and the alternative interface in Exhibit 2 that they have described as more “user friendly”. The defence, however, will show that Exhibit 2 is only useful in a rather narrow range of programs - much narrower, in fact, than Exhibit 1. It is our contention that a tool of limited applicability shows greater disregard for code users than one that, in the spirit of C++, allows the user full freedom of expression.”

The defence barrister was George Sharpe, an old adversary, and he was revelling in the usual obscure style of language used in courtrooms across the world. My attention began to wander. I couldn’t help feeling sorry for the jury. Did they understand what the lawyer was saying? Did they care? Perhaps they were thinking about what they would have for lunch or whether the neighbour had remembered to take their dog for a walk. Suddenly, I realised I had been called for cross-examination.

“Detective Inspector Blunt, how would you define an Event?” Sharpe always starts with an easy question. I gave the stock answer, “An Event is a sequence of pointers to functions.” The barrister repeated my answer as a question. “A sequence of pointers to functions?" He says he does this for the jury's benefit – to be sure they understand. But I think it's just to annoy me. "That is the standard definition", I said. "I see. And this covers all kinds of sequence, all kinds of pointer and all kinds of function, does it?" I answered in the affirmative and Sharpe's questions flowed smoothly on. "So, for example, a sequence might be a list or a set or an array?" Before I could answer Sharpe fired another salvo. "A pointer might be a built-in pointer, an `auto_ptr` or a `shared_ptr`? And a function might be a member function, a non-member function or a function object?" He wasn't giving me time to think and I sensed he was preparing a trap. I responded with a cautious, "In principle, yes."

There was a pause while Sharpe presented Exhibit 4 to the court.

```
// A bound callback that calls a member function.
template<typename Pmf>
class Bound_Callback
  : public Callback::Function<typename argument<Pmf>::type>
{
public:
    typedef typename   result<Pmf>::type Result;
    typedef typename   class_<Pmf>::type Class;
    typedef typename argument<Pmf>::type Arg;

    Bound_Callback(Event<Arg>& e, Pmf f, Class* p)
      : event(e), function(f), pointer(p),
        position(event.insert(event.end(), this))
    {}

   ~Bound_Callback() {event.erase(position);}

    virtual Result operator()(Arg arg)
    {
        return (pointer->*function)(arg);
    }

private:
    Event<Arg>& event;
    Pmf         function;
    Class*      pointer;

    typename Event<Arg>::iterator position;
};
```

Exhibit 4 – A bound callback for member functions.

"Now, Detective Inspector, I am sure you recognise this piece of code." Sharpe placed a sheet of paper in front of me and continued, "It is the code you wrote to demonstrate that a bound callback with a user-friendly interface can be implemented, is it not?" I nodded and Sharpe pressed on before the judge could remind me to answer so that the court could hear.

Turning to the jury he explained that the test program for Exhibit 4 used an `Event<int>` that is a list of built-in pointers to the abstract base class `Callback::Function<int>`. Then, turning back to me, he asked "Does this code work for Events that are sets of pointers to functions?" "No" I replied steadily. "No? Well then, does it work for arrays of pointers to functions?" I did not respond. "Does it work for lists of smart pointers to functions? Does it work for lists of pointers to ordinary C++ functions, or function objects not derived from `Callback::Function<int>`? Does it work in any of these situations, Detective Inspector?" I started to say that it was never intended to be that general, but Sharpe cut me off. "Yes or no" he demanded. I looked to the prosecution team for support, but they didn't stir. I turned towards the judge to appeal against this line of questioning, but he was entirely unsympathetic. "Yes or no, Detective Inspector Blunt?" thundered Sharpe. "No, it doesn't support those uses." I confessed.

A Brief Interlude

I was livid. It looked as though Joe Coder might get off on a technicality. More importantly, I had been made to look a proper Charlie. At the next recess I called the team together and demanded to know why no-one had anticipated Sharpe's underhand tactics. There were no answers. There were plenty of sheepish faces and lots of excuses, but no answers.

There was an awkward silence. A young police constable tried to cheer me up. "Sir?" I turned and raised an eyebrow, inviting her to continue. "Sharpe was talking about `auto_ptr` and `boost::shared_ptr` - smart pointers that manage the lifetime of their target object." I didn't see how this was helping. "Well, sir, that makes no sense. The `Bound_Callback<Pmf>` classes are designed to be members of the `Observer` class, so their lifetime is tied to the `Observer`'s. And, anyway, you can't put `auto_ptrs` in standard library containers." "I know that, Jenkins", I snapped, "you, know that and Sharpe certainly knows that, but do you think the jury realises it?"

The Judge's Summing Up

Justice Bright is renowned for his clear and insightful summing up. He reminded the jury that the charge was coding without due care and attention and with reckless disregard to the welfare of other code users. "In this case, the intentions of the defendant are crucial." he explained. "If he believed his fellow programmers would be best served by generic, but somewhat verbose facilities for bound callbacks you must find him not guilty. If, on the other hand, he thought that a more limited, but more user-friendly facility was better you must find him guilty as charged. In reaching your verdict you do not need to consider the defendant's competence as a programmer. Better and worse solutions to the bound callback problem undoubtedly exist, but we can not condemn a man for failing to reach perfection nor is it the responsibility of this court to apply remedies for lack of ability or training."

The Jury Retires

The judge's words were going round in my head as we waited for the jury to consider their verdict. The case would be decided by the jury's assessment of the defendant's character, but the technical question behind the trial remained unresolved. Exhibit 1 illustrates a general, but verbose mechanism for implementing bound callbacks; Exhibit 2 shows a more limited, but easier-to-use alternative. Could there be a bound callback that is both fully generic and easy to use?

I tried to imagine what that perfect bound callback would look like, but it was too big a problem to solve all at once.

Then I considered some specific Event types:

1. a list of pointers to abstract function objects (the common case),
2. an array of pointers to simple functions (the simplest case) and
3. a set of shared pointers to function objects (associative container + user-defined pointer).

What would callbacks bound to these types of Event have in common? How would they differ? And how would they invoke handler functions of any compatible type? I fell into deep thought…

In case 1 the bound callback has to provide a concrete implementation of the abstract function, its constructor should insert a new pointer into the list and its destructor should erase the pointer from the list. This is a generalisation of the code shown in Exhibit 4. Sketch 1 illustrates the idea.

```
template<typename Event, typename Function>
class Bound_Callback
{
public:
    Bound_Callback(Event& e, const Function& function)
      : event(&e)
      , adapter(function)
      , position(event->insert(event->end(), &adapter))
    {}

    ~Bound_Callback() {event->erase(position);}

private:
    Event*                                 event;
    typename adapter<Event,Function>::type adapter;
    typename iterator<Event>::type         position;
};
```

Sketch 1 – A bound callback for a list of pointers to abstract functions.

Here, adapter is a meta-function that generates the type of a concrete function object and iterator is a meta-function that generates the list iterator type. This much was clear to me while waiting for the jury's verdict; the precise details could be worked out later.

Case 2 is very different. No function object is necessary and arrays do not support insert or erase operations. Instead, the callback's constructor could assign a function pointer to an array element and the destructor could reset that array element to a pointer to a no-op function. This would look something like Sketch 2:

```
template<typename Position >
class Bound_Callback
{
public:
    typedef typename   target<Position>::type Pointer;
    typedef typename   target<Pointer >::type Function;
    typedef typename   result<Function>::type Result;
    typedef typename argument<Function>::type Arg;

    static Result no_op(Arg) {}

    Bound_Callback(Position p, const Function& f)
      : position(p)
    {
        *position = &f;
    }

    ~Bound_Callback() {*position = no_op; }

private:
    Position position;
};
```

Sketch 2 – A bound callback for an array of pointers to functions.

Although they are small class templates there are several differences between Sketches 1 and 2. In particular, the template parameter lists are completely different. The data storage requirements and the constructor parameters are different, too. In fact the differences are large enough to call into question the whole idea of a single fully generic bound callback.

In case 3 a concrete function object may need to be created, the constructor would create a shared pointer and insert it into the set, and the destructor would erase the shared pointer.

```
template<typename Event, typename Function>
class Bound_Callback
{
public:
    typedef typename element<Event>::type Pointer;

    Bound_Callback(Event& e, const Function& f)
      : event(&e),
        position(event->insert(make_pointer<Pointer>(f)))
    {}

    ~ Bound_Callback() {event->erase(position);}

private:
    Event*                          event;
    typename iterator<Event>::type position;
};
```

Sketch 3 – A bound callback for a set of shared pointers to abstract functions.

This is similar to Sketch 1 except that a function object is created on the heap; the `make_pointer` function template creates the function object and returns a shared pointer to the new object. An intelligent `make_pointer` function could select an intrusive shared pointer if the user supplies a reference-counted function object, or a non-intrusive shared pointer otherwise.

In all three cases the bound callback might need to provide a function adapter to convert the value published by the Event to the type required by the user-supplied handler function. Sketches 1 and 3 already use a function adapter, so it can be extended for this purpose. In Sketch 2, a function adapter would have to be introduced specially.

[concluded at foot of next page]

Taming Complexity: A Class Hierarchy Tale

by Mark Radford

Introduction

Some years ago, I was involved in the design of the software for a security system[1]. The software had to process events from various sensor devices detecting motion in the building. However, consider the operation of such a system installed in a typical office building: obviously alarms shouldn't ring when people are in the building during normal working hours. Actually that's not strictly correct: there may be areas of the building that are normally off limits, and so should be monitored in normal working hours. Further, there may be cases when people are working during the night in certain parts of the building. Clearly this security system needed to be flexible – i.e. it needed a lot of configurability designing into it. To this end I had the task of designing a logic engine in support of configurability far beyond the simple description given above.

From this design comes the class hierarchy shown in Listing 1.

The **`trigger`** class represents a trigger event, i.e. an event that potentially participates in the triggering of an alert. Motion being detected in a certain area and a door that should be closed being open, are both examples of triggers going active. Further, a trigger can be a simple time interval, 1800-0800 for example – i.e. not during daytime office hours. In the domain terminology, such events are said to cause a trigger to "go active". It is possible that a single trigger event could trigger an alarm, motion being detected in an off-limits part of the building, for example. However, motion detected in certain (most) parts of the building between 1800 and 0800 – i.e. a combination of two trigger events – is the kind of combination of events requiring an alert to be raised. This is where the **`evaluator`** class hierarchy comes in. I don't propose to go any further into the problem domain analysis, but it turns out that when two trigger events are of interest, there are two cases where it may be necessary to raise an alert:

- The case where both triggers need to be active (logical AND), and
- The case where either or both triggers need to be active (logical OR)

The **`evaluator`** hierarchy shown in Listing 1 facilitates this, and also leaves open the possibility of extending the system to handle the logical XOR case. Note in passing, that I'm ignoring cases where an alert is raised if only a single trigger goes active – these do not form part of this illustration.

The C++ Three Level Hierarchy

I've used the **`evaluator`** class hierarchy and spent some time explaining it, because I like the idea of a realistic example to work with. The above comes from a real project, although what I've shown is a simplified version for illustration purposes.

I'm using this hierarchy as an example of the three level structures that are idiomatic in C++ class hierarchy design: interface class (see [Radford] for an explanation of the term) at the top, common (but abstract) implementation in the middle, and the most derived classes forming the (concrete) implementations.

However the tale of this hierarchy has a slight twist in it. Let me just digress for the moment and look at another three level hierarchy – one I trust will need no explanation.

The **`circular_shape`** hierarchy, shown in Listing 2 on page 26, uses the three level approach already discussed. However in this case, the **`common_circular_shape`** class (the middle class) captures state common across the concrete (most derived) classes, while the derived classes themselves have their own **specific** state. Also, note the constructors: there is a lot of repetition of code here, in fact the constructors of **`circle`** and **`ellipse`** differ in name only.

Compare the **`circular_shape`** and **`evaluator`** hierarchies. In the latter case, the concrete (most derived) classes have no state of their own – all state is common and captured in the common implementation. However, there is something **`and_evaluator`** and **`or_evaluator`** have in common with **`circle`** and **`ellipse`**: they are saddled with the repetitive constructor baggage. That is to say, the derived classes must each have an almost identical constructor just to initialise state that is kept entirely in a base class. The areas of variability in **`and_evaluator`** and **`or_evaluator`** are:

- The name of the constructor – everything else about the constructors is the same for both classes
- The two characters denoting logical operation in **`evaluate()`**– other than that, the implementation of this virtual function is exactly the same for both classes, and the same would apply if an **`xor_evaluator`** were ever to be added

1 That is, security of buildings, as opposed to networks.

[continued from previous page]

I began to think about merging these sketches into a single, truly generic bound callback. The three possibilities took on geometric shapes in my mind. They seemed to float before me, drifting around like globules of oil in a lava lamp, rising, falling, merging, splitting. It was relaxing, soothing.

Verdict

I woke up with a start. My sergeant's hand was shaking my shoulder. "Inspector, Inspector", he was saying. "They've reached a verdict." I roused myself and went back into the courtroom. The judge and jury filed in. The clerk of the court went through the ritual of asking if the members of the jury were all agreed. They were. "Do you find the defendant guilty or not guilty?" Perhaps it was my imagination, but the foreman of the jury seemed to pause for dramatic effect and then said in a clear and confident voice, "Not guilty."

So that was it. The prosecution I'd worked hard on for most of the last 6 months had failed to achieve a conviction. It seemed a waste. It wasn't a big case, but it had given my team a purpose, a reason to press on through the humdrum routine of police work, a feeling of doing something worthwhile. "Oh well, you win some, you lose some." I told myself and turned to leave.

Phil Bass
Phil@stoneymanor.demon.co.uk

References

[1] Phil Bass, "The Curious Case of the Compile-Time Function", *Overload 62*, August 2004.

Acknowledgements

I would like to thank Mary Opie for suggesting the names Blunt and Sharpe.

```
class evaluator
{
public:
  virtual ~evaluator();
  virtual bool evaluate() const = 0;
...
};

class common_evaluator : public evaluator
{
protected:
  common_evaluator(const trigger* t1, const trigger* t2)
  : one(t1), two(t2)
  {}

  const trigger& trigger_1() const { return *one; }
  const trigger& trigger_2() const { return *two; }

private:
  const trigger* const one;
  const trigger* const two;
};

class and_evaluator : public common_evaluator
{
public:
  and_evaluator(const trigger* t1, const trigger* t2)
  : common_evaluator(t1, t2)
  {}
private:
  virtual bool evaluate() const
  {
    return trigger_1().active() && trigger_2().active();
  }
};

class or_evaluator : public common_evaluator
{
public:
  or_evaluator(const trigger* t1, const trigger* t2)
  : common_evaluator(t1, t2)
  {}

private:
  virtual bool evaluate() const
  {
    return trigger_1().active() || trigger_2().active();
  }
};
```

Listing 1: `evaluator` Class Hierarchy

Everything else is the same between these two classes (and this also applies to a future `xor_evaluator`).

This suggests a need to investigate ways of simplifying the design. Herein is the subject matter for the rest of this article.

I will investigate two ways to separate the commonality and variability: one using delegation, and the other using static parameterisation using C++ templates.

Approach Using Delegation

This approach draws on the **STRATEGY** design pattern [Gamma et al]. First, let's deal with the `evaluate()` virtual function – I'm going to factor out `and_evaluator` and `or_evaluator`'s implementations into `and_operation` and `or_operation` respectively, and derive these from the interface class `operation`. The resulting hierarchy is shown in Listing 3 on page 27.

```
class circular_shape
{
public:
  virtual ~ circular_shape();

  virtual void move_x(distance x) = 0;
  virtual void move_y(distance y) = 0;

...
};
class common_circular_shape : public circular_shape
{
protected:
  circular_shape(const point& in_centre)
  : centre_pt(in_centre)
  {}
  point centre() const;
...
private:
  point centre_pt;
};

class circle : public common_circular_shape
{
public:
  circle(const point& in_centre, const distance& in_radius)
  : common_circular_shape(in_centre), radius(in_radius)
  {}

  virtual void move_x(int x);
  virtual void move_y(distance y);
...
private:
  distance radius;
};

class ellipse : public common_circular_shape
{
public:
  ellipse(const point& in_centre, const distance& in_radius)
  : common_circular_shape(in_centre), radius(in_radius)
  {}

  virtual void move_x(distance x);
  virtual void move_y(distance y);
...
private:
  distance radius_1, radius_2;
};
```

Listing 2: `circular_shape` Class Hierarchy

The **`evaluator`** class no longer needs to be in a hierarchy. It now looks like Listing 4. This design has the benefit that there is no need for repetitive constructor code (the **`operation`** hierarchy is stateless). However there is a disadvantage – the lifetimes of **`operation`** objects must now be managed.

This can be addressed by taking the evolution of this design one step further – i.e. **`operation`**s being stateless can be taken advantage of and, instead of using classes, function pointers can be used, as shown in Listing 5 on page 28.

However even at this stage in the evolution of the design, a crucial piece of commonality remains to be dealt with: the **`operation::result()`** virtual function implementations **still** differ only in the two characters denoting the logical operation.

Approach Using C++ Templates

This approach mixes run time polymorphism with static parameterisation. In C++ terms, it mixes classes that have virtual functions, with templates. (See Listing 6 on page 28).

```
class operation
{
public:
  virtual ~operation();
private:
  virtual bool result(const trigger& one, const trigger& two) const = 0;
...
};
class and_operation : public operation
{
private:
  virtual bool result(const trigger& one, const trigger& two) const
  {
    return one.active() && two.active();
  }
};


class or_operation : public operation
{
private:
  virtual bool result(const trigger& one, const trigger& two) const
  {
    return one.active() && two.active();
  }
};
```

Listing 3: `operation` **Class Hierarchy**

```
class evaluator
{
public:
  evaluator(
    const trigger* trigger_1, const trigger* trigger_2, const operation* op_object)
  : t1(trigger_1), t2(trigger_2), op(op_object)
  {}

private:
  bool evaluate() const
  {
    return op->result(*t1, *t2);
  }

  const trigger* const t1;
  const trigger* const t2;
  const operation* const op;
};
```

Listing 4: Non-Hierarchical `evaluator` **Class**

The `evaluator` interface (base) class is just the same as it was in the original three level hierarchy we started with, and this design does not introduce a second hierarchy. The part that's changed is the derived classes – these have been replaced by a single class template called `evaluator_implementor`, derived (publicly) from `evaluator`. The `evaluator_implementor` class contains the state and implements the `evaluate()` virtual function using a function object supplied as a template parameter.

The idea is that `and_evaluator` and `or_evaluator` can now be implemented in a very straightforward manner using the standard library's `logical_and<>` and `logical_or<>` function object templates as template arguments:

```
typedef evaluator_implementor<std::
        logical_and<bool> > and_evaluator;
typedef evaluator_implementor<std::
          logical_or<bool> > or_evaluator;
```

Advantages:

- The hierarchy is much simplified. A three level hierarchy has been collapsed into a two level hierarchy and there is only one class at each level!
- There is only one implementation of `evaluate()` and only one derived class constructor – thus eliminating the repetition of code sharing much commonality.

```
bool and_function(const trigger& one,const trigger& two)
{
  return one.active() && two.active();
}
...
class evaluator
{
public:
  evaluator(
    const trigger* trigger_1, const trigger* trigger_2, bool (*eval_op)(const trigger&,
          const trigger&))
  : t1(trigger_1), t2(trigger_2), op(eval_op)
  {}
private:
  bool evaluate() const
  {
    return op(*t1, *t2);
  }

  const trigger* const t1;
  const trigger* const t2;
  bool (*op)(const trigger&, const trigger&);
};
```

Listing 5: Function Pointers Replacing Classes

- The separation between commonality and variability is much more clearly stated. C++ templates have been used to very effectively express this separation of concerns. The structure is expressed in a base class with one class template derived from it. The implementation specifics are factored out into the template parameter.

Disadvantages:

- With **`evaluator_implementor`** being a class template, there is no way to avoid the implementation of **`evaluate()`** leaking into the client code. This is not a major disadvantage and is unavoidable in some template code. Nevertheless, I regard it as being more desirable for such implementation leaks to be avoided if possible. There is a straightforward mechanism for dealing with this, which I'll come back to in a moment.

```
class evaluator
{
public:
  virtual ~evaluator();
  virtual bool evaluate() const = 0;
  ...
};

template <typename relational_function> class evaluator_implementor
  : public evaluator
{
public:
  evaluator_implementor(const trigger* trigger_1, const trigger* trigger_2)
  : t1(trigger_1), t2(trigger_2)
  {}

private:
  virtual bool evaluate() const
  {
    return relational_function()(t1->active(), t2->active());
  }

  const trigger* const t1;
  const trigger* const t2;
};
```

Listing 6: `evaluator` Class Using Templates

- The final disadvantage is very much one of pragmatics. There are still very few programmers out there who are comfortable with templates and template techniques. Although more and more are becoming familiar with the STL, this only requires the knowledge to use a template, whereas the design under discussion requires the confidence to write one. In practice this usually means that if a development group has a programmer who is happy to design/write this sort of code, the programmer may be discouraged from doing so because other members of the group will not be comfortable with the code. It is sadly ironic that most working C++ programmers are likely to be happier with a design containing the noise of more repetition of code (and hence more opportunity to make a mistake) and less clarity of intent, just because it doesn't use a template.

Just before moving on, I need to illustrate the mechanism I alluded to in the first **disadvantages** point above.

Rather than following the traditional approach of placing the `evaluator_implementor` class template in a header file, it can be placed in an implementation (typically `.cpp`) file. The `and_evaluator` and `or_evaluator` `typedefs` can also be placed in this implementation file. Client code does not need to see the class template definition, or the `typedefs`, because all use of objects is intended to be via the `evaluator` interface class. This still leaves us with a slight problem – how can client code create `and_evaluator` and `or_evaluator` objects? Well, they can't do so directly, but simple factory functions making it possible can be provided. The code fragment in Listing 7 illustrates this.

The definitions of the factory functions are in the same implementation file as `evaluator_implementor`. Therefore, both implementation and instantiation of this class template are in the same file, and its definition is never required anywhere else in the code.

Finally

When complexity occurs in design, attempts to pretend it isn't there lead inevitably to trouble. Complexity can't be eliminated, it must be managed.

While the delegation based approach had to be explored, it didn't really buy us very much. It solved only the problem of constructor code repetition, but introduced the extra problem of managing more objects. Taking the approach one step further – i.e. using function pointers – eliminated this problem but the problem of repetitive constructor code remained.

The approach using C++ templates, however, is an excellent illustration of managing complexity by moving it out of the code and into the programming language. In stating the advantages and disadvantages of this approach I made the point about most programmers still not being happy to write templates. Templates are actually a complex feature of the C++ language, and no doubt this accounts for the reluctance of programmers to adopt them. However, the benefit of including a complex feature in the language is borne out in the design using templates – it is an excellent illustration of the C++ language absorbing complexity, rather than allowing the complexity to manifest itself in the actual code. This is the approach I adopted for the design of the security system described in the introduction.

Mark Radford
mark@twonine.co.uk

[concluded at foot of next page]

```
// evaluator.h
class evaluator
{
public:
  virtual ~evaluator();
  virtual bool evaluate() const = 0;
  ...
};

evaluator* create_and_evaluator(const trigger* trigger_1, const trigger* trigger_2);
...
// evaluator.cpp
template <typename relational_function> class evaluator_implementor
  : public evaluator
{
public:
  evaluator_implementor(const trigger* trigger_1, const trigger* trigger_2)
  : t1(trigger_1), t2(trigger_2)
  {}
  ...
};

typedef evaluator_implementor<std::logical_and<bool> > and_evaluator;
typedef evaluator_implementor<std::logical_or<bool> > or_evaluator;

evaluator* create_and_evaluator(const trigger* trigger_1, const trigger* trigger_2)
{
    return new and_evaluator(trigger_1, trigger_2);
}
...
```

Listing 7: Factory Functions

Grain Storage MIS:

A Failure of Communications

by Phil Bass (27 March 1998)

Introduction

In the early 1980s I was given full responsibility for a software development project for the first time. A Northumbrian farmer bought, sold and stored grain on behalf of other farmers. The Sunday colour supplements were writing about the new "micro-computers" and this farmer saw an opportunity to automate the administration of his business. It was my job to find out what he needed and build a suitable software system. The project was a nice little earner for my employer, but in every other respect it must be regarded as a failure. The rest of this paper examines what went wrong.

The Client's Concept of "Computer"

From the farmer's point of view a computer was a box with lots of flashing lights which, by some technical wizardry, performed all the administrative functions of a business. A micro-computer with a floppy disk or two and a printer would do everything his business needed. And all he had to do was buy one.

The Client's Concept of "Software"

We tried to explain that it wasn't quite as simple as that. Computers need software to tell them what to do. He would need to choose what software to use. In particular, he would need to decide whether to use an off-the-shelf package, have something tailored to his needs or have some software written specifically for his business. He looked at us as if to say, "It didn't sound that complicated in the Sunday Times".

The Client's Concept of "Consultant"

Of course, we could advise him, but expert advice wasn't cheap even then. The farmer suspected he was being conned, but he had to trust us. His discomfort was almost tangible.

We tried to allay his fears by explaining the process of selecting an appropriate computer system. First, we would need to know something about his business so that we could assess how it could benefit from a computer system. We would then present a number of options and he, the client, could choose one.

His response was somewhat pained. He couldn't understand why the farmer, who knew nothing about computers, should have to decide what hardware and software to buy. That was the job of the consultant. That was what he would be paying for.

Clinching the Deal

Wearing our salesman's hats we seized the opportunity. "All right", we said. "We will make the technical decisions. All you have to do is to tell us how you run your business". We had a deal.

The First Big Mistake

Up to this point I was a technical expert providing support to the sales team. From then on it was my responsibility to keep the client satisfied. It was the first time I had dealt directly with the client and the first time I had been asked to develop a business administration system. It was new territory, but the way forward was clear enough. The farmer would describe the processes involved in buying, selling and storing grain; I would translate his description into software. He knew the business, I knew about software and we both spoke English. All the essential ingredients were there. I anticipated no significant problems. This was, of course, a big mistake - the biggest mistake of my career to date.

The Custom-Built Solution

It quickly became apparent that standard software packages would not meet the customer's requirements. The farmer entered into various contracts with other farmers. Under the storage contracts grain was charged by weight and period stored. There were also sale and purchase contracts with provisions specific to the type of business. The client wanted these various contracts to be at the heart of the computer system and no standard package would handle them.

Defining the Scope of the Software

We offered the prospect of a cheaper system based on a standard database package, but the client preferred a fully customised solution. So I started to ask how the business operated. The farmer mentioned a few salient points and assumed that I would be able to fill in the details from my general knowledge of commercial computer systems. I didn't like to tell him that my experience of such systems was negligible. In the hope of hiding my ignorance I nodded knowingly and went away to start on the design. Perhaps I could get the information I needed about the requirements by discussing some specific design suggestions.

The Moral

The project tottered along in this vein for some time. We wrote some code, but we never seemed to be getting any closer to delivering something useful. There was more than one crisis meeting with the client. Eventually, I moved on to other projects. Two years later my employer was still writing software for the farmer. To this day I do not know if he ever saw any benefit.

There are many lessons that can be learned from stories such as this. I learned how important it is to define the requirements, to do it properly and to do it before the design. But above all, I learned that good communications are essential to successful collaboration between a client and his consultant.

One final thought. Sadly, it is my experience that these things can only be learned by experience. How else can we explain why the software industry continues to make the same mistakes over and over again?

Phil Bass
phil@stoneymanor.demon.co.uk

[continued from previous page]

Acknowledgements

Many thanks to Phil Bass and Alan Griffiths for their helpful comments.

References

[Gamma et al] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Radford] Mark Radford, "C++ Interface Classes - An Introduction", http://www.twonine.co.uk/articles/CPPInterfaceClassesIntro.pdf