contents

credits & contacts

Editor: John Merrells
merrells@acm.org
241 Heartwood Lane,
Mountain View,
CA 94041-11836,
U.S.A

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

Readers:
Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Mike Woolley
mike@bulsara.com

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
Thaddaeus.frogley@creaturelabs.com

Membership and subscription enquires:
David Hodge
membership@accu.org
31 Egerton Road
Bexhill-on-Sea, East Sussex
TN39 3HJ, UK

Advertising:
Peter Goodliffe
ads@accu.org
4 Malvern Road
Cherry Hinton
Cambridge CB1 9LD, UK
01223 518579

Editorial
Engineering Notebooks

It was years before I started keeping notes. Even in my college days I rarely took notes, relying on handouts and memory instead. In my first few professional jobs I just did my day-to-day thing. I had the one project, and I always knew what state it was in. I felt no need of notes. I was foot-loose, fancy free, and unencumbered.

I read an article in 1994 espousing the wonders of keeping a work jotter. I tried it for a week or two, but like diary keeping after new years day, it was a good idea that just wouldn't take hold.

Then I started working with a compulsive work diarist. He'd trained as a scientist, and consequently knew more about the ear canal of the common cricket then your average software engineer. Thankfully he also knew more about software engineering then your average software engineer too. He had an array of A4 books on his shelf, neatly labelled and dated. Every meeting, decision, conversation, thought, was written up, in the style of the scientific method, with a green tortoise shell fountain pen. Consequentially his knowledge of our product, its history, and who said what, where, and when was encyclopaedic.

Spurred by this inspiration I resolved to start an engineering notebook at the start of my very next major project. Two years later I bought myself the thickest A4 notebook available and set out to document my journey through a major piece of software development. I approvingly noticed that my VP of engineering carried a notebook to keep track of the many intertwined projects within our division. But, her notebook was A10. Size matters. My notebook was too big. It couldn't be with me always. My entries petered out.

Three years later, at the inception of my next project, I went for a midsize, thin, A5 college notebook. Success! I have now been a compulsive note taker for two years. And, I'm not going back to my unenlightened self.

I find it serves multiple purposes for me, over different time frames. In the short-term, day-to-day and week-to-week, I mostly maintain to-do lists, and project status information. Interspersed with this is some commentary, and description of problems that arise, the possible directions to take, and the currently preferred solution.

In the mid-term, month-to-month, I scan back through the pages seeking threads of thought that went un-concluded. Or, for a reminder of the problems that were hastily swept under the carpet in the interests of unhindered forward progress. [The code written in the rush of pub lunch confidence.]

In the long-term, year-to-year, I get some historical perspective over how the project progressed, both of the team and of myself. Writing my end of year appraisal is no longer the soul-searching agony of: 'what the hell have I been doing all year, all that time and only a few thousand lines of code to show for it.' I now have a day-by-day, blow-by-blow record of what I was doing.

Reading your own forgotten words a year later can be quite enlightening. From how incredibly insightful you can be about the final form of the project, to how incredibly naïve and deluded you were on how much effort it would actually take.

Professional advancement comes from considering feedback about your performance, from your management, from your peers, and from your own introspection. There's none as devout as the redeemed, so I fervently recommend the maintenance of an engineering notebook. Start yours today.

Apologies

Apologies to Björn Karlsson for two mistakes made in the development and presentation of his Boost article published in Overload 47. Firstly, a draft of the article was published in place of a more polished final revision, and more noticeably the header and byline of the article were omitted. Graciously Björn has accepted our apologies and has offered to write a follow up article covering one of the boost libraries in more detail. I hope that other boost library authors will follow suit in discussing their fine work within the covers of Overload.

John Merrells
merrells@acm.org

Notes

`http://www.meadweb.com` – My brand of notebook. I suggest green for work, and blue for your Overload articles ;-)

Copy Deadline

All articles intended for publication in *Overload 49* should be submitted to the editor by May 1st, and for *Overload 50* by July 1st.

Thinking about "reuse"

By Allan Kelly

Call me a heretic if you like, but it seems to me that "reuse" – that Holy Grail of software coders – seems to be a false idol. I'm not alone in this, Kent Beck and the Extreme Programming crowd have been denying reuse for several years in their efforts to "do the simplest thing that works" and Kevlin Henney [Henney2002] has questioned some assumptions too.

What is happening here? Well for me the serious doubts set in last year when reviewing Generative Programming by Czarnecki and Eisenecker [Czarnecki+2000]. For me, their cure was worse than the disease!

Further, times are changing and so are the economics of software. We are no longer confined to traditional models of reuse. The development of web services means there are new models appearing. Got a good library of chemical simulators? Why not SOAP enable it and charge per use from your web server? And why not experiment with new ways of funding development? Chris Rasch has suggested using the bond market to fund software development [Rasch].

How do we get here?

The appeal of reuse is obvious – especially if you have to pick up the bill for software development. If we could just reuse what we have already done…

This was one of the early promises of object-orientation. The idea was that objects represented some kind of entity, which would be applicable in more than one program. As Czarnecki and Eisenecker point out, the idea that reusable objects would simply drop out of a system development proved false.

It transpires that producing an object usable in more than one system is very difficult, in some cases more than double the effort. Czarnecki and Eisenecker introduce the idea of "designing for reuse" and "designing with reuse". (I agree with them on everything up to this point. Beyond this, my problem with *Generative Programming* is that I don't see how their solutions can be used in a real world environment. Much of the code they produce comes dangerously close to being unmaintainable.)

This division seems to recall ideas of the "problem domain" and "solution domain": in analysing the problem domain we identify and create objects which could be reused, when working in the solution domain we take these prefabricated modules to create our solution.

It appears that finding the right granularity for reuse is somewhat difficult: function level reuse gave way to object reuse and object reuse has given way to component reuse. (The word module is so malleable that I'm deliberately avoiding it here.)

All the time reuse seems almost within our grasp. The truth is, as an industry we reuse more code than ever before. We make

extensive use of code libraries, code-generating wizards (which are basically a form of meta programming), cut-and-pasted code, and packaged applications with macro languages or automation interfaces. So maybe we are not giving ourselves credit where credit is due.

So, why do I say reuse is a false idol?

There are two things wrong with reuse as we generally talk about it. First, I think we need to look at the costs of reuse. Second, reuse is a vague term; not only does it depend on who is using it, but it covers a multitude of ideas.

The cost of reuse

Sometimes costs are obvious: Joe will be tasked with creating a reusable library of widgets, so the cost is Joe's salary for the six months of development. More often the cost of reuse is hidden. As a developer you will decide to make your object slightly more generalised than it needs to be, or you will allow for some obvious enhancement which is not yet required. In effect you are doing more work than you need to, so the cost is perhaps 20% of your salary.

In either case you should have some idea of when the payback will come – a few weeks? months? Are you sure the payback will come at all? Even if it does come, who will see the benefit?

In addition to the financial cost there is the cost of work foregone. If a less generalised object had been produced in half the time you would be free to move onto the next work item. Economists call this "opportunity cost", it is the cost of not doing something else.

Building more generalised code, frequently (but not always) leads to more complex code. By its nature this is more prone to errors and more difficult to maintain.

Further, can we be sure that the additional as yet unused functionality is tested as well as the functionality that is to be used? If I create a Widget class with three methods, and then I decide to add an extra three methods (which I'm sure we are going to need in future) do I test the second three as completely as the first three?

There are other issues with this scenario:

- If we change the class before we get to use the three extra methods do we need to change them?
- If Fred takes over the class from me, won't he be wondering what is going on?
- If we need to port the class, or rewrite it, or refactor it, we have twice as many methods to work with than we actually need.

If these methods aren't used we have wasted our time writing them. Even if they are used, between creating them and using them we have to support them, they represent the "cost of carry" – to borrow another term from economists.

Finally, consider those three methods again. When they are finally used they aren't even being reused – they are being used for the first time! Sure, the Widget object is being reused but are we putting the cart before the horse here?

What does reuse actually mean?

The next problem with reuse is the very word itself. In part this varies according to your perspective. If I create my Widget class for program Foo, and I use it again in program Bar, then I have *reused* it because I have *used* it before. But if you were to write program Bar are you *reusing* Widget or just *using* it?

Adding "re" to the front of the word "use" doesn't actually buy us very much. When you got on the number 65 bus this morning, were you using the bus to get to work? Or were you reusing the bus to get to work?

OK, maybe I'm playing semantic games here but when we talk about code reuse we cover a lot of ground:

Reuse in the car industry

The car industry has many differences to the software industry. Most obviously in production costs; once your magnum opus software is complete production costs are minimal, just the cost of CD duplication or server bandwidth. In the motor industry completing the design is but the first stage: production costs dominate.

In recent years *just in time* assembly lines and *lean* practices have revolutionised car production, while increased competition between manufacturers has led to increased product diversification and a reduction in the lifetime of models. Consequently, competitive advantage rests with those manufacturers that can produce a diverse range of frequently updated models. This places the onus back on design.

In an effort to reduce design costs manufacturers have turned to reuse: within seven years Volkswagen moved from making 30 different models on 16 floorpans to 54 models on 4 floorpans [*The Economist*, 27th April 2000], look how the Passat is an Audi A4, is a Skoda Octavia.

With this background Cusumano and Nobeoka [*Thinking Beyond Lean*, Freepress, 1998] have looked at the design process, and in particular how design reuse is enabling car companies to meet this challenge. Although *Thinking Beyond Lean* was published in 1998 its lessons are not dated and provide valuable insights into reusing design and transferring technology between projects. Almost as vindication, since 1998 we have seen platform sharing as a major factor behind the Renault-Nissan, Mercedes-Chrysler-Mitsubishi and Ford-Mazda-Volvo link ups.

Cusumano and Nobeoka identify four development models, three of which involve transferring technology:

- New design : "projects that primarily develop platforms from scratch"
- Concurrent technology transfer: "a new project begins to borrow a platform from a base preceding project before the base project has completed the design work. Generally, this transfer occurs within two years"
- Sequential technology transfer : "a project inherits a platform from a base project that has finished the design work"
- Design modification : "a project that replaces an existing product without creating a new platform or borrowing a platform from another production line"

I suggest that these models hold true for software development. A car *platform* differs from a software *framework* in that a platform is built for a specific product but may be reused with modification; while a software framework is designed from the start as a generic environment. Platforms provide greater focus on the initial project.

Cusumano and Nobeoka concentrate on Toyota, and their use of development centres and concurrent technology transfer of platforms between Toyota and Lexus lines. Honda uses more

- using a third party library, maybe even the STL
- using code salvaged from the system we are replacing
- using code from a different project within the company
- developing two programs in parallel and sharing some code

This list could go on, but I hope you get my point. What is *use*? And what is *reuse*? The term reuse is not just overloaded but is actually pretty devoid of any real meaning!

So where does this leave me?

So far I've pointed out that reuse is a good thing. I've also claimed that much of what we do in the name of reuse is wasteful. And finally, I've attacked the word itself. So, where am I going with this?

Reuse has become too generalised, too much of a good thing. I think we need some new words, some new terms, new ideas to describe what we are actually talking about here.

OK, so we will end up sounding like management consultants, but I don't think one word can really cover all the things we use it for. (And maybe, just maybe, we will see our salaries get a bit closer to those of management consultants!)

What else can I say?

Now what I'm about to suggest may come across as jargon-making, for which I apologise. However, jargon can serve a useful purpose: to specifically identify an idea or concept. So here goes, here are some terms and ideas.

Commodity

This is a word I like. I was already playing around with it in this context before I saw Kevlin Henney's use of the word in Overload 47. If we are developing an object, a module, a component, or whatever, which we are intending to use in several places, *commodity* fits: we are developing a piece of software which can be used in various ways.

Commodity also has a nice overtone of trading; we buy and sell commodities. This highlights the commercial nature of most software development.

Another benefit of *commodity* is that it implies certain properties, interfaces and standards – some commonality. A television is a commodity, we can be sure that in the USA it will use one standard, and in England another – inconvenient yes, but a known issue. We can develop DVD players to work with this interface for the TV, we don't need specialised sets.

Product has similar commercial overtones to commodity but must be rejected for our purposes. The word carries too much other baggage. It implies a complete, or completed, item. Commodity is much more granular.

Technology Transfer

Although this term smells a lot like management-consultant-speak, I think it is much more accurate than reuse. It clearly states what we are talking about and is free of the baggage that reuse carries.

As the sidebar outlines, the term is already used elsewhere and it can be expanded to describe the temporal characteristics of the transfer.

There are two distinct forms of software transfer that can be identified:

- **Horizontal transfer**: this is the sharing of common commodities between diverse projects, e.g. when we use a strings library or a threading library we are transferring technology horizontally. The projects may have little business commonality but they have a great deal of raw technology in common.
- **Vertical transfer**: this is the transfer of technology between similar products, typically within a family of software. For example, a bank that uses the same evaluation model for equities options, foreign exchange trading and risk management is transferring technology vertically.

traditional matrix management practices, but has great success sharing common platforms between more diverse products – witness the Civic platform, which is also used for the CRX sports car and CRV SUV while internationally the same platform was used by Rover for several years. Elsewhere Chrysler's continued use of sequential transfer and design modification in the 1990s reduced short-run costs, but led to an outdated product line that had problems competing in the market.

From analysis to coding, software development is an exercise in design, so comparisons with car design are highly relevant. Perhaps the biggest lesson is: reuse does not just happen, you have to manage it, you have to put processes in place to deliver reuse. (Which brings us back to Conway's law – "align process and design.")

My experience is that most code-level reuse is bottom-up, it comes from engineers designing a part to be reusable. Cusumano and Nobeoka show how management can be aligned to this process and how they can encourage it. If you want reusable software you must align your process, engineers must know how to reuse, management must be aligned.

Listing the lessons of Cusumano and Nobeoka would take an article in their own right, better you go and read the book. However, here are a few ideas:

- Look to design a product with a platform; plan for the platform to be re-used, follow-on projects should overlap with the original project
- Organise around product lines not functional abilities
- Use strong, autonomous, project managers to drive projects forward
- Physically co-locate project teams together
- Integrate R&D with product development
- Keep communication complexity to a manageable level
- The "company memory" approach does not work well, leading to reuse of old components and infrequent replacement cycles
- Informal "tech clubs" can be used to promote reuse without formal multi-project management.
- Written documentation is good for transferring component knowledge but poor at transferring integrative knowledge; perhaps worryingly, evidence shows that placing a lot of emphasis on retaining prior knowledge can hamper innovation and use of new technologies

Finally, sometimes we need to forget about reuse, sometimes reinventing the wheel brings benefits. Mazda's Miata (MX-5) was developed by a "guerrilla team", physically separated from the main development groups, freed from reuse restraints, with a mandate to produce a car to enhance Mazda's image. Reuse is a valuable tool, but it is not an end in its own right.

Few, if any, software projects are on the scale of designing a car, still, there are useful lessons and ideas here. The software industry is not an island, we must look to established business, management and engineering models to understand and improve ourselves.

Vertical transfer usually relates to the application domain so it is more common within organisations, while horizontal transfer, which frequently relates to the system domain, is more common between organisations. Program families, as discussed by Parnas [Parnas1976], are akin to vertical technology transfer.

Most of the well-known libraries available to developers (e.g. Boost, Rogue Wave Source Pro, Microsoft MFC, etc.) are horizontal libraries. They deal well-known themes (threads, database connectivity, GUIs, etc.) which developers are actually rather good at solving because we deal with the same problems all the time.

A vertical library inhabits a niche that is less well known. For example NAG's libraries are less well known than Rogue Wave because fewer people need advanced mathematical algorithms. Even so, more developers know about NAG's libraries than know about, say, libraries for computing fuel consumption in cars, because many developers have a mathematical background.

Integration

Integration is the process of embedding the transferred technology into another development. We may want to distinguish between a donor project (where the technology comes from) and a recipient project (where the technology is used.)

Intellectual Property – IP

IP is the stuff that technology businesses are based on. Increasingly, this is becoming a commodity as patent databases are set up and companies are encouraged to place their property on line for others to buy. In the hardware world this is established practice – think of the way ARM has built an entire business on the licensing of its CPU technology.

Software can travel the same road. Our intellectual property is expressed in terms of source code, UML diagrams and documentation – although a significant part is still locked up inside developers' heads.

When we use a third party library we are not just avoiding the need to write some code ourselves – we are actively seeking to use others' work. For example, rather than try and understand the mathematics behind optimisation theory I just go and buy the work NAG have done; I use their intellectual property to save myself time in development, testing and verification.

A word of warning though: some people have taken a dislike to the idea of IP expressed in software. The problem is not with the idea of IP itself, but with some of the patents granted, and enforced, which are giving it a bad name.

Publishing

Once we start to think of source code as a commodity with the ideas and work contained therein as intellectual property, it becomes natural to think of publishing the work. Once our IP is expressed in a commodity, which may be bought and sold, we need to publish it.

Publishing may be external, we may place an advert in Dr Dobbs and request payments, or it may be internal, we may announce to our company that we now have a library that performs some business function, or may be just publishing at the level of our development group.

Commissioning

To continue the publishing metaphor we may like to think of the initial request as a commission. One could imagine a large company, say a bank, identifying vertical application families with common features, e.g. credit appraisal, and commissioning one development group to develop a commodity solution which other groups could also use.

Packaging

Once we have our intellectual property, we must ensure it is packaged for delivery as a commodity for publication. We may choose to package it as source code, a shared library, or a service accessible over the internet. How we package it will in part be determined by our revenue model, and by how we wish to divide the market.

For example, look at the way Cola-Cola subdivides its market. Although it is nominally selling the same product (sugared water with flavouring) the pricing depends on the packaging. A 330ml can (sold to people on the move) costs more litre-for-litre than the 2 litre bottle (sold in the supermarket to families at home) which in turn costs much more, litre-for-litre, than the cola concentrate sold to restaurants for sale with food – and where it is mixed with water and marked up.

We may make our IP available as a SOAP enabled web service and charge per use. We may repackage this as a DLL and sell it for a fixed price to a large corporation. Finally, we may enter into revenue sharing deals with other producers who compile our commodity into their own.

Conclusion

That several people in the software industry are questioning "The Holy Grail of Reuse" is probably a sign of a maturing industry. Reuse is a worthy goal, and has its uses, but it should not be all consuming. We must understand its costs and implications.

Reuse will probably remain a general all embracing term, it is too entrenched to go away, but we may be better served by coming up with new labels for some of the ideas it covers. I'm not saying my labels are the right labels or that they will stick but I'm starting the debate.

Nor am I claiming that we can simply replace the word reuse. We may try to do a mental "global replace 'reuse' with 'technology transfer'" but this wouldn't change the real situation. We need to change the way we think about our code and the way we use it.

Even without our own debate on what constitutes software reuse the economics of software are changing. For many applications we can now assume an Internet link which enables us to think about pay-as-you-go and leasing models.

Allan Kelly
allan@allankelly.net

References

[Czarnecki+2000] Czarnecki & Eisenecker, *Generative Programming*, Addison-Wesley, 2000

[Parnas1976] David Parnas, "On the development and design of Program Families", 1976, reprinted in *Software Fundamentals: collected papers by David L. Parnas*, edited by Hoffman & Weiss, Addison-Wesley, 2001

[Henney2002] Kevlin Henney, "minimalism : the imperial clothing crisis", *Overload 47*, February 2002

[Rasch] Chris Rasch, "The Wall Street Performer Protocol : using software completion bonds to fund open source development" http://www.firstmonday.dk/issues/issue6_6/rasch/index.html

The C++ Template Argument Deduction

By Andrei Iltchenko

Introduction

With the increasing popularity of the Standard C++ library and the STL, it is becoming more and more important for a C++ programmer to understand the underlying mechanics of generic programming in C++. In this article I will look into the little known details of template argument deduction, an area of the C++ language that a lot of programmers find difficult to comprehend and a knowledge of which is vital for using Standard C++ efficiently.

This article is intended for experienced C++ programmers who wish to know the exact and yet clearly explained details of how argument deduction is done in the C++ language. Deduction of template arguments is an incredibly vast area and covering it in detail and in all its entirety in one article would be impossible, so I chose to pick out the most frequently used contexts that call for template argument deduction and explain them in detail rather than gloss over all cases of use of argument deduction. The following uses of argument deduction are covered in this article: "Deducing template arguments from a function call expression", "Deducing template arguments when taking the address of a (member) function template specialization", "Deducing template arguments for explicit specialization of a (member) function template".

Template argument deduction only pertains to function templates. This means that whenever you reference a specialization of a class template, you need to explicitly specify all template arguments. I will therefore refer to function templates only in the remainder of the article.

Although there are different cases when the language calls on template argument deduction, they are all based on deducing template arguments from a type. So it is here that the discussion will start.

Deducing template arguments from a type

Template parameters come in three flavors: 1) template template parameters (associated with a class template, which I will denote as `TT` for the remainder of the article), 2) type template parameters (associated with a type, denoted as `T`), and 3) non-type template parameters (most often associated with a constant value of integral or enumeration type, denoted as `i`).

To deduce a template argument from a given type means that you have two types – a *type-id* that contains one or more template parameters, I will call such a *type-id* `P` for the rest of the article (some combinations of `TT`s, `T`s, and `i`s within one `P` are possible). For the moment I will omit the details of how `P` is obtained, suffice to say that in most cases it is one of the function parameters of a function template declaration. The other type that you have is a *type-id* that comprises no template parameters, which I will call `A`, and an attempt is made to find a class template for a `TT`, a type for `T`, and a value for `i` so that `P` becomes identical to `A`.

Example:

```
template<template<class> class TT, class T>
T* alloc(TT<T>& a) {
  return a.allocate(16);
}
```

For this template `P` can be `TT<T>`. If the *type-id* `A` from which we want to deduce `TT` and `T` is `std::allocator<int>`, then argument deduction deduces `TT` to be `std::allocator` and `T` to be `int`. This results in `P` becoming `std::allocator<int>`, i.e. identical to `A`.

So far so good, but what kind of type can `A` be and what combinations of `TT`s, `T`s, and `i`s are allowed in `P` so that template argument deduction is possible on the template parameters that `P` consists of? The first part of the question is easier to answer since for template argument deduction the language allows `A` to be any valid C++ type (not a constant reference to constant function, of course). As for `P`, only the following combinations are allowed:

P's most specific type kind	Allowable form for P
Object type	`cv-seq`$_{opt}$ `T`
Pointer type	`T*`
Reference type	`T&`
Array type	`T[integral-constant-expression]`, `type[i]` , `T[i]`
Function type	`type(T)`, `T()`, `T(T)`
Pointer to data member type	`T type::*`, `type T::*`, `T T::*`
Pointer to member function type	`T(type::*)()cv-seq`$_{opt}$, `T(type::*)(T)cv-seq`$_{opt}$, `T(T::*)()cv-seq`$_{opt}$, `T(T::*)(T)cv-seq`$_{opt}$, `type(type::*)(T)cv-seq`$_{opt}$, `type(T::*)()cv-seq`$_{opt}$, `type(T::*)(T)cv-seq`$_{opt}$
Class template specialization (class type)	`class-template-name<T>`, `class-template-name<i>`, `TT<T>`, `TT<i>`, `TT<>`

Where `cv-seq` designates any valid sequence of `const` and `volatile` qualifiers, `opt` stands for optional, `(T)` designates a list of function parameters where at least one parameter is `T`, `<T>` designates a list of template arguments where at least one template argument is `T`, and, finally, `<i>` designates a list of template arguments where at least one template argument is `i`. `type` refers to any C++ type which consists of no template parameters, i.e. it does not depend on the template parameters of the function template. The construct `class-template-name` designates a name of a class template that is not a template template parameter. For instance this is shown in the following code snippet, where `P` can be `std::allocator<T>` and `class-template-name` is thus `std::allocator`.

```
template<class T>
T* alloc(std::allocator<T>& a);
```

A few explanations are necessary to make sense of the above. When there is more than one `T` in some of the combinations shown, the `T`s do not need to be the same type template

parameter. A good example that demonstrates this is the standard object generator for pointers to member functions – the function template `mem_fun_ref`. Here is one of its declarations (the name `mem_fun_ref` is overloaded):

```
template<class R, class T, class A>
mem_fun1_ref_t<R,T,A> mem_fun_ref(R(T::*)(A));
```

Here `P` can be `R(T::*)(A)`, which is a variant of `T(T::*)(T)`.

The second point that must be made about the allowable combinations is that they can be applied recursively – any of the 24 forms shown can be used in place of `T` in forming other permissible combinations, provided, of course, that the resulting form is a legal C++ *type-id*. For instance `P` having a form of `type(T::*)(TT<T>,char)const` is composed of `type(T::*)(T)const` and `TT<T>` and is accepted by template argument deduction (remember `(T)` designates a list of function parameters where at least one parameter is `T`, so `(TT<T>,char)` is a variation on `(T)`). The following function template declaration clarifies this example further:

```
template<template<class> class Alloc,
                          class C, class T>
void dummy_alloc(
              void(C::*)(Alloc<T>,char)const);
```

Here one possible form of `P` is `void(C::*)(Alloc<T>)const`. If the corresponding `A` designates the type `void(Myclass::*)(std::allocator<int>)const`, then template argument deduction will deduce `Alloc` to be `std::allocator`, `C` to be `Myclass`, and `T` to be `int`.

The forms involving template template parameters `TT<T>`, `TT<i>`, `TT<>` require special elucidation. Each of these forms specifies the use of a class template `TT`, i.e. a *class template specialization*, meaning that whenever you use any of the above three combinations in `P`, you need to supply the same number of template arguments as the declaration of `TT` specifies, unless the declaration of `TT` contains one or more *default template arguments* in which case the trailing template arguments can be omitted. Note that the angle brackets must be present even if all `TT`'s template parameters have defaults. The following example demonstrates the use of a template template parameter in `P` where `TT` has default template arguments for all its parameters.

```
template<class T, template<class=T> class TT> >
void foo(int(*arr)(TT<>));
```

In this example `P` can be `int(*arr)(TT<T>)` and given `A` of `int(*)(std::allocator<int>)` template argument deduction will resolve `T` to `int` and `TT` to `std::allocator`.

Although there is a form `TT<>` among the allowable choices for `P`, it shouldn't be taken as a template template parameter with no template arguments. In fact it just means that the list of arguments doesn't contain any other template parameters and must contain non-dependent types, which is illustrated by the following example:

```
template<template<class, class> class Sequence>
void empty_sequence(Sequence<int,
              std::allocator<int> >& seq) {
   seq.empty();
}
```

When deducing template arguments for the above function template one possible `P` is `Sequence<int, std::allocator<int> >`, which corresponds to `TT<>` since none of the arguments given to the specialization of `Sequence` are dependent on the template parameters of the function template `empty_sequence`. If template argument deduction is supplied a corresponding `A` of the form `std::vector<int, std::allocator<int> >`, it will then deduce `Sequence` to be `std::vector`.

While on the subject of using the forms `TT<T>`, `TT<i>`, `TT<>` in `P`, it is vital to point out that due to the language permitting to use any of the 24 allowable forms recursively it is possible that a type template parameter `T` is mentioned more than one time in `P`. This is allowed and does not in any way affect the deducibility of `T`. This is demonstrated by a slightly modified version of the previous example:

```
template<template<class, class> class Sequence,
         template<class> class Alloc, class T>
void empty_sequence(Sequence<T,
                     Alloc<T> >& seq) {
   seq.empty();
}
```

In this function template, `P` can be `Sequence<T, Alloc<T> >`, which is a variant of `TT<T1,T2>` (it is allowed to treat the form `TT<T>` as `TT<T1,T2>` where `T1` and `T2` are type template parameters with `T2` being `TT<T>`, see the notes to the table of allowable forms) where, as you can see, the type parameter `T1` is `T` and `T2` is `Alloc<T>`. Given a corresponding `A` of `std::vector<int, std::allocator<int> >`, template argument deduction will be able to deduce `Sequence` to be `std::vector`, `Alloc` to be `std::allocator`, and `T` to be `int` in spite of the `T` appearing twice within the `P`. In fact it will deduce `T` twice and, with above `A`, each time to the same type.

But what happens if `P` references a type template parameter `T` more than once and the corresponding `A` supplies two different types for each occurrence of `T` in `P`? For instance, in the previous example if `A` had a form of `std::vector<int, std::allocator<char> >`, then template argument deduction would deduce `T` to have type `int` in one place and `char` in the other. The language prohibits these scenarios and deduction fails for `P`'s function template when the context supplies such an `A`.

A careful look at the table of allowable forms reveals that it's not possible to form a *qualified* `P` by using them (even if the use is recursive). This means that if you make `P` qualified, template argument deduction is not possible for it. For example:

```
template<template<class, class> class Sequence,
   template<class> class Alloc, class T>
void empty_sequence(T::Sequence<T, Alloc<T> >&);
```

In this code sample one possible `P` is `T::Sequence<T, Alloc<T> >` and because of this `P` being qualified, template argument deduction is not possible regardless of the form that the corresponding `A` has.

As I said before, in forming `P` a standard conforming compiler is allowed to replace `T` in any of the allowable forms with another form from the table, but no such provision is made for `i`. There are only four forms that contain `i` in the table, these are `type[i]`, `T[i]`, `class-template-name<i>`, and `TT<i>`. And in all these forms `i` is used as an *expression* that has the form of a simple identifier. Thus, by strictly following the rules it is not possible to get such `P` that would contain `i` in a form different from a simple identifier. It follows that if you have a `P` wherein `i` is used in an expression which doesn't have the form of a simple identifier, template argument deduction is not possible.

```
template<unsigned i>
void foo(int(&p)[+i]);
```

In this example **P** can be **int[+i]**. And no matter what form the corresponding **A** has, no argument deduction is possible for this **P**.

In a similar manner, no recursive use of the allowable forms enables to get **P** of, for example, this form: **type(*)[sizeof(T)]**. This is illustrated in the following code snippet:

```
template<class T, template<class> class TT>
void foo(int(*arr)[sizeof(TT<T>)]);
```

Which again means that should argument deduction be presented with such **P**, it will not deduce any template parameters in it.

In general, a **P** which is qualified or contains **i** involved in an expression other than a simple identifier (that is, not used by itself) or contains any other template parameter used in an expression such as **sizeof** is conventionally called a *nondeduced context*. Having such a **P** in a template declaration is not ill-formed, it's just that template argument deduction cannot deduce any template parameters in that **P** regardless of the form of the corresponding **A**. This effectively means that the template parameters that such **P** comprises must be either explicitly specified or deduced elsewhere. I'd like to note here that it is not possible to get a nondeduced context by recursively applying only the allowable forms. When consulting the table of allowable forms you should regard a nondeduced conext as a non-dependent type, which is denoted there as **type**.

Now that I've explained the concept of deducing template arguments from a type, it is high time that I embark upon some of the contexts that call on template argument deduction.

Deducing template arguments when taking the address of a (member) function template specialization

It is not uncommon for C++ programmers to initialize an object of type pointer to (member) function with an expression referring to a (member) function. As is the case with non-template (member) functions, (member) function template specializations can be used in such contexts too, much like normal non-template functions:

```
ptrdiff_t count_chr(const char* str,
                    const char ch) {
  // using the specialization of the standard
  // 'count' algorithm
  ptrdiff_t (* pf)(const char*, const char*,
     const char&)
     = std::count<const char*, char>;
  return (*pf)(str, str+std::strlen(str), ch);
}
```

One thing to remember here is that there are no implicit conversions etween pointers to functions of different types and pointers to member functions of the same class of different types. There must be an exact match:

```
struct test {
  void update1(char*, char*);
  void update2(const char*, const char*);
  void update3(const char*, const char*,
                                    int=0);
};
void  testing() {
  // Error, no exact match
  void (test::* pmf1)(const char*,
          const char*) = &test::update1;
  // OK
  void (test::* pmf2)(const char*,
          const char*) = &test::update2;
  // Error, default arguments are not
  // part of function type
  void (test::* pmf3)(const char*,
          const char*) = &test::update3;
}
```

Before plunging right into the nitty-gritty details of argument deduction, it would be helpful if I clarified the meaning of the term *function template specialization*, which has already been used a number of times in this article (member function template specializations follow the same conventions and will be omitted from the following discussion).

A function template specialization is a use of a function template name with a full set of template arguments that uniquely identifies exactly one specialization. For instance in the code example above **std::count<const char*,char>** is a specialization of the function template **count**. Sometimes a set of template arguments is given explicitly in a specialization and the number of arguments in the set matches the number of template parameters of the corresponding template, sometimes the set contains fewer template arguments than the number of template parameters of the matching function template, and sometimes it contains no template arguments at all (even the angle brackets could be left out). Whenever some or all template arguments are omitted, they must be deducible from the context and the result of the deduction is a function template specialization. When a full set of template arguments is specified, no template argument deduction takes place.

The point is that a C++ programmer never deals with function templates themselves (except, for example, when they declare, define or explicitly specialize them). Most of the time it is a specialization of a given template that a programmer uses. Below an earlier code fragment has been modified to clarify the points that were made earlier:

```
ptrdiff_t count_chr(const char* str,
                    const char ch) {
  // Using the specialization of the standard
  //    'count' algorithm without specifying
  //    template arguments explicitly.
  // 1. Name lookup finds the function template
  //  template<class InIter, class T>
  //  typename
  //    iterator_traits<InIter>::difference_type
  //      count(InIter first, InIter last,
  //                          const T& val);
  // 2. Template argument deduction deduces the
  //  template arguments to have types 'const
  //  char*' and 'char' respectively. Thus
  //  making the expression 'std::count'
  //  equivalent to 'std::count<const char*,
  //  char>'
  //
   ptrdiff_t (* pf)(const char*,
       const char*, const char&) = std::count;
   return (*pf)(str, str+std::strlen(str), ch);
}
```

Returning to template argument deduction when taking the address of a (member) function template, what I need to explain is how **P**'s and **A**'s are formed when some template arguments are not specified. The answer is surprisingly simple – there is just one **P** and one **A**. **P** is the type of the initializer expression (possibly converted to a type of pointer to function) and **A** is the type of the object being initialized.

For instance in the code sample above:

P is `typename iterator_traits<InIter>::difference_type(*)(InIter, InIter, const T&)`, which in terms of the conventions used in the table of allowable forms is a variant of `type(*)(T)`.

A is `ptrdiff_t(*)(const char*, const char*, const char&)` and template argument deduction will be seeking such types for `InIter` and `T` that will make **P** identical to **A**.

If the initializer expression was `std::count<const char*>`, only one template argument would have to be deduced (since one is specified explicitly), i.e. in that case **P** would be:

```
typename iterator_traits<const char*>::
                          difference_type(*)(
   const char*, const char*, const T&)
```

It is easy enough to write the initializer in such a way that no matching specialization could be found. For example the initializer of the form `std::count<char*>` will result in **P** being:

```
typename iterator_traits<char*>::
   difference_type(*)(char*, char*, const T&),
```

which doesn't enable template argument deduction to make such **P** identical to the corresponding **A**.

If the name of the function template whose address is being taken is overloaded, then there are as many **P**s as there are overloaded function templates by that name and template argument deduction will be performed against all of them with the same **A**. Those templates for which deduction does not succeed will not be considered further. If the initializer expression does not contain the angle brackets, non-template functions will also be considered, i.e. if the namespace `std` contained a function declaration:

```
ptrdiff_t count(const char*,
                const char*, const char&);
```

this function would also be considered in the example above. It is then up to overload resolution to decide which function template specialization is the most specialized and choose it. Note that a non-template function is always preferred to any function template specialization, as a non-template function is considered more specialized than a corresponding function template specialization.

It is quite common for **P** to contain multiple references to the same template parameter and my example with the standard `count` algorithm showed it. The first reference was in the return type `iterator_traits<InIter>::difference_type`, which is a nondeduced context and is therefore ignored, meaning that `InIter` must be deducible from its other occurrences in the **P**. The second and the third appearances are the same and have the form `InIter`. As the corresponding parts of **A** also constitute the same type `const char*`, `InIter` can be deduced from either part to be the same type. If they were different, for instance, if **A** was `ptrdiff_t(*)(const char*, char*, const char&)`, then `InIter` would be deduced to be `const char*` in one part and to `char*` in the other. Whenever such a situation arises, argument deduction fails. See the discussion on this issue in the previous chapter.

Deducing template arguments for explicit specialization of a (member) function template

Consider the following piece of code in which two overloaded declarations of the function template `foo` are defined:

```
template<class T>
struct example {
   template<class U>   // 1st member template
   int foo(U);
   template<class U>   // 2nd member template
   int foo(U*);
};
```

Now suppose I have an explicit specialization of the following form in the same translation unit:

```
template<> template<>
int example<char>::foo(int*);
```

But which member function template is explicitly specialized here? One possible answer is that it is the second one, but is it really? The truth is that for explicit specializations of (member) function templates, template argument deduction is done in a way similar to that I covered in the previous chapter – **A** is the function type of an explicit specialization named **N** and **P**'s are the function types of templates with name **N** that are members of the same scope as the explicit specialization. Template argument deduction then makes an attempt to make each of the **P**'s identical to **A** (for more on that see the chapter "Deducing template arguments from a type" at the beginning of this article). Those templates for which argument deduction fails are not considered further. Those for which it does are submitted to overload resolution and the latter makes a decision as to which of the templates is more specialized than the others. When it can make this decision, the explicit specialization is considered to specialize the most specialized template, otherwise the explicit specialization declaration is ill-formed.

In the example shown **A** is `int example<char>::foo(int*)`. Given that the explicit specialization is a member of the class scope `example<char>`, there are two **P**'s: $\mathbf{P}_1$ is `int example<char>::foo(U)`, and $\mathbf{P}_2$ is `int example<char>::foo(U*)`. For $\mathbf{P}_1$ argument deduction succeeds with `U` deduced to be `int*`. For $\mathbf{P}_2$ it succeeds too resulting in `U` having type `int`. So both member function templates are submitted to overload resolution for finding out which of them is more specialized than the other, and the latter selects the second, meaning that the explicit specialization specializes the second member template.

When declaring an explicit specialization, it is possible to explicitly specify some or all the template arguments of the function template being specialized. The arguments will actually be applied to the templates of the same name and scope as the explicit specialization. This can come handy in influencing the result of argument deduction. For example if the same translation unit contains another explicit specialization of the form:

```
template<> template<>
int example<char>::foo<int*>(int*);
```

Things will proceed as follows:
A is `int example<char>::foo(int*)`,
$\mathbf{P}_1$ is `int example<char>::foo(int*)`,
and $\mathbf{P}_2$ is `int example<char>::foo(int**)`.

No argument deduction will take place at all since the two member templates have just one template parameter and it has been specified. Since **P_2** now has a different type from the type of the explicit specialization it is no longer considered, thus the explicit specialization specializes the first member template.

Deducing template arguments from a function call expression

When name lookup finds a name in a function call expression to denote a function template or a (non-special) member function template and the call leaves one or more trailing template arguments unspecified or has a form without the angle brackets, template argument deduction comes into action:

```
int main() {
  int data[] = {3, 0, -1, -3, 16, };

  // Name lookup finds the function template
  // template<class RandomAccessIterator>
  // void sort(RandomAccessIterator first,
  //           RandomAccessIterator last);
  //
  // Template argument deduction deduces the
  // template argument to have type 'int*'.
  // This results in a call to the function
  // template specialization
  // 'void sort<int*>(int* first, int* last)'.

  std::sort(data,
            data + sizeof data/sizeof*data);

  std::vector<int, std::allocator<int> >
       container, empty;

  // Name lookup finds the member function
  // template
  // template<class InputIterator>
  // void vector<int,allocator<int> >::assign(
  //   InputIterator first, InputIterator last);
  //
  // Template argument deduction deduces the
  // template argument to have type 'int*'.
  // This results in a call to the function
  // template specialization
  // 'void vector<int,allocator<int> >::
  //               assign<int*>(int*, int*)'.

  container.assign(data,
             data + sizeof data/sizeof*data);

  // Argument dependent name lookup finds the
  // namespace scope function template
  // template<class T, class Alloc>
  // bool operator!=(const vector<T,Alloc>&,
  //                 const vector<T,Alloc>&);
  //
  // Template argument deduction deduces T to
  // be 'int' and Alloc to be std::allocator.

  if(container != empty)  container.clear();
}
```

Unlike the contexts requiring template argument deduction that I looked at before, which were all more alike than different in terms of how **P**'s and **A**'s were formed, this case is really distinctive and you will see why shortly. It is also the most frequently used form of template argument deduction, so I will back its description up with more case studies.

The distinction from what I showed earlier is that when deduction starts for a (member) function template: 1) there can be more than one **P**, 2) the number of **A**'s is always the same as the number of **P**'s. In fact, any function parameter which depends on a template parameter that has not been explicitly specified (see the earlier discussion on function templates and function template specializations) constitutes a **P**. If all template arguments are explicitly specified, there are no **P**'s and no argument deduction takes place. If the i-th function parameter of a function template qualifies as **P** (I will call it **P_i** for the rest of the article), then the i-th argument of the corresponding function call expression is **A** (which I will call **A_i**).

Different to the cases I covered before, here every **P_i** and **A_i** can undergo a transformation before template argument deduction takes place. Below are the details of how this process goes:

First every **A_i** is studied and:

- if **A_i** is a reference type, it is converted to the underlying type of the reference and is considered an lvalue*;
- if **A_i** is an lvalue of array type and **A_i** is not a reference type, the array-to-pointer conversion is applied to **A_i**;
- if **A_i** is an lvalue of function type and **A_i** is not a reference type, the function-to-pointer conversion is applied to **A_i**;
- if **A_i** is not a reference type, top-level const and volatile qualifiers are removed from **A_i**.

After that is done, every **P_i** is examined and:

- top-level const and volatile qualifiers (if any) are removed from **P_i**;
- if **P_i** is a reference type, the underlying type of **P_i** is substituted for it.

For the rest of this section, whenever I mention **P_i** or **A_i**, I'll be referring to their transformed versions.

With the information from the previous paragraphs in mind and the example I presented at the beginning of this section, I can now show how **P**'s and **A**'s are formed for the expression **`container != empty`**, for which name lookup finds the function template **`operator!=(const vector<T,Alloc>&, const vector<T,Alloc>&)`** as a possible candidate. This template has two function parameters, which depend on the template parameters. Both the function parameters are of reference type and are the same and hence the underlying type of the references is used in determining **P**'s – **P_1**, **P_2** are **`const vector<T, Alloc>`**. The corresponding **A**'s are also the same – **A_1**, **A_2** are

* In fact, this is an important trait of C++ that has a broader scope than deduction of template arguments from a function call expression. The general rule is that before being semantically analyzed every C++ expression that initially has a reference type is converted to the underlying type of the reference and is considered an lvalue. For example, given a conforming compiler, the following piece of code will never trigger the assertion:

```
int  main() {
  int i,  & ri = i;
  assert(typeid(i) == typeid(ri));
}
```

`std::vector<int, std::allocator<int> >`. Template argument deduction then deduces `T` to be `int` and `Alloc` to be `std::allocator`, which results in a call to the function template specialization

```
bool  operator!=<int,std::allocator>(
            const vector<int,std::allocator>&,
            const vector<int,std::allocator>&)
```

The fact that argument deduction makes each P_i identical to the corresponding A_i has a major implication on calling (member) function templates, for it leaves no room for user-defined conversions on `A`'s. Here is an example:

```
#include <algorithm>
#include <functional>
#include <iostream>

struct example {
  static void foo(int i) {
    std::cout << i << '\n';
  }
  typedef void  fun_t(int);
  operator  fun_t*() const {
    return  foo;
  }
};

int main() {
  int data[] = {3, 0, -1, -3, 16};
  example inst;

  // std::for_each(data,
  //          data + sizeof data/sizeof*data,
  //          std::ptr_fun(inst));

  std::for_each(data,
              data + sizeof data/sizeof*data,
              example::foo);
}
```

Let's look at the lines that were commented out and the function call expression `std::ptr_fun(inst)` in particular. What happens there is that name lookup finds the following two function templates in the scope of the namespace `std`:

```
template<class Arg, class Res>
pointer_to_unary_function<Arg,Res>
                    ptr_fun(Res(*)(Arg);

template<class Arg1, class Arg2, class Res>
pointer_to_binary_function<Arg1,Arg2,Res>
                    ptr_fun(Res(*)(Arg1,Arg2);
```

For the first function template there is one `P` and one `A`. P_1 is `Res(*)(Arg)`, which, in terms of the conventions used in the table of allowable forms, is a variant of `T(*)(T)`. The corresponding A_1 is the type `example`. It is obvious enough that there exist no such types for the template parameters `Res` and `Arg` that would make P_1 identical to A_1. There is, however, a user-defined conversion from an expression of type `example` to type `void(*)(int)`. If it was selected (like in the case of non-template functions), argument deduction would deduce `Res` to be `void` type and `Arg` to be `int`. But, as I said before, the language prohibits this and argument deduction fails. Quite similarly, template argument deduction fails for the second function template. This all leaves the set of candidates submitted to overload resolution empty, thus making the construct `std::ptr_fun(inst)` ill-formed in the context shown above.

Since you can always perform user-defined conversions inside the body of a function template, the restriction of precluding user-defined conversions on `A`'s can be easily circumvented. And a nice example of that can, not surprisingly, be found in the Standard C++ library. Let's take a look at the function object generator `bind2nd`, here is its possible implementation:

```
template<class BinOp, class T>
binder2nd<BinOp> bind2nd(const BinOp& op,
                         const T& second) {
  return binder2nd<BinOp>(op, typename
         BinOp::second_argument_type(second));
}
```

When called, this function template deduces `T` to be whatever type the second function argument has (except that it will effectively strip the type of the second argument of the possible `const` qualifier) and then explicitly converts it to the type required. Here is a small program that shows how this works:

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>

struct  example {
  operator int() const {
    return 3;
  }
};

int main() {
  int data[] = {3, 0, -1, -3, 16};
  example inst;
  std::transform(data,
   data + sizeof data/sizeof*data,
   std::ostream_iterator<int>(std::cout, "  "),
   std::bind2nd(std::modulus<int>(), inst));
}
```

There's also another way in which argument deduction from a function call expression differs from the cases I looked at in the previous sections. It differs in that it allows each P_i to be not identical to the corresponding A_i. The following is allowed:

- P_i can be more const/volatile qualified than A_i;
- if P_i is a pointer or pointer to member type, it can be a type to which an expression of type A_i can be converted via a qualification conversion;
- if P_i has one of the following forms: `class-template-name<T>`, `class-template-name<i>`, $\texttt{cv-seq}_{opt}$ `class-template-name<T>*`, or $\texttt{cv-seq}_{opt}$ `class-template-name<i>*` it can be a type derived from A_i.

Of course, providing that it is possible to find such types for the template parameters of P_i that P_i becomes identical to its A_i, these three alternatives are not considered.

The example of the use of the function template `bind2nd` that I just presented also demonstrates the case where `P`'s are different from its corresponding `A`'s. Indeed, in the function call

expression `std::bind2nd(std::modulus<int>(), inst)` A_1 is the type `std::modulus<int>` and A_2 is the type `example`. The corresponding `P`'s are: P_1 is `const BinOp`, P_2 is `const T`. And template argument deduction deduces `BinOp` to be `std::modulus<int>` and `T` to be `example`, which results in a call to the function template specialization:

```
binder2nd<std::modulus<int> >
      bind2nd(const std::modulus<int>&,
              const example&)
```

Given the way the deduction of template arguments from a function call expression goes, some familiar techniques can give surprising results. For example, the extensively studied in this article function template `bind2nd` is written in such a way that it does not allow either of its template parameters `BinOp` and `T` to be deduced to be a const-qualified type. Think about it. The same is true of, for example, the function template `make_pair` from the standard header `utility`.

Conclusion

The three cases that I explained in this article are not the only ones when the language uses template argument deduction. The other cases of note are "Deducing template arguments of a conversion function template", "Partial ordering of (member) function templates", "Deducing template arguments for explicit instantiation of a (member) function template", and "Referencing a (member) function template specialization in a friend declaration". Although not covered in this paper, these cases are based on the same principles and I believe that the information given in this article will assist the interested reader in mastering them, should such a need arise. As a starting point I can say that the topics "Deducing template arguments for explicit instantiation of a (member) function template" and "Referencing a (member) function template specialization in a friend declaration" are nearly identical to what I explained in "Deducing Template Arguments for Explicit Specialization of a (member) Function Template."

In any case, I'm now working on a follow-up to this article wherein I plan to give details of how things stand with respect to "Deducing template arguments of a conversion function template" and "Partial ordering of function templates."

I welcome any feedback from readers, which you can send directly to me at the address below.

Andrei Iltchenko
iltchenko@yahoo.com

Exceptional Java

By Alan Griffiths

It must be approaching twenty years since I visited the computer science department of the local university and on one of the notice board spotted a chart for the next decade of development in the industry. The feature of this that I remember was that every odd year predicted "the end of Fortran" and every even year "the end of Cobol". As the author (and I) expected, these languages are still thriving far beyond the end of that chart.

While to work in the software industry is to be exposed to constant change there is much that is constant in spite of that change. While we are regularly exposed to new tools (technologies, languages, techniques, etc) our existing knowledge and tools remain stubbornly useful. This is especially true when we distinguish the fundamental ideas from work-arounds for a specific tool.

For some years I've been very happy with the use of exceptions in C++ programs [Griffiths1999]. Recently I accepted a position at a company working primarily in Java, and consequently had to address the problems being encountered by developers using this language.

Received wisdom

Before I go on to describe the problems encountered by my new colleagues, let me revisit the "received wisdom" of the Java development community. This is represented clearly by the following quote from "Thinking in Java" [Eckel2000] (similar views are expressed by other sources):

> *Keep in mind that you can only ignore `RuntimeException`s in your coding, since all other handling is carefully enforced by the compiler. The reasoning is that a `RuntimeException` represents a programming error:*
>
> *An error you cannot catch (receiving a null reference handed to your method by a client programmer, for example).*
>
> *An error that you, as a programmer, should have checked for in your code (such as `ArrayIndexOutOfBoundsException` where you should have paid attention to the size of the array).*

It is sensible to use `RuntimeExceptions` to report programming errors – the availability of a call-stack aids reporting them meaningfully. The fact that they can be handled far up the call stack makes implementing an application-wide policy for handling them easier than trying to do so at every point an error is detected. (Examples of policies from different types of application domain are: to abort the current operation, to restart the subsystem, or to terminate the process.)

However, the "received wisdom" very clearly directs a developer towards using ordinary, checked exceptions (i.e. those not derived from `RuntimeException`) in the design of an application. The use of `RuntimeExceptions` is discouraged: "programming errors" do occur; but – beyond having a policy for dealing with them when detected – we should not be creating a design to cater for them! (Trying to cater for bugs only leads to hard to test code that is, itself, a breeding ground for bugs.)

The difference between theory and practice...

The developers that I joined had attempted to apply this guidance and run into a number of problems. However, because of pressure to "just write the code" no attempt had been made to formulate a workable design policy. Letting developers struggle on independently can waste a lot of time over the course of a project – so I called a meeting of the programmers on the team I was working with to discuss the problems they were having with exceptions.

We'll be examining more of the problems they described in the rest of this article – this section deals only with those that bear directly on the above theory (that checked exceptions should be used for everything that isn't a programming error). Before proceeding, I want to make it clear that this development group isn't the only one to experience these problems. They are in good company – as I confirmed at the ACCU Spring conference last year. It seems that this theory doesn't work in practice.

The relevant problems described fall into three categories.

Breaking encapsulation

Consider the example of a factory method that is responsible for creating objects. From the point of view of the client code there is no obvious reason why it should fail – so the interface doesn't have a throws clause. From the point of view of an implementation that retrieves objects from a database it is necessary to handle `SQLExceptions`. For the sake of this discussion let us assume that these reflect something catastrophic – like connectivity to the database being lost.

The `SQLExceptions` we are considering are not the result of programming errors. Accordingly, we are exhorted not to propagate them as unchecked exceptions. On the other hand we cannot handle them locally (except by the unhelpful expedient of returning a `null` reference). This leaves two options: either adding a throws clause to allow the factory method to propagate an `SQLException` or throwing our own exception (normally one that "wraps" the original exception).

Either approach places a burden on the client code – which will generally be in no better position to handle the error than the factory method itself. (Clearly, this is an iterative argument; but, somewhere far up the call stack there will be some code that manages the error.)

Loss of information

The strategy of "wrapping" exceptions prior to propagating them, alluded to in the last section, has the unfortunate side effect of making it difficult to detect the distinction between different problems programmatically. Essentially, one ends up with the situation that all that the programmer can be sure of is that "something went horribly wrong". Admittedly, that is often enough but it occasionally limits options. For example, how can one decide if is it worth retrying the operation that failed?

The alternative strategy (of allowing the original exceptions to propagate) can lead to a similar loss of information. Writing multiple, nearly identical, catch blocks is tedious and a potential source of the familiar maintenance issues caused by "cut and paste". Sooner or later, someone just writes "`catch (Exceptions e)`" – forgetting the (usually unintended) side effect that this also catches `RuntimeExceptions`.

Information overload

Depending upon whether exceptions are allowed to propagate or are "wrapped" two things can happen. Either, an increasing and incoherent set of exceptions begin to appear in the throws clauses of methods dependent on others – until developers get fed up with this insanity and introduce "`throws Exception`". Or, nearly every method has code to catch exceptions propagated from the methods it depends upon, "wrap" them in a new exception that makes sense within its interface and throw the new exception.

The theory sounds bad enough but, in practice, there is another thing that happens (although no developer admits to doing this intentionally). That is to consume an inconvenient exception by catching and ignoring it.

If these problems that occur in practice are not enough to justify considering alternatives there is a further difficulty: you may be coding to a function signature that doesn't allow you to throw any checked exceptions. This can happen when you are implementing an interface that you don't control (for example `Comparator`).

Fundamental ideas

In C++ I use exceptions to report problems that it is unreasonable to expect the immediate client code to deal with. In particular, the example of the factory function described above seems to satisfy these criteria: The part of the system that knows how to deal with loss of the database connection is likely to be many layers away from a factory function that retrieves objects from a database.

The problems related in the last section suggest that this approach isn't working – so either the approach is wrong or it is being implemented incorrectly. There are many differences between C++ and Java but there are also lots of similarities between them. Specifically, there are enough similarities in the exception handling mechanisms that I'd expect to use Java Exceptions for the same things that I'd use C++ exceptions for.

What the problems identified above illustrate is the ways in which Java checked exceptions are not like C++ exceptions. Declaring a checked exception places an obligation on the caller of a method to do something explicit with the exception – and that is precisely what isn't desired. What is desired is to transfer program flow in an orderly manner to some point far up the call stack.

There is something in Java that looks far more like my familiar C++ exceptions than Java's checked exceptions: Java's unchecked exceptions. To me they looked like the answer – it doesn't take much thought to conclude that approaching the above scenario by wrapping the `SQLExceptions` in an unchecked exception causes none of the above problems.

I outlined this approach at the meeting, and there was general consensus that it made sense, but a number of concerns were expressed: mostly regarding the choice that exists between the two exception-handling mechanisms. One of the senior team members agreed to use his notes from the meeting to draft a guideline "exceptions strategy" paper. This would be reviewed at a subsequent meeting when everyone had had an opportunity to think about it. (It is a good idea to review such solutions a few days later – it is very easy to be seduced by an attractive idea and overlook a killer issue during a brainstorming session.) The current version of this paper is included below.

Later on that day I was approached by one of the more thoughtful team member who was concerned that while he couldn't see anything wrong with what I was suggesting he couldn't find any books – or reference material on the internet – that agreed with it. I'm always pleased to be approached like this as I can be wrong, and much of the value of such meetings is lost if people don't think and research for themselves. In this case, I view this as a reflection of the immaturity of the Java community – the hype surrounding the language often gets in the way of recognising an issue and finding a solution. It took the C++ experts from 1990 (when they were introduced as "experimental" in the ARM [Stroustrup1990]) to 1997 (when the C++ standard library was revised to specify its behaviour in the presence of exceptions) to get a consensus on how to use exceptions.

I knew from my experience with C++ that there are ways to write code that works in the absence of the compiler prompting the developer to deal with exceptions. Indeed, I knew the fundamental ideas used in managing exceptions in C++ could also be applied to Java: I presented a translation of them at the ACCU conference 2001 [Griffiths2001]. (Anyway, it isn't the first time I've disagreed with authority – and it surely won't be the last!)

One more problem

There was one more significant problem that had been observed in a number of existing systems. In these, it had been found to be difficult to handle the errors reported via exceptions effectively. This was believed to be a result of every type of failure reported being reported using the same exception type – albeit with different message text. (If this sounds to you like the `java.sql` package and ubiquitous `SQLException` then you won't be surprised that this was mentioned.) The problem was actually worse than with the `java.sql` package since many parts of the system delivering differing types of functionality threw this same exception, and there was no equivalent of the well established (and fixed) set of `SQLState` values to deal with.

To address this we concluded that there needed to be guidance covering the choice of the specific exception to use. By requiring any exceptions specified in throws specifications to belong to the package that propagates them we hoped to discourage this habit. And, by checking conformance to the guidelines as part of the class design review, we encouraged a careful consideration of the contract between client code and implementation.

Exceptions policy

Following the review meeting the team adopted the suggested policy document. (It was updated to clarify it then and a couple of times later, but has remained close to the original discussion.) It reads as follows:

Background

There is at present no clear-cut policy for Java Exception Handling within any of the current OPUS Java systems. This has caused inconsistencies in the use of Exception Handling and these have resulted in problems.

This document addresses the use of two categories of exceptions: checked exceptions and unchecked exceptions.

Checked exceptions provide a mechanism for ensuring that the caller of a method deals with the issue they report. (Either by explicitly handling the exception, or by propagating it.)

Unchecked exceptions should only be considered for "long-distance" exception propagation. (To enable reporting of fairly catastrophic events within the system.)

To support these options all exceptions raised within the system will be subclasses of either `OpusException` (which extends `Exception`) or of `OpusRuntimeException` (which extends `RuntimeException`). These provided the facility to wrap exceptions.

Guidelines

It is the responsibility of the Class Designer to identify issues that would result in a checked exception being thrown from a class method. Those reviewing the class design check that this has been done correctly. Exception specifications are not changed during implementation without first seeking agreement that the class design is in error.

Exceptions that propagate from public methods are expected to be of types that belong to the package containing the method.

Within a package there are distinct types of exception for distinct issues.

If a checked exception is thrown (to indicate an operation failure) by a method in one package it is not to be propagated by a calling method in a second package. Instead the exception is caught and "translated". Translation converts the exception into: an appropriate return status for the method, a checked exception appropriate to the calling package or an unchecked exception recognised by the system. (Translation to another exception type frequently involves "wrapping".)

Empty catch-blocks are not used to "eat" or ignore exceptions. In the rare cases where ignoring an exception is correct the empty statement block contains a comment that makes the reasoning behind ignoring the exception clear.

In practice

This policy seems to have worked well both in terms of being followed without much difficulty by the original team members and for induction of new team members. Subsequently, other teams have also adopted it. To that extent it has been a success. However, it isn't the end of problems with the use of exceptions.

The remaining problems are much more manageable and fall into two categories:]

Catching exceptions at too low a level – rather than allowing them to propagate, they are caught by a piece of code that doesn't have sufficient context to deal with them effectively; and,

Catching too general a range of exceptions – for example, rather than catching `OpusException` and `SQLException` separately, and handling each, there is a single catch clause for `Exception` that then uses `instanceOf` to identify the exception type. Sometimes such code fails to take account of the possibility of `RuntimeExceptions` – and "eats" them.

These problems are not as widespread as those reported originally – indeed the majority of developers on the project are unaware of them. Both of these issues reflect poor coding technique and can be addressed by educating developers. This education could have occurred seamlessly as part of the project had we instituted code reviews; but I chose to postpone introducing them in favour of other process changes.

Conclusion

Java developers are rightly encouraged to use unchecked exceptions with caution. However, the current wisdom is too extreme. Unchecked exceptions in Java correspond to exceptions in C++, Smalltalk and C# – and shold be used in the same way: sparingly.

It seems that I'm not the only one to have doubts about this. Shortly after I wrote the first draft to this article Kevlin Henney pointed out that Bruce Eckel had opened discussion on the same point [Eckel].

Alan Griffiths
agriffiths@microlise.co.uk

References

[Eckel2000] Bruce Eckel, *Thinking in Java*, Prentice-Hall Inc, 2000, ISBN 0130273635

[Eckel] Bruce Eckel, "Does Java need Checked Exceptions?", `http://www.mindview.net/Etc/Discussions/CheckedExceptions`

[Griffiths1999] Alan Griffiths, "Here be Dragons", *Overload 40* or `http://www.octopull.demon.co.uk/c++/dragons/`

[Griffiths2001] Alan Griffiths, "Exception Safe Java", *ACCU Spring Conference*, 2001

[Stroustrup1990] Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

C++ Exceptions and Linux Dynamic Libraries

By Phil Bass

A Cautionary Tale

Once upon a time there was a bright young C++ programmer… Well, not so young, actually. And, sadly, not bright enough to know how to use C++ member functions in a Linux dynamic library. I'll tell you his poignant story in the hope that you can avoid his mistakes.

A Little Knowledge

I will call this programmer "Phil" (not his real name, apparently). Now, Phil had read the manual and he knew how to build a shared library using gcc. Just pass the `-fPIC` flag to the compiler (to generate position-independent code) and pass the `-shared` flag to the linker (to generate a shared library). He also knew how to load the library into his main program using `dlload()`, get the address of a function or data item in the library using `dlsym()`, check for errors using `dlerror()` and close the library using `dlclose()`. Now, `dlsym()` takes the name of a symbol and returns its address. But what is the name of a C++ member function? Typically, it is a mangled version of something like "`Isotek::SignalMonitor* Isotek::SignalMonitor::create()`". Not only is this quite a lot of typing, it is also tricky to find the mangled name and it will change with the slightest change to the function's name or interface.

Clever Clogs

"No sweat," thought Phil, for he had solved that problem when creating plug-in libraries for Win32 operating systems. The trick is to have the plug-in library insert its objects into a suitable registry when it loads. The main program creates the registry, loads the plug-in library and uses the objects that magically appear in the registry. No symbol look-ups required. Perfect.

Well, not quite. There was a two-way dependency between the main program and the plug-in library which, in Win32, meant that linking was a real pain. But Phil had another trick up his sleeve. Move the registry into its own shared library. Now both the main program and the plug-in library can be linked with the registry library in the normal way. The operating system automatically loads the registry library when the main program is started and it's already there when the plug-in library loads.

Pride Before The Fall

So, Phil built the registry library, the plug-in library and the main program, ran a test and saw that it was good. The main program found the object created by the plug-in library and correctly called its member functions. The problem of calling C++ functions with mangled names in Linux plug-in libraries was solved.

Then Phil tested an error condition (as every good programmer does). A function in the plug-in library detected the error and threw an exception; the exception handler in the main program failed to catch the exception and aborted. Phil checked his code carefully, but there was no obvious coding error. He poked around in the debugger, he tried to think of explanations for this behaviour, he discussed it with a colleague, but to no avail. So, finally, as an experiment, he tried to catch the exception within the function that threw it. The program still crashed. Exception handlers were not invoked for exceptions thrown from a function in the plug-in library.

Back To The Drawing Board

This was serious. Exceptions are often a good way of handling error conditions. The C++ standard library throws exceptions. Even the core language throws exceptions (`bad_alloc` and `bad_cast`). Did this mean that we could only use a subset of C++ in dynamic libraries? After more discussions and a search on the Web Phil discovered that the problem had been mentioned in a newsgroup post. The newsgroup thread contained just two messages. The first provided code that demonstrates the problem; the second said "but, it works for me". The difference had to be the compiler/linker switches. Sure enough, in a stripped-down sample program, exceptions were not caught when the `-nostartfiles` switch was provided, but were caught when this switch was absent.

Unfortunately, Phil needed the `-nostartfiles` switch. His strategy relies on the plug-in registering its objects when it loads. To do this, the programmer provides a function with C linkage called `_init()` and the operating system calls this function when the library loads. The programmer may also provide a `_fini()` function called when the library is unloaded. But, the C/C++ start-up files contain `_init()` and `_fini()` functions, too[1]. The `-nostartfiles` switch prevents "multiple definition" errors from the linker by suppressing the inclusion of the standard start-up files in the executable file.

The Punch-Line

The moral of this story? Don't supply your own `_init()` or `_fini()` function in Linux dynamic libraries containing C++ functions that may throw exceptions. That rules out C++ functions that use `new`, `std::vector`, `std::string` (to name but three), either directly or indirectly. And that doesn't leave very much.

Codicil

Phil fixed his problem by removing the leading underscore from the `_init()` and `_fini()` functions in the library. The re-named functions must now be called explicitly using `dlsym()` to look up their addresses. Note, however, that only these two functions need to be looked up by name and they are both simple functions with C linkage and unmangled names[2].

Phil Bass

phil@stoneymanor.demon.co.uk

---

[1] I confess I don't know what the `_init()` and `_fini()` functions in the start-up files do. Presumably they perform some initialisation of the C/C++ library and this includes initialisation of the exception handling mechanism.

[2] Thaddeus Frogley suggested a simple class that loads a dynamic library and makes the `init()`/`fini()` calls via `dlsym()`. Good idea. Thanks, Thad.

From Mechanism to Method – Function Follows Form

By Kevlin Henney

Is programming the manufacture of code? I would suggest that of all the metaphors applied to the development of software, manufacturing rates as perhaps one of the least useful and most harmful. Where in the manufacturing metaphor is the idea that programming is an act of communication? And not just with the compiler. Code is more often read than written, and writing code is just that: writing. You are an author with an audience: Today it may be just you and the compiler, but tomorrow it will include others… Which includes you: the "What was I thinking when I wrote this?" or "Which idiot wrote this? … Oh" syndrome.

This perspective lends a quite different weight to the use of language features in a program. In C++ we have a formal notation for working with concepts as close to or as far from the metal as we chose. The compiler cares little for how clearly we write the code, how fit for purpose it is, or how we work in teams to develop systems. It cares only for the well-formedness of the code as C++ (or at least the compiler's closest approximation). All those other, non-functional considerations are about code as a means of communication with others.

C++ offers an extensive shopping list of mechanisms. It is left to the programmer to make sense – and sensible use – of these, bringing method and clarity to bear on the expression of code, coding to communicate intent idiomatically to others. But too often we find that code looks like, well, code: a cipher whose key is known privately and exclusively to its author – and sometimes, alas, even this much is not true.

Overloading – especially operator overloading – is one of those mechanisms that, when first encountered, can raise eyebrows and open mouths. This response comes in two opposing flavours: "Great! I can see that we could use this all over the system" or "Oh no, I don't think that's for us. Sounds different to what we normally do… too radical". The former can often lead to the most cunning of ciphers with the clarity of hieroglyphics (pre Rosetta Stone); the latter to verbose code that misses the effectiveness of established idioms and the benefits of template-based generic programming. There is, however, a centre ground of practice between these two.

Principles

> *Express coordinate ideas in similar form.*
> *This principle, that of parallel construction, requires that expressions similar in context and function be outwardly similar. The likeness of form enables the reader to recognize more readily the likeness of content and function. Strunk and White [Strunk+1979]*

This advice works as well with the written keyword as it does with the written word. It expresses the idea that similar constructs should have similar meanings, a goodness of fit between intent and realization, interface and implementation, reader and writer. This principle of substitutability [Liskov1987] is often expressed with respect to inheritance and runtime polymorphism [Coplien1992], but applies equally well to the compile-time polymorphism you have with conversions, overloads, and templates [Henney2000a].

Common Name Implies Common Purpose

The principle that overloaded functions work to similar ends is the one that makes the most sense of this feature. As a practice it frees programmers from mangling names to distinguish otherwise similar functions with differing argument lists (this is the job of the compiler).

Following well-established conventions, where possible, clearly makes sense. For instance, the standard C++ library establishes a common set of names and semantics, conventions clarifying that `empty` means "is empty?" not "to empty", `clear` means "to clear" and not "is clear?", etc. Note that judgement and resourcefulness are still needed:

- The standard defines a relatively small set of names, clearly not enough to cover your whole domain of application.
- The standard is not always consistent in its use of names, e.g. the unhelpfully named `get` member function in `auto_ptr` is a query without a side effect, whereas `get` on a `basic_istream` is a query with a significant side effect.
- There are other well-established sources of terminology that provide names you can draw upon. There may be times these clash with the standard. For example, depending on context, `begin` yields an iterator or initiates a transaction. It is for you to determine whether or not such overloading of meaning, as well as name, is clear.

Operator Underloading

Whatever care is applied to the use of named function overloading applies doubly so to operator overloading. It can be a fertile ground for fertile imaginations. An opportunity to communicate clearly or to resurrect a Tower of Babel.

The built-in types both set expectations in the reader and offer a spec for the writer: "When in doubt, do as the `int`s do" [Meyers1996]. As with any style principle, this one is elasticated: `operator+` for the standard `basic_string` is not commutative, but its meaning is clear nonetheless. Bitshift operators, `operator<<` and `operator>>`, for I/O stream insertion and extraction stretches the elastic taut by an appeal to scripting notations. However, the long history and established presence in the standard library qualifies this idiom as *effectively* built-in. Do not assume the same distinguished fate awaits any other 'creative' operator deployment! So, as a corollary, it may be worth considering that when in serious doubt, do not do it.

In deciding the suitability or otherwise of operator overloading, keep in mind that it only really makes sense for value-based [Henney2000b] rather than indirection-based objects. Value-based objects represent fine-grained information concepts, typically live on the stack or embedded within other objects, and are passed around by copy or `const` reference. Syntactically this emphasizes their value and allows easy use of operators. Indirection-based objects, by contrast, represent more significant chunks of system information or behaviour, typically live on the heap, and are passed around by pointer. Syntactically this emphasizes their identity but makes use of operators awkward: having to dereference the pointer explicitly before being able to use an operator somewhat defeats the intended transparency of operator overloading.

Smart Pointers

One of the most common C++ idioms involving overloading is the SMART POINTER, ranging from reference-counted pointers to the essentially simple but surprisingly intricate standard `auto_ptr`. However, it is a common myth that all smart pointers are concerned with memory management, and that all smart pointers support `operator->` and `operator*` as their pointer-like operations.

The Three Rs

Use determines definition, and clearly not all smart pointers are intended for the same use. We can consider operators for pointers in three categories, the three Rs: (de)referencing, relational, and arithmetic. According to purpose, we can select if and how we provide these:

- Dereferencing comes in the familiar forms of `operator*` and `operator->`, as well as the less familiar and often overlooked `operator->*` [Meyers1999] and `operator()`.
- Relational operators make sense for pointer or smart pointers that have a natural ordering, such as raw pointers in the same array or random access iterators. Having only equality (and hence inequality) comparison makes sense for many other pointers, such as reference-counted pointers. They typically test for identity rather than value, which is why `auto_ptr` does not support such comparison: Exclusive ownership means that in a well-formed system `auto_ptr` equality comparison will always return `false`.
- Pointer arithmetic, such as `operator++` and `operator+`, makes sense for smart pointers that encapsulate some concept of interval or progression, such as iterators.

Function Objects

A common piece of advice offered to developers making a transition from procedural to object-oriented code is that a class should not model a function. Such classes are often named as actions, and typically sport a principal or single member function named "do such and such". While this advice does guard against a common pitfall, it is not always poor practice. Those that have taken this rule of thumb to heart as a legalistic rule need to unlearn a little to appreciate how objects can encapsulate tasks and, in particular, mimic functions. The COMMAND pattern [Gamma+1995] demonstrates the power of task-based objects. The FUNCTOR idiom [Coplien1992] focuses on functional objects that overload `operator()` to achieve the appearance and transparency of use of conventional functions.

The standard library provides for the use of function objects with generic functions and templated containers, categorizing them as unary or binary functions. It also defines specific function object classes – e.g. `less` for ordered comparison – and function object adaptors – e.g. `pointer_to_unary_function` to wrap up naked function pointers. The Boost library [Boost] extends this with other function object classes, adaptors, and the *nullary* function category, for function objects taking no arguments.

Re-member

As an example of a function object class, focusing on the nullary form for `void` returning functions, **Listing 1** shows code for a member function adaptor. You may have already come across the `mem_fun_t` family of adaptors in the standard library. However, there are key differences:

- A `remember_function` bundles a target object together with a member function pointer for later callback through nullary `operator()`, whereas a `mem_fun_t` object simply holds a member function pointer and uses the argument to `operator()` as its target.
- Although it is of little practical consequence for a nullary, `void`-returning function, a variant for `const` member functions is not required because the member pointer's type is parameterized as a whole.
- The target pointer type need not be a raw pointer: smart pointers supporting `operator->*` will also work.
- The member pointer type need not be a member function pointer: a member data pointer that points to a nullary function object will also work.

The `remember` template function is a helper that simplifies composition of `remember_function` objects, automatically deducing the parameter types in the manner of `make_pair` for `pair`, `bind2nd` for `binder2nd`, or `ptr_fun` for `pointer_to_unary_function` and `pointer_to_binary_function`.

Generalized Function Pointer

The need for event-driven callbacks, such as timer-triggered actions, is often met with pointers to functions or an implementation of OBSERVER [Gamma+1995]. The former approach is fine for simple event handlers:

```
class timer {
public:
  void set(const time &delay,
                    void (*callback)());
  ...
};
```

But it is inflexible, handling only functions and not context objects. The OBSERVER-based solution introduces a base class that a concrete handler class must implement:

```
template<typename target_ptr_type,
         typename member_ptr_type>
class remember_function {
public:
 remember_function(target_ptr_type on,
                   member_ptr_type call)
   : ptr(on), member(call) {}
 void operator()() const {
   (ptr->*member)();
 }
private:
 target_ptr_type ptr;
 member_ptr_type member;
};

template<typename target_ptr_type,
         typename member_ptr_type>
remember_function<target_ptr_type,
                  member_ptr_type>
remember(target_ptr_type on,
         member_ptr_type call) {
 return remember_function<target_ptr_type,
                member_ptr_type>(on, call);
}
```

Listing 1. Function object class and helper for binding target object and member function pointer.

```
class handler {
public:
  virtual void run() = 0;
  ...
};

class timer {
public:
  void set(const time &delay,
           handler *callback);
  ...
};
```

However, this introduces a level of indirection that leads to additional memory management responsibilities, and imposes an intrusive base class participation on users for what is a relatively simple scenario. Using arbitrary objects or functions for callback would be preferred. Overloading multiple **set** member functions in **timer** is a kitchen-sink solution, leading to a wide interface that attempts to please all people and an awkward **timer** implementation.

Function objects at first appear to offer a route out: A nullary function pointer or object could be passed in, including a **remember_function** binding of member to target, and later called back. A member template function would accommodate the substitutability of all the variations:

```
class timer {
public:
  template<typename nullary_function>
  void set(const time &delay,
           nullary_function callback);
  ...
};
```

However, this raises a fundamental problem: How does a **timer** object later execute the **callback** passed in? Unlike many examples of member template functions in the standard library, this one does not execute the function or function object immediately – it would not be much of a timer if it did! The **timer** needs to store the callback for later use. Without parameterizing the whole **timer** class on the **nullary_function** type, rather than just the **set** member, this does not appear to be possible. Templating the whole class is undesirable because it means that for each different type of callback, a different timer class instantiation is needed.

A further problem with the member template approach is that a member template function cannot be declared **virtual**. This would be significant if the **timer** class were an abstract rather than a concrete class, i.e. an interface to timer features rather than a single implementation. The attempt to decouple both the mechanism of the timer and the target type like this would lead to the following illegal code:

```
class timer
{
public:
  template<typename nullary_function>
                              // illegal
  virtual void set(const time &delay,
         nullary_function callback) = 0;
  ...
};
```

On the Outside

It is possible to resolve the tension in the design by approaching it from a different angle. We can take a step back and ask what simple interface to **timer** would also simplify its implementation. What is needed is some kind of abstraction of a function pointer that is both generic and generic: generic in the sense of supporting the generic programming style of the STL, and generic in the sense that it is general purpose and easily used in any context:

```
class timer {
public:
  void set(const time &delay,
           const function_ptr &callback);
  ...
};
```

To satisfy the requirements for simplicity in **timer** and our expectations of a function pointer, **function_ptr** needs to support syntax for initialization, assignment, and execution. **Listing 2** shows such an interface.

```
class function_ptr {
public:
  function_ptr();
  function_ptr(const function_ptr &other);
  template<typename nullary_function>
    function_ptr(nullary_function function);
  ~function_ptr();
  function_ptr &operator=(
                const function_ptr &rhs)
  void operator()() const;
  ...
};
```

Listing 2. Smart function pointer interface.

On the Inside

This is all very well, but it has yet to solve the problem fully: It looks nice, but how is it implemented? How can a **function_ptr** object hold arbitrary representation, constrained only by the requirement that it must support an **operator()** with no arguments? The technique used is based on the EXTERNAL POLYMORPHISM pattern [Cleeland+1998], in particular the use of inheritance and runtime polymorphism to adapt template-based genericity for value-based objects through a level of indirection [Henney2000b]. **Listing 3** opens up **function_ptr** to show this collaboration in practice, including the conversion (i.e. initialization) from any arbitrary nullary function object or pointer.

Clone Me

function_ptr is a value type, so it stands to reason that it should support copying through construction and assignment – an identity form of inward conversion [Henney2000b]. The body of a **function_ptr** cannot be copied directly because of the decoupling of interface from implementation, which leads to the polymorphic copying, or cloning, technique shown in **Listing 4**.

```
class function_ptr {
public:
  function_ptr()
    : body(0) {}
  template<typename nullary_function>
  function_ptr(nullary_function function)
    : body(new adaptor<nullary_function>
                              (function)) {}
  ~function_ptr() {
    delete body;
  }
  ...
private:
  class callable {
  public:
    virtual ~callable() {}
    virtual callable *clone() const = 0;
    virtual void call() = 0;
  };
  template<typename nullary_function>
  class adaptor : public callable {
  public:
    adaptor(nullary_function function)
      : adaptee(function) {}
    virtual callable *clone() const {
      return new adaptor(adaptee);
    }
    virtual void call() {
      adaptee();
    }
    nullary_function adaptee;
  };
  callable *body;
};
```

Listing 3. Smart function pointer representation and basic construction.

```
class function_ptr {
public:
  ...
  function_ptr(const function_ptr &other)
    : body(other.body
             ? other.body->clone()
             : 0) {}
  function_ptr &operator=(
             const function_ptr &rhs) {
    callable *old_body = body;
    body = rhs.body
             ? rhs.body->clone()
             : 0;
    delete old_body;
    return *this;
  }
  ...
};
```

Listing 4. Smart function pointer copying.

The assignment operator uses the COPY BEFORE RELEASE idiom [Henney1998] for exception- and self–assignment-safety. A non-throwing `swap` could also be used for this [Sutter2000], but for this article the interface to `function_ptr` is being kept small and based only on operators.

Call Me

The final piece of the jigsaw is to dereference a `function_ptr` – fetch and execute. A raw function pointer supports dereferencing through `operator*`, which is the identity operation on a function pointer, `operator()`, which can be called directly on a function pointer without using `operator*`, but no `operator->`. This is the model that `function_ptr` should follow, and does so in **Listing 5**. For a null pointer, the execution assumes that for no function there is no function, as opposed to undefined behaviour as per built-in pointers.

```
class function_ptr {
public:
  ...
  void operator()() const {
    if(body)
      body->call();
  }
  function_ptr &operator*() {
    return *this;
  }
  const function_ptr &operator*() const {
    return *this;
  }
  ...
};
```

Listing 5. Smart function pointer dereferencing and calling.

Remember Me?

A focus on forms of substitutability – in this case derivation, overloading, and templates, each a way of establishing an interface – can decouple a system, allowing greater suppleness and clearer code. Putting it all together, we can put together a simple scenario based around the proposed `timer` interface. Consider the interface to a device that can be turned on or off at particular times, e.g. a heating or an air conditioning system:

```
class device {
public:
  virtual void turn_on() = 0;
  virtual void turn_off() = 0;
  ...
};
```

The following example combines the concepts and features presented so far:

```
void set_up(device *target, timer *scheduler,
const time &on, const time &off) {
  scheduler->set(on,
        remember(target, &device::turn_on));
  scheduler->set(off,
        remember(target, &device::turn_off));
}
```

Conclusion

There is little that excites programmers' passions more than discussions of style, but there is little that helps them more than common understanding. Overloading is a powerful feature whose reasoned use underpins many idioms at the heart of the modern C++ programmer's vocabulary: smart pointers, function objects, iterators, etc.

Within each of these idioms there is variation for expression rather than any simplistic, one-size-fits-all, cookie-cutter rule. A function object will support some form of `operator()`, and a smart pointer must support some form of dereferencing, but this does not by necessity include `operator->`, as demonstrated by `function_ptr`, a smart function pointer.

Kevlin Henney
kevlin@curbralan.com

References

[Boost] Boost library website, `http://www.boost.org`.
[Cleeland+1998] Chris Cleeland, Douglas C Schmidt, and Tim Harrison, "External Polymorphism", *Pattern Languages of Program Design 3*, edited by Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998.
[Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.
[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[Henney1998] Kevlin Henney, "Creating Stable Assignments", *C++ Report* 10(6), June 1998, also available from `http://www.curbralan.com`.
[Henney2000a] Kevlin Henney, "From Mechanism to Method: Substitutability", *C++ Report* 12(5), May 2000, also available from `http://www.curbralan.com`.
[Henney2000b] Kevlin Henney, "From Mechanism to Method: Valued Conversions", *C++ Report* 12(7), May 2000, also available from `http://www.curbralan.com`.
[Liskov1987] Barbara Liskov, "Data Abstraction and Hierarchy", *OOPSLA '87 Addendum to the Proceedings*, October 1987.
[Meyers1996] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.
[Meyers1999] Scott Meyers, "Implementing `operator->*` for Smart Pointers", *Dr. Dobb's Journal*, October 1999.
[Strunk+1979] William Strunk Jr and E B White, *The Elements of Style*, 3rd edition, Macmillan, 1979.
[Sutter2000] Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000.

Template Titbit – A Different Perspective

By Phil Bass

Background

Oliver Schoenborn's article, "Tiny Template Tidbit", in Overload 47 illustrates some of the wonderful things you can do with templates. I particularly liked the way it described the thought processes Oliver went through when designing some curve-fitting software and his clear explanation of the problem he set himself. Whilst reading an earlier draft of "Tidbit" a slightly different solution occurred to me, which I shared with Oliver before publication. With the deadline approaching there wasn't much time for discussion, so we agreed that I should write up those ideas that seemed interesting, but that were not strictly relevant to Oliver's article.

Recap

The "Tidbit" article used a `fitCurve()` function template to illustrate the techniques of interest:

```
template <class CoordContainer>
Curve fitCurve(
          const CoordContainer& coords) {
  // ...
  // set coord1 to first Coord in coords
  // set coord2 to second Coord in coords
  Coord c = coord1 + coord2;
  // ...
}
```

The challenge was to make this template work for containers whose elements are: points, objects that contain a point, or pointers to any of those types.

Oliver called his point class "Coord". The key to his solution was a `getCoord()` function template that returned the `Coord` corresponding to a given container element. Different specialisations of `getCoord()` were defined for different types of element.

```
// General function template
template <class PntClass> inline
const Coord& getCoord(const PntClass& p)
{ return p.coords; }

// Partial specialization for pointers
// to things
template <class PntClass>
const Coord& getCoord(const PntClass* p)
{ return p->coords; }

// Complete specialization for Coord
inline
const Coord& getCoord(const Coord& p)
{ return p; }

// Complete specialization for pointer
// to Coord
inline
const Coord& getCoord(const Coord* p)
{ return *p; }
```

This version of the `getCoords()` function template covers element types that are: `Coord`, pointer to `Coord`, something containing a 'coords' data member, and pointer to something containing a 'coords' data member. It doesn't cover elements whose `Coord` is accessed via a member function. For this, a `getData()` function template was introduced and the appropriate `getCoord()` specialisations were re-written in terms of `getData()`.

```
template <class PntClass>
const Coord& getCoord(const PntClass& v) {
  return getData(v, &PntClass::coords);
}

// partial specialization for pointers
template <class PntClass>
const Coord& getCoord(const PntClass* v) {
  return getData(*v, &PntClass::coords );
}

template <class PntClass>
inline
const Coord& getData(const PntClass& pp,
                     Coord PntClass::* dd) {
  return pp.*dd;
}

template <class PntClass>
inline
const Coord& getData(const PntClass& pp,
   const Coord& (PntClass::* mm)() const) {
  return (pp.*mm)();
}
```

This is a neat and, as far as I know, original idea that makes it possible to use either data members or member functions interchangeably in generic functions.

Suggestions

I made three suggestions:

1. pass a pair of iterators to the curve-fitting function rather than a const reference to a container,
2. use a traits class to extract the co-ordinates from objects of arbitrary type and
3. use a curve-fitting function with "external state".

Oliver liked the traits idea, but found it didn't give him what he wanted. And, from his perspective, the other two suggestions just weren't relevant to the problem at hand. After some thought I realised that it was all a question of your point-of-view. As Kevlin Henney often points out, the solution to a problem depends on the context. Oliver had chosen the context of a specific project; I was thinking of a general-purpose library.

Let's look at these ideas in a bit more detail.

Iterator Range or Container?

Passing a pair of iterators to the `fitCurve()` function is more general than passing a container, but it's often less convenient for the calling code. An iterator range allows containers without `begin()` and `end()` functions to be used (such as arrays) and makes it easy to fit a curve to a subset of the points in a given container. On the other hand, it means passing two parameters instead of one.

For a general-purpose library flexibility is very important. Oliver's project doesn't need that much flexibility, so ease-of-use dominates the trade-off. Context matters here.

Traits Classes or Function Templates?

The "Tidbit" article shows how to write some generic components that support containers in which the element type is in one of four fairly general categories. In that article Oliver makes the point that this may not be general enough for a library and I agree. What I had in mind was a traits class template that defines a mechanism for accessing the co-ordinates of a point class. The following template and partial specialisation illustrates the idea for points in the X,Y plane.

```
// The general version of the point
// traits class.
template<typename point_type>
struct point_traits {
  static double x(const point_type& p) {
    return p.x();
  }
  static double y(const point_type& p) {
    return p.y();
  }
};

// A partial specialisation of the point
// traits class for pointers.
template<typename point_type>
struct point_traits<point_type*> {
  static double x(point_type* p) {
    return p->x();
  }
  static double y(point_type* p) {
    return p->y();
  }
};
```

The general version of the template provides co-ordinate access functions for point classes having member functions `x()` and `y()`. The partial specialisation does the same for *pointers* to classes with `x()` and `y()` member functions. For any other type of point class the user would have to define a further specialisation of the traits template. For example:

```
// A complete specialisation of the point
// traits class for POD_Point.
struct POD_Point { double x, y; };

template<>
struct point_traits<POD_Point> {
  static double x(const POD_Point& p) {
    return p.x;
  }
  static double y(const POD_Point& p) {
    return p.y;
  }
};
```

The `point_traits<>` member functions are similar to Oliver's `getCoord()` functions. They are *template* functions and their purpose is to extract information from some sort of point object. They are used in the curve fitting algorithm in a similar way. My implementation of a least-squares line algorithm, for example, contains the following lines:

```
template<typename point_type>
void add(point_type point) {
  double x = point_traits<point_type>::x(point);
  double y = point_traits<point_type>::y(point);
  // ...
}
```

In principle, the traits member functions could be implemented in terms of another function template, like Oliver's `getData()`, which would make it possible to handle both data members and member functions out of the box. I prefer not to do this, though. The traits mechanism is general enough to be used with arbitrary types of point object and the inconvenience of having to define a traits specialisation for point classes without the `x()` and `y()` access functions is small. In fact, the lack of support for data members could even be seen as an advantage – it should encourage the use of fully-fledged, properly encapsulated point classes. More importantly, though, I wouldn't want the name of a member of a class in the client code to appear in my library functions the way 'coords' appears in Oliver's `getCoord()` functions. If we were to do this in a general purpose library what would we call the member? 'coords', 'point', 'm_2Dpoint', 'xyPoint_', … The list is endless!

Of course, if you are working on a project that already has to deal with a variety of point classes, some of which use data members, the need for extra traits specialisations could be a nuisance. But, then again, how many different point classes do you want to use in the same application? Not many, I think.

Once again, context matters.

External or Internal Algorithm State?

The only curve fitting algorithm I am familiar with is the least-squares line and its 3-dimensional cousin, the least-squares plane. In fact, the least-squares line can be defined as follows:

> Given a set of points (x_i,y_i), where i = 1..n, the least-squares line through those points is given by y = mx + c, where:
> $m = (n? x_iy_i - ? x_i? y_i) / (n? x_ix_i - ? x_i? x_i)$ and
> $c = (? y_i - m? x_i) / n$

A straightforward implementation of this formula in code involves four totals ($? x_i$, $? y_i$, $? x_ix_i$ and $? x_iy_i$) and a point count. Old fashioned C++ code might look like this…

```
// Initial data.
struct Point {double x, y;};
Point point[20] = { ... };
unsigned n = 0;
double x_sum = 0, y_sum = 0,
       xx_sum = 0, xy_sum = 0;

// Accumulate sums for each (x,y)
// point...
for (int i = 0; i < 20; ++i) {
  ++n;
  x_sum += point[i].x;
  y_sum += point[i].y;
  xx_sum += point[i].x * point[i].x;
  xy_sum += point[i].x * point[i].y;
}

// Calculate slope and offset.
double m = (n * xy_sum - x_sum * y_sum)
         / (n * xx_sum - x_sum * x_sum);
double c = (y_sum - m * x_sum) / n;
```

The algorithm reminded me of the standard accumulate function, so my first thought for a least-squares function in the modern style was modelled on `std::accumulate()`.

```
// A line in the X,Y plane.
class line;

namespace least_squares {
  // The current state of the least-
  // squares line algorithm.
  struct state {
    state() : n(0), x_sum(0), y_sum(0),
              xx_sum(0), xy_sum(0) {}
    operator line() { ... }
    // ...
    unsigned n;
    double x_sum, y_sum,
           xx_sum, xy_sum;
  };

  // Fit a least-squares line to the
  // (x,y) points in [first,last)
  template<typename iterator_type>
  state fit(iterator_type first,
            iterator_type last,
            state = state());
}
```

The curve fitting function is given an initial state and returns a new state after accumulating points in the iterator range [first,last). This is what I have called a "function with external state". The default initial state corresponds to no points accumulated.

The state class has a conversion operator that provides an implicit conversion to a line. The combination of external state and conversion operator makes it possible to fit a least-squares line in several stages.

```
using least_squares::state;
using least_squares::fit;

state algorithm_state =
            fit(point, point + 10);

line best_fit_line =
            fit(point + 10, point + 20,
                algorithm_state);
```

Unfortunately, there are several unpleasant features in this design. The state class has public data and an implicit conversion operator. Passing the state into and out of the `fit()` function by value leads to unnecessary copying. And the simple case of fitting a least-squares line to all the points in a container carries the excess intellectual baggage of the state object.

In thinking about these points I finally settled on a design offering two interfaces that differ in generality and convenience. The less general/more convenient interface is provided as a single template function, `least_squares::fit()`.The more general/less convenient interface is provided as two classes: `least_squares::state` and `least_squares::add_point`. The point traits idea has been retained to allow the underlying implementation to adapt to different point classes.

```
// Summary of least-squares components
// for the X,Y plane.
class point;
class line;
template<typename T> struct point_traits;

namespace least_squares {
  // General interface.
  class add_point;
  class state;
  // Convenient interface.
  template<typename iterator_type>
  line fit(iterator_type first,
           iterator_type last);
}
```

The state class is at the heart of this new design. It now properly encapsulates its data and provides a minimal interface. It has just two member functions: one to add a point and one to create a line.

```
// The state of the least-squares
// line algorithm.
class least_squares::state {
public:
  state();
  template<typename point_type>
  void add(point_type point);
  ::line line();
private:
  unsigned n;
  double x_sum, y_sum, xx_sum, xy_sum;
};
```

The `add()` function uses the point traits template to extract individual co-ordinates from each point, calculates new values for the four totals and increments the point count.

```
template<typename point_type>
void least_squares::state::add(
                           point_type point) {
  ++n;
  double x =
         point_traits<point_type>::x(point);
  double y =
         point_traits<point_type>::y(point);
  x_sum += x;
  y_sum += y;
  xx_sum += x * x;
  xy_sum += x * y;
}
```

The `line()` function calculates the slope and offset of the least-squares line and returns a line object. At least two points must have been accumulated before this function is called. The pre-condition is checked using `assert()`, here; throwing an exception or the use of a policy class might be more appropriate for a real library.

```
::line least_squares::state::line() {
  assert(n > 1);
  double m = (n * xy_sum - x_sum * y_sum)
           / (n * xx_sum - x_sum * x_sum);
  double c = (y_sum - m * x_sum) / n;
  return ::line(m, c);
}
```

The state class actually provides everything we need to find the least-squares line for a set of points. On its own, however, it is slightly inconvenient to use. The programmer using the curve fitting library needs to iterate through the points, either by writing a loop or by passing a suitable function object to `std::for_each()`. The `add_point` class provided by the library is just the right sort of function object for `std::for_each()`. Its function call operator simply calls the `state:add()` function.

```
class least_squares::add_point {
public:
  add_point(state& s) : algorithm_state(s) {}
  template<typename point_type>
  void operator() (point_type point) {
    algorithm_state.add(point);
  }
private:
  state& algorithm_state;
};
```

With this convenience class, even the general interface to the least-squares algorithm becomes quite easy to use, as the following example shows.

```
using std::foreach;
using least_squares::state;
using least_squares::add_point;
// Find the line that best fits
// point[0]..point[19].
state best_fit_data;
for_each(point, point + 20,
         add_point(best_fit_data));
line best_fit_line = best_fit_data.line();
```

The alternative interface just wraps up these three lines of code in a template function.

```
template<typename iterator_type>
line least_squares::fit(iterator_type first,
                        iterator_type last) {
  state algorithm_state;
  std::for_each(first, last,
                add_point(algorithm_state));
  return algorithm_state.line();
}
```

The previous curve fitting example can now be re-written as a one-liner, which I leave as an exercise for the interested reader.

So, my answer to the external/internal state question is, "Both". The general interface uses external state in the form of an add_point function object; the convenient interface keeps the algorithm state entirely within the fit() function. Once again, context matters, and this time the library can not guess which is the more important.

Final Thoughts

The curve-fitting code presented here illustrates how Object-Oriented Programming and Generic Programming can work together. The `least_squares::state` class obeys the principles of OOP; the `fit()` function is a generic algorithm. Together they provide a safe, flexible and efficient software component.

The difference between Oliver's code and mine comes down to the context of the problem. It's like spelling. The dictionary in my version of Microsoft Word says "tidbit" is an error and offers "titbit" as the correct spelling. My Chambers dictionary has both spellings. Context matters!

Phil Bass
phil@stoneymanor.demon.co.uk