# contents



## Copy Deadlines

All articles intended for publication in *Overload 71* should be submitted to the editor by January 1st 2006, and for *Overload 72* by March 1st 2006.

# credits & contacts

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

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

**Advisors:**

**Phil Bass**
phil@stoneymanor.demon.co.uk

**Thaddaeus Frogley**
t.frogley@ntlworld.com

**Richard Blundell**
richard.blundell@gmail.com

**Pippa Hennessy**
pip@oldbat.co.uk

**Advertising:**
**Thaddaeus Frogley**
ads@accu.org

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

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

**Information and Membership:**
Join on the website or contact

**David Hodge**
membership@accu.org

**Publications Officer:**
**John Merrells**
publications@accu.org

**ACCU Chair:**
**Ewan Milne**
chair@accu.org

# Editorial: The "Safe C++ Standard Library"

At a conference some years ago a group of us developed an analogy for software development over a series of lunchtime conversations. (I won't mention names as my memory is sufficiently vague as to who participated in these conversations and who was simply around at the time.) The analogy was with the preparation of food. The circumstances under which software is developed vary from "fast food" to "gourmet" – and the way it is developed differs just as much. At the time the point of these discussions and the building of this analogy was a discussion of the differences between types of developer and the way in which they approach their work.

One of the points we explored was that the approach to developing software reflected the context in which the developers work. There is a place in the world for quickly developed "burger and fries" software – I occasionally write throwaway scripts to munge data from one format to another or to generate test data. This type of software is effective in meeting the immediate needs of the business, but achieves this with potentially poor error detection and handling, or with performance characteristics that scale poorly, and is written with no thought to reuse or maintenance. Also, it is all too common for such software to be unusable without the author on hand to deal with odd behaviour. There is, of course, a medium to long term cost of this sort of diet. Since this conversation Morgan Spurlock shot to fame with his demonstration of this in the context of food: "Supersize Me". (And a similar problem arises when organisations clog up their arteries with this sort of software.)

For the more discerning organisation there is a requirement for developers to craft something that can be used without the author standing by to fix problems if they occur, or that may be maintained over a long period of time, or used in a variety of contexts, or meets stringent performance characteristics. Different skills come into play when writing such software. A lot more care with the preparation of the ingredients, a different set of tools and a lot more thought. But the results are worth the effort: a diet of quality software base makes for a fitter and happier organisation.

One of the things that we were considering was the mismatch that occurs if a developer accustomed to working in one way encounters a situation that required the other approach. We can all imagine the consequence of swapping the kitchen staff of a burger chain with those from a good restaurant. (I hope I've not just invented another theme for "reality TV".) When this happens the frustration of all involved in such situations will be obvious – even if the true causes are not. All too often the incompetence is assumed where a failure to communicate what is needed is the cause.

During these conversations we also examined the role that tools played in the analogy – are great tools required to produce great software? Or can bad software be avoided by the use of good tools? It was my contention that it is the skills that matter: a top class chef would be able to produce good quality food even when separated from their kitchen. On the other hand, if separated from the freezer full of frozen burgers the results would not be so good for the McDonald's kitchen staff. The killing argument against this however was that "a real chef would not be separated from his knives".

These conversations – especially the usefulness of potentially dangerous tools to an expert in the craft – came to mind recently. I was reading the reactions of a group of C++ experts to the recent discovery that a vendor plans to ship an implementation of C++ that produces messages like the following:

```
c:\program files\microsoft visual studio
8\vc\include\algorithm(637) : warning C4996:
'std::swap_ranges' was declared deprecated
c:\program files\microsoft visual studio
8\vc\include\algorithm (625) : see
declaration of 'std:: swap_ranges'
Message: 'You have used a std:: construct
that is not safe. See documentation on how to
use the Safe Standard C++ Library' ...
```

This discussion was the first I'd heard of the "Safe Standard C++ Library" – which is a bit surprising as for some years I've been involved with both the BSI and ISO working groups that define the standard C++ library. And, as it was the latter group who were discussing this development, I'm pretty sure most of the rest of them did not know about it either. We were also surprised to see the term "deprecated" used in this context – it has a technical meaning within the standard that is not applicable here.

Let me be quite clear about this: the so called "Safe Standard C++ Library" has no standing outside of Microsoft – it is neither an official standard nor a de-facto one. Also the ISO C++ working group has not deprecated `std::swap_range` (or any of the other

functions that can lead to these messages appearing). So, what on earth is this all about?

There are representatives of Microsoft that participate in the standardisation of C++, and they were able to supply some of the details. It seems that Microsoft have identified a number of "unsafe" constructs in the standard C and C++ libraries: that can overwrite memory, or functions that return pointers to static data, or ... There is nothing very contentious about there being "sharp knives" in the C++ library – although there may be some debate about some of the specific choices made by Microsoft.

To assist themselves in eliminating these "unsafe" uses from their own codebase Microsoft have modified their compiler to flag them. This is illustrated by the message shown above. (This was posted to the library working group reflector – I presume that somewhere in the part of the error message I've not seen it identifies the code that uses `swap_ranges`.) If carelessly written this code could lead to memory being overwritten.

Microsoft have also developed some "safe" alternatives to the standard functions they have "deprecated" – and this alternative is what they have called "The Safe C++ Library". Well, their code is developed in their "kitchen" – so they are perfectly entitled to ban sharp knives there and from their account of their experience it seems that they had good results (although it hasn't been made clear to me how they measured this).

Full of enthusiasm for these benefits that this initiative has achieved, they've decided that all their users will gain from having these facilities made available to them. However, many of the experts working in C++ resent the idea of these rules being imposed upon their kitchens! In these kitchens the sharp knives are there because they are useful and are (hopefully) used with the care they require.

Many of the experts involved in this discussion provide libraries that need to compile cleanly with a wide range of C++ implementations. Some have already announced that they will be disabling this message in their code – they clearly don't want to spend time producing a special version of their code for clients that happen to use the Microsoft compiler.

There are also concerns for the ordinary C++ developer. Like many others I require clean compiles, and while a few of us might recognise that the above message is misleading I've certainly worked for organisations where an extended debate would ensue about how to adapt to it. (Which is a waste of time and energy.) And there will be some organisations or developers that will blithely follow the advice to use "The Safe C++ Library" without realising that doing so locks their codebase into a library that is only available from a single vendor.

The Microsoft representatives seem to be surprised at the negative reaction of the other experts to their plans. They had come up with a way to improve the quality of their customer's code and had not foreseen the possibility that people would not want to have these "unsafe" uses of C++ highlighted or to be offered a safe alternative. They really hadn't considered how their efforts would be perceived by the remainder of the C++ community.

Now there are many things that Microsoft could have done differently if only they had realised the need. And, after the feedback they have received, they may indeed do things differently when they ship the next version! They could have avoided terminology that suggests that their coding guidelines are connected with the C++ standard, they need not have made these messages the default option, and they could have provided better mechanisms for controlling these messages. (In the current version if a library vendor disables them for the duration of a header from their library, then the messages are not always emitted for users that choose to adopt Microsoft's guidelines.)

A more generous option would be to ensure that their "Safe Standard C++ Library" is made widely available – preferably under a licence that allows it to be used and supported on other platforms. If it brings the benefits to others that Microsoft have experienced then it could be of real benefit to developers. I don't know how common they are in the wider community but there is certainly a class of buffer-overrun errors addressed by these efforts. If they are as common as Microsoft believes, it would be a shame if these warnings are ignored (or simply disabled) by developers as "yet another attempt to achieve vendor lock-in" – but that has been precisely the reaction of those developers I've consulted about this.

The "Safe Standard C++ Library" might even form the basis for future revisions to the C++ standard. The Microsoft representatives have indicated that the parts of this work applicable to the C standard have already been adopted by the ISO C working group as the basis for a "Technical Report" (due next year) and that "once there is a little more community experience" Microsoft intends to do the same with the work on C++. So, in a future revision of C++, using `swap_ranges` with pointer arguments (which Microsoft considers an unsafe practice) may indeed become deprecated!

From what I've seen on TV every chef thinks the way that they run their kitchen is the right way – and that everyone else can gain by emulating them. So it is not too surprising that the developers at Microsoft think the same way. On the other hand the resistance to new ideas cannot be absolute – otherwise we'd still be using wooden implements to cook over open fires. Chefs (and developers) are impressed by results – and if the results of using these tools are good enough they will be adopted.

And what will I be doing in "my kitchen"? Well, I see C++ developers writing needlessly platform specific code far more often than I see them misusing `swap_ranges` (and I don't think I've seen `gets` used since the mid-'80s). So I'll be turning that warning firmly to the "off" setting.

*Alan Griffiths*
overload@accu.org

# Letter to the Editor

## Adding Stakeholder Metrics to Agile Projects (Article in Overload 68)

**From: Anthony Williams**

Tom, Alan,

Whilst I found this article interesting, and accept that Tom has found "Evo" to be a useful process for developing software, there are a few comments in the article that indicate a lack of awareness of other agile processes.

In the detail of the process description, item 1 (gather critical goals from stakeholders), the article says *By contrast, the typical agile model focuses on a user/customer 'in the next room'. Good enough if they were the only stakeholder, but disastrous for most real projects.* Most agile processes view the "Customer" as a role, and whoever fills that role has the responsibility of obtaining the key goals from all stakeholders, not just one; this role is often filled by the project manager. Most agile processes also recommend that the Customer is in the same room as the rest of the team, not the next one.

**TG:** *In that case we should call it a 'Stakeholder Representative'. My experience is that:*

1. *People are very incomplete (not just in Agile methods) in defining the many stakeholders, and their requirements*
2. *I would not trust a representative to define their goals clearly, nor would I trust them to give feedback from real evolutionary deliveries, that should be trialled by real stakeholders as far as possible.*
3. *Keep in mind that even on small projects we find there are 20 to 30 or more interesting stakeholders, each with a set of requirements.*

**AW:** I agree with you on the importance of identifying who the real stakeholders are, and ensuring they actively participate in the development process.

Scott Ambler has an article on the subject of Active Stakeholder Participation, at:

`http://www.agilemodeling.com/essays/activeStakeholderParticipation.htm`

**TG:** *Great paper - I heartily agree with him and wish more of these thoughts got into agile culture.*

Item 2 says *Using Evo, a project is initially defined in terms of clearly stated, quantified, critical objectives. Agile methods do not have any such quantification concept.* Again, this flies in the face of my experience with agile methods - most agile processes recommend that requirements are expressed in the form of executable acceptance tests; I cannot think of any more quantified objective than a hard pass/fail from an automated test.

**TG:** *This remark just proves my point! A testable requirement is NOT identical with a quantified requirement. Please study Sample Chapter: How to quantify any quality:*

`http://books.elsevier.com/bookscat/samples/0750665076/0750665076.pdf`

*to understand the concept of quantification.*

**AW:** If my comment proves your point, I guess I didn't explain myself correctly, or I really don't understand where you're coming from.

Mike Cohn says in *User Stories Applied* that stories need to be testable: a bad story is "a user must find the software easy to use", or "a user must never have to wait very long for a screen to appear"; he then goes on to point out that better stories are "novice users are able to complete common workflows without training", and "new screens appear within two seconds in 95% of cases". The first is still about usability, so needs manual testing with real "novice users", but the second can be automated. Agile practices prefer automated tests to manual ones.

Do Mike's examples better reflect your quantification goals?

**TG:** *YES we are getting there, I found his chapter at* `http://www.mountaingoatsoftware.com/articles/usa_sample.pdf` *and he is moving in the right direction. I never saw things like that from any other agile source. I wonder how frequently this is applied in practice? The term 'story' is misleading here since he is really speaking about what others would call requirements. But OK!*

*Thanks for pointing this excellent source out to me I will certainly quote it to others in the agile field! I wrote him an email and copied you. Nice website.*

Though item 5 doesn't explicitly say anything about what agile processes do or don't do, the implication is that they don't do what is recommended here. Popular agile processes recommend that the person or people in the Customer role writes the acceptance tests; it is their responsibility to ensure that what they specify represents the needs of the stakeholders.

**TG:** *Craig Larman (Agile and Iterative Development: a Managers Guide) has carefully studied my Evo method and compared it to other Agile methods in his 2003 book.*

*The difference here between what I am trying to say and what a reader might wrongly guess I am saying is probably due to misunderstandings. A good picture of my Evo method in practice is the Overload 65 paper by Trond Johansen of FIRM. Part of this can deeper arguments can be found by downloading my XP4 talk "WHAT IS MISSING FROM THE CONVENTIONAL AGILE AND eXtreme METHODS? ..." available at* `http://xpday4.xpday.org/slides.php`

**AW:** I've just looked at the slides; I think they explain your point better than the Overload article.

Finally, the Summary says "A number of agile methods have appeared ... They have all missed two central, fundamental points; namely quantification and feedback". Quite the contrary: these are fundamental principles of agile methods.

**TG:** *the summary must be interpreted in the light of the preceding detail. I agree there is some feedback and some quantification in agile methods, but my point was there is no quantification of primary quality and performance objectives (merely of time and story burn rates). The main reasons for a project are NOT quantified in any of the other Agile methods. Consequently the feedback is not based on measurement of step results compared to estimated effect (see above FIRM example to see how this works). I do apologize that my paper might not have been long enough or detailed enough to make their points clear to people from a non quantified quality culture (most programmers). But I hope the above resources help to clarify.*

The relentless testing at both unit level (TDD) and acceptance test level is designed to get fast feedback as to the state of the project, both in terms of regressions, and in terms of progress against the goals.

**TG:** *I agree, but this is not the quantified feedback I was referring to.*

The iterative cycle is designed to allow fast feedback from the stakeholders as to whether the software is doing what they want. It also provides feedback on the rate of development, and how much progress is being made, which allows for realistic estimates of project completion to be made.

**TG:** *There is no quantified feedback, in conventional agile methods, as to progress towards the quantified goals. For example FIRM set a release 9.5 target of 80% intuitiveness and met it. Such concepts are entirely alien to the rest of the agile world. They cannot even conceive of quantification of such usability attributes. The rate of story burn is a very crude measure of progress. But the actual impact on most quality levels is impossible to say anything about from a story burn rate.*

**AW:** I have no idea what it can possibly mean for delete "to" for something to have "80% intuitiveness", so I guess you're right that "they cannot even conceive of quantification of such usability attributes". Could you explain in more detail?

Story burn rates give you real measures of progress in terms of money spent, features completed, and estimated project completion time. If you want to know details about how close you are to specific goals, then you need to look at **what** the features are that have been completed.

Automated acceptance tests provide the "quantification" Tom seeks: the stakeholders specify desired results, and the acceptance tests tell them whether the software meets their targets. These don't have to be functional tests, they can be non-functional too - if it is important that the time for a given operation is less than a specified limit, the team is encouraged to write a test for that. Likewise, if it is important that a certain number of simultaneous users are allowed, or the system must handle a certain amount of data, then there should be tests for those things, too.

**TG:** *Well this sounds like moving in the right direction. But it is not anywhere near enough, for real systems, and not done at all as far as I have seen, and not instructed in the agile textbooks. Automated acceptance tests DO NOT by any stretch of the imagination provide the quantification I seek! Nowhere near!*

**AW:** I agree that this is not necessarily focused on in the agile textbooks. They generally skip over it by saying that the acceptance tests define what the application should do. It is up to the team to realize that if the performance matters, then they need a test for that; if the usability matters, they need a test for that, etc.

However, agile processes are not just defined by the textbooks. They are defined by the community of people who use them, and are continually evolving; there's recently been discussion of performance testing on the agile-testing mailing list, for example, and there is a whole mailing list dedicated to the discussion of usability, and how to achieve it with agile methods.

**TG:** *Show me a real case study that like FIRM tracks 25 simultaneous quality attributes over 10 evo steps to a single release cycle, and in addition spends one week a month working towards about 20 additional internal stakeholder measures of quality (I am happy to supply illustrations for the latter, they are recent).*

**AW:** Examples of the tracked quality attributes would be useful. Just to recap, whilst I don't disagree with Tom's recommendations, he does other agile methods an injustice in suggesting that these are new recommendations: much of what he suggests is already a key part of agile methods.

***Anthony***

**TG:** *I maintain they are new, if you understand what I mean by quantification. I do apologize if I have done conventional agile methods an injustice. I have no such intent. But of course I cannot know undocumented private practices! I have made these assertions to several agile conferences and not one participant, who heard and saw what I said suggested that I had misrepresented the current state of agile methods. I think the main problem here is things like believing that testing of the conventional kind constitutes measurement of quantified qualities. I hope that a study of the above links will help clarify this. My apologies if I have not in my paper been able to convey what I intended immediately to some readers. I hope this extensive reply to Anthony will enable readers to progress their understanding of my intent.*

*Best wishes.*

***Tom***

**AW:** I now feel that I understand what you were getting at. Most agile methodologies are very generic in their descriptions, and just identify that the customer representative should write acceptance tests for the stories, with no explicit focus on what the tests should look like, or what should be tested. Your key recommendation, as I understand it, is that when writing stories and acceptance tests, then the customer should focus on the benefits the stakeholders are expecting to see, and identify strict quantifiable measurements for these benefits, which can then be used for evaluating the software, even if these benefits are not obviously quantifiable, like "improved usability".

If I have understood correctly, then I think this is indeed an important focus, and not something that the descriptions of most agile methods focus on in any great depth. As a reader, I would appreciate an article explaining how to achieve such focus, and how to quantify such things as "ease of use" and "intuitiveness".

**TG:** *See CE chapter 5[1], but here are some other:* (*See "How to Quantify Quality" in this Overload -AG*)

**AW:** Primarily, my reaction was to the tone of your Overload article, which struck me as very adversarial - EVO good, other agile methods bad. From my point of view, everything you have suggested falls on the shoulders of the agile Customer; it is their responsibility to identify the stakeholders and their needs, and write acceptance tests that reflect those needs. If your article had started with this in mind, and presented the same information as a way of assisting the agile Customer in fulfilling their responsibilities, it would have come across better to me; I felt that the slide from your XPDay presentation did a better job of this than the Overload article.

**TG:** *It is of course the responsibility of the customer and stakeholders to articulate what they want. But they are not professionally trained to do it well and they usually do it badly. So one possibility is that 'we' train ourselves to help them articulate what they really want, and not expect too much clarity from them. But we do expect them to know their business and be able to have a dialogue leading to clarity with professional assistance.*

**AW:** Again, thank you for taking the time to respond to my comments. I have found the discussion most enlightening.

**TG:** *Me too thanks Anthony, hope to meet you someday soon, maybe at XP5?*

1 CE = "Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering using Planguage" by Tom Gild published by Elsevier.

# The Curate's Wobbly Desk

by Phil Bass

## The Vicar's Lodger

The story of the curate's egg is well known [1], but I bet you've never heard about the curate's wobbly desk.

When Aubrey Jones was first ordained he was appointed curate to St Michael's church, Belton Braces. It was a small parish serving just a few quiet hamlets and there would have been nowhere for Mr. Jones to stay if he had not been offered lodgings at the vicarage. The vicar, the Reverend Cuthbert Montague-Smith, was a large and imposing man with strong opinions on how to do God's bidding. The timid Aubrey Jones found him rather... well, intimidating.

The bishop had suggested that Aubrey would benefit from studying the early history of the Christian church and the vicar expressed a hope that this research would turn up some interesting material for the parish magazine. Eager to please, Aubrey went out and bought a cheap, mass-produced, self-assembly desk to work at. The D.I.Y. shop was running a special offer – free delivery – and the curate felt he was getting a bargain.

## Secret Drawers

Reading through the assembly instructions Aubrey was delighted to find that the desk contained a secret drawer. He had fond notions of keeping a diary and one day, perhaps, writing a book on the life of an English pastor based on his own ministry and experiences. But he suspected that Cuthbert would regard such activities as of little practical value and it would be better for the vicar to remain ignorant of those particular jottings.

As chance would have it, the vicar's desk also had a secret drawer. I don't know what Cuthbert kept in that drawer, or even if he knew it was there. But I do know that both Aubrey's and Cuthbert's secret drawers were operated by two catches. If you press both catches at the same time the drawer slides out. I remembered the vicar, the curate and the secret drawers when I was working on the software for a computer game. (It would have been called "Murder at the Vicarage", but Agatha Christie got there first.) "Let's use that in the game", I thought, and wrote an interface class representing the secret drawer based on classes from my Event/Callback library [2]. Listing 1 shows the Event class template used for the drawer release mechanism and the `Callback::Function` class template used for the catches.

The `Drawer` class interface is shown in Listing 2. The key feature here is that `Digital_Input` is an abstract base class with a pure virtual function call operator. When catch A's `Digital_Input` callback is invoked the lock/unlock state for catch A must be updated, the state of catch B must be read and, if both catches are in the unlock position, the drawer release mechanism must be activated by triggering the `Digital_Output` event. The overall effect is that the `Drawer` class behaves like an And gate in an electronic circuit.

In the game, a `Drawer` object is created and GUI push-button widgets are attached to the drawer's inputs (the catches). The drawer itself is represented by an `Image` widget which shows either a closed drawer or an open drawer. A callback that toggles between the open and closed images is attached to the drawer's output (the drawer release mechanism). The player has to find a way of pressing both of the catches at the same time to open the drawer – not easy using just a mouse. A rough sketch of the client code is shown in Listing 3.

The rest of this paper describes two ways of implementing the

```
// Event classes.
template< typename Arg >
struct Event : std::list< Callback::Function<Arg>* >
{
  typedef Arg                       argument_type;
  typedef Callback::Function<Arg> callback_type;
  void notify( Arg arg )
  {
    std::for_each( this->begin(), this->end(),
       bind_2nd(
          memfun( &callback_type::operator()),
          arg ));
  }
};
// Callback function base classes.
namespace Callback
{
  template< typename Arg >
  struct Function
  {
    typedef Arg argument_type;
    virtual ~Function() {}
    virtual void operator() ( argument_type ) = 0;
  };
}
```

**Listing 1 - the Event and Callback::Function classes.**

```
// A value type with just two values.
enum Digital_Value { off, on };

// Input and output types.
typedef Callback::Function< Digital_Value >
   Digital_Input;
typedef Event< Digital_Value > Digital_Output;

// A secret drawer operated by two catches.
class Drawer
{
public:
    Drawer();
    Digital_Input& catch_A();  // catch A
    Digital_Input& catch_B();  // catch B
    Digital_Output& output();  // drawer release
private:
    . . .
};
```

**Listing 2 - Secret drawer interface.**

```
// Internal objects.
Drawer drawer;                          // secret drawer

// User interface.
GUI::Button  catch_A, catch_B;          // buttons to release the drawer
GUI::Image   picture;                   // picture of open/closed drawer
Toggle_Image toggle_image(picture);     // functor that toggles open/closed

// Connect the UI widgets to the internal objects.
catch_A.push_back( &drawer.catch_A() );
catch_B.push_back( &drawer.catch_B() );
drawer.output().push_back( &toggle_picture );

// Run the GUI scheduling loop.
GUI::run();
```

**Listing 3 - Using the Drawer class.**

`Drawer` class with the help of our curate, his vicar and their furniture.

## The Curate's Self-Assembly Desk

As soon as he had a spare half-hour Aubrey Jones opened the box containing his flat-pack desk, carefully laid out the panels, runners, feet, dowels, nuts and bolts, and began to assemble them. He paid particular attention to the secret drawer. The drawer itself had a grain-effect finish that looked remarkably like real wood, but probably wasn't. The release mechanism was an integral part of the drawer, located at the back. The secret catches were separate – metal, with knobs in the same fake wood as the drawer and disguised as decorative features. The catches had to be fastened to the front of the drawer and connected to the release mechanism with two long, thin and worryingly flimsy metal rods.

The structure of my code was very similar, as you can see from the overview in Listing 4 and the function bodies in Listing 5. The drawer release mechanism was represented by `Digital_Value` and `Digital_Output` members of the `Drawer` class. The catches were separate classes (`Catch_A` and `Catch_B`) and they were attached to the `Drawer` class by pointers. With this design the functions in the public interface are trivial and shown here defined within the class declaration.

This design is conceptually simple, but it didn't feel quite right. Like cheap, mass-produced furniture it seemed inelegant and unsatisfying. Did the `Catch` classes really have to store a pointer to their parent object? After all, the address of the `Drawer` object is a fixed offset from each of the `Catch` objects. Couldn't we just subtract that offset from the `Catch` object's this pointer and apply a suitable cast?

After some thought I decided that pointer arithmetic and casting would be worse than the disease I was trying to cure. A case of premature optimisation, and an ugly one at that. I needed to think like the master craftsmen of old. And that reminded me of the vicar's desk.

## The Vicar's Antique Writing Table

Cuthbert Montague-Smith loved his big sturdy old desk. It was reminiscent of the magnificent library writing table at Harewood House, near Leeds [3]. Cuthbert suspected it was built by Thomas Chippendale himself, although he was unable to provide a shred of evidence to support that view.

I don't suppose the great furniture maker would appreciate the finer points of software design in C++, but I tried to imagine the approach he would use. He would surely pay considerable attention to detail and not rest until he had discovered a method that was both elegant and practical.

With this in mind I thought again about the `Drawer` class implementation. The curate's desk design in Listings 4 and 5 contains `Catch` classes that reference an external object (the `Drawer` itself); that is why it needs those inelegant pointers. If we could move the external data into the `Catch` classes the pointers would not be necessary. So the question is, how can we make the `Drawer` state variables part of two separate `Catch` objects?

It's no good putting member variables into the concrete `Catch` classes because that

```
class Drawer
{
public:
  Drawer();
  Digital_Input& catch_A() { return catch_A_input; }
  Digital_Input& catch_B() { return catch_B_input; }
  Digital_Output& output() { return output_event; }
private:
  struct Catch_A : Digital_Input
  {
    Catch_A( Drawer* d ) : value( off ), drawer( d ) {}
    void operator()( Digital_Value );
    Digital_Value value;
    Drawer* const drawer;
  };
  struct Catch_B : Digital_Input
  {
    // . . . same as Catch_A . . .
  };
  void sync();    // set output value from inputs
  Catch_A        catch_A_input;  // catch A's lock/unlock state
  Catch_B        catch_B_input;  // catch B's lock/unlock state
  Digital_Value  output_value;  // drawer release state
  Digital_Output output_event;  // drawer release event
};
```

**Listing 4 - Overview of the Drawer with separate Catch classes.**

```
// Operate Catch A
inline void Drawer::Catch_A::operator()( Digital_Value value )
{
  drawer->catch_A_input.value = value;
  drawer->sync();
}
// Operate Catch B
inline void Drawer::Catch_B::operator()( Digital_Value value )
{
  drawer->catch_B_input.value = value;
  drawer->sync();
}
// Create the Drawer
inline Drawer::Drawer()
  : catch_A_input( this ), catch_B_input( this ), output_value( off )
{}
// Set and publish the state of the drawer release mechanism
inline void Drawer::sync()
{
  output_value =
     Digital_Value( catch_A_input.value & catch_B_input.value );
  output_event.notify( output_value );
}
```

**Listing 5 - Implementation of the Drawer with separate Catch classes.**

would just duplicate the data; and we can't put data into the `Digital_Input` class because that would compromise the Event/Callback library. The only option is to put them in a shared base class. The key to the desk is virtual inheritance.[1] Listing 6 and Listing 7 show how I chose to use this technique.

All the variables have been moved to the private nested class, `Data`. As this is an implementation detail of the `Drawer` class I have not bothered to create separate interface and implementation sections for `Data`. Purists can add a private section and suitable access functions if they wish. It is appropriate, however, to provide `Data` with a constructor and a function that sets the drawer release mechanism's state from the `Catch` values to avoid duplicating these operations in both the `Catch` classes.

The `Catch` classes themselves use virtual inheritance to "import" the shared `Data` object. They also provide a function that updates their own lock/unlock state, calculates the drawer release state and publishes the drawer release state to the client code.

The `Drawer` class could inherit directly from the `Catch` classes, but

```
class Drawer
{
public:
  . . .
private:
  struct Data
  {
    Data();
    void sync();
    Digital_Value  catch_A_value,
                   catch_B_value;
    Digital_Value  output_value;
    Digital_Output output_event;
  };
  struct Catch_A : Digital_Input, virtual Data
  {
    void operator()( Digital_Value );
  };
  struct Catch_B : Digital_Input, virtual Data
  {
    void operator()( Digital_Value );
  };
  struct Body : Catch_A, Catch_B {} body;
};
```

**Listing 6 - Overview of the Drawer using virtual inheritance.**

```
// Operate Catch A
inline void Drawer::Catch_A::operator()(
   Digital_Value value )
{
  catch_A_value = value;
  sync();
}
// Operate Catch B
inline void Drawer::Catch_B::operator()(
   Digital_Value value )
{
  catch_B_value = value;
  sync();
}
// Initialise the Drawer's state
inline Drawer::Data::Data()
   : catch_A_value( off ),
   catch_B_value( off ),
   output_value( off )
{}
// Set and publish the state of the drawer
// release mechanism
inline void Drawer::Data::sync()
{
  output_value = Digital_Value(
     catch_A_value & catch_B_value );
  output_event.notify( output_value );
}
```

**Listing 7 - Implementation of the Drawer using virtual inheritance.**

1 I claim poetic licence for bending the rules of English grammar here.

[concluded at foot of next page]

# Better Encapsulation for the Curiously Recurring Template Pattern

by Alexander Nasonov

C++ has a long, outstanding history of tricks and idioms. One of the oldest is the curiously recurring template pattern (CRTP) identified by James Coplien in 1995 [1]. Since then, CRTP has been popularized and is used in many libraries, particularly in Boost [3]. For example, you can find it in `Boost.Iterator`, `Boost.Python` or in `Boost.Serialization` libraries.

In this article I assume that a reader is already familiar with CRTP. If you would like to refresh your memory, I would recommend reading chapter 17 in [2]. This chapter is available for free on `www.informit.com`.

If you look at the curiously recurring template pattern from an OO perspective you'll notice that it shares common properties with OO frameworks (e.g. Microsoft Foundation Classes) where base class member functions call virtual functions implemented in derived classes. The following code snippet demonstrates OO framework style in its simplest form:

```
// Library code
class Base
{
  public:
    virtual ~Base();
    int foo() { return this->do_foo(); }

  protected:
    virtual int do_foo() = 0;
};
```

Here, `Base::foo` calls virtual function `do_foo`, which is declared as a pure virtual function in Base and, therefore, it must be implemented in derived classes. Indeed, a body of `do_foo` appears in class `Derived`:

```
// User code
class Derived : public Base
{
  private:
    virtual int do_foo() { return 0; }
};
```

What is interesting here, is that an access specifier of `do_foo` has been changed from protected to private. It's perfectly legal in C++ and it takes a second to type one simple word. What is more, it's done intentionally to emphasize that `do_foo` isn't for public use. (A user may go further and hide the whole `Derived` class if she thinks it's worth it.)

The moral of the story is that a user should be able to hide implementation details of the class easily.

Now let us assume that restrictions imposed by virtual functions are not affordable and the framework author decided to apply CRTP:

```
// Library code
template<class DerivedT>
class Base
{
  public:
    DerivedT& derived() {
       return static_cast<DerivedT&>(*this); }
    int foo() {
       return this->derived().do_foo(); }
};
// User code
class Derived : public Base<Derived>
{
  public:
    int do_foo() { return 0; }
};
```

Although `do_foo` is an implementation detail, it's accessible from everywhere. Why not make it private or protected? You'll

---

[continued from previous page]

that would mean exposing the `Data` class and the concrete `Catch` classes to its clients. Instead, I have chosen to write a nested `Body` class that completes the virtual inheritance diamond and then store a `Body` member within the `Drawer` class. That way, none of the classes used in the `Drawer` implementation pollute the namespaces used in the client code.

The `Catch` class member functions in the vicar's desk design are slightly simpler than those in the curate's version. Moving the data to a base class enables them to access the variables directly instead of via those ugly and unnecessary pointers. Initialisation of the data members is also slightly simpler because there are no pointer values to set (which means one less opportunity for a bug). And the `sync()` function is the same in both designs. Chippendale would have been proud.

## It All Falls Apart

The curate struggled to assemble his desk. The instructions seemed to have been translated from a foreign language (badly), the diagrams didn't seem to match the parts he had and Aubrey Jones' mind wasn't good at 3D visualisation. The release mechanism for the secret drawer, in particular, baffled the poor man. Eventually, the desk was completed and moved into position under the window of Aubrey's bed sitting room where he could look out over the garden.

The desk was always a bit wobbly and the secret drawer never did work. (The connecting rods were not installed correctly, so the drawer release would not activate.) Aubrey never quite found the time to research the early history of the Christian church and with no place to hide his private writings he soon lost the urge to keep a diary. Indeed, after a few years as the vicar of a provincial parish on the borders of London and Essex his ecclesiastical career took a turn for the worse [4]. He may yet write a book. If he does, it will be about temptation, greed and the frailty of man, but sadly it will not be about a life of service to the church.

*Phil Bass*
phil@stoneymanor.demon.co.uk

## References

1 See `http://www.worldwidewords.org/qa/qa-cur1.htm`, for example.
2 Phil Bass, "Evolution of the Observer Pattern", *Overload 64*, December 2004.
3 `http://www.harewood.org/chippendale/index2.htm`.
4 `http://www.lyricsfreak.com/g/genesis/58843.html`

find an answer inside function `foo`. As you see, the function calls `Derived::do_foo`. In other words, base class calls a function defined in a derived class directly.

Now, let's find an easiest way for a user to hide implementation details of `Derived`. It should be very easy; otherwise, users won't use it. It can be a bit trickier for the author of `Base` but it still should be easy to follow.

The most obvious way of achieving this is to establish a friendship between `Base` and `Derived`:

```
// User code
class Derived : public Base<Derived>
{
  private:
    friend class Base<Derived>;
    int do_foo() { return 0; }
};
```

This solution is not perfect for one simple reason: the `friend` declaration is proportional to the number of template parameters of `Base` class template. It might get quite long if you add more parameters.

To get rid of this problem one can fix the length of the `friend` declaration by introducing a non-template `Accessor` that forwards calls:

```
// Library code
class Accessor
{
  private:
    template<class> friend class Base;
    template<class DerivedT>
    static int foo(DerivedT& derived)
    {
        return derived.do_foo();
    }
};
```

The function `Base::foo` should call `Accessor::foo` which in turn calls `Derived::do_foo`. A first step of this call chain is always successful because the `Base` is a `friend` of `Accessor`:

```
// Library code
template<class DerivedT>
class Base
{
  public:
    DerivedT& derived() {
       return static_cast<DerivedT&>(*this); }
    int foo() {
       return Accessor::foo(this->derived()); }
};
```

The second step succeeds only if either `do_foo` is public or if the `Accessor` is a `friend` of `Derived` and `do_foo` is protected. We are interested only in a second alternative:

```
// User code
class Derived : public Base<Derived>
{
  private:
    friend class Accessor;
    int do_foo() { return 0; }
};
```

This approach is taken by several boost libraries. For example, `def_visitor_access` in Boost.Python and `iterator_core_access` in Boost.Iterator should be declared as friends in order to access user-defined private functions from `def_visitor` and `iterator_facade` respectively.

Even though this solution is simple, there is a way to omit the `friend` declaration. This is not possible if `do_foo` is private – you will have to change that to protected. The difference between these two access specifiers is not so important for most CRTP uses. To understand why, take a look at how you derive from CRTP base class:

```
class Derived : public Base<Derived> { /* ... */ };
```

Here, you pass the final class to `Base`'s template arguments list.

An attempt to derive from `Derived` doesn't give you any advantage because the `Base<Derived>` class knows only about `Derived`.[1]

Our goal is to access protected function `Derived::do_foo` from the `Base`:

```
// User code
class Derived : public Base<Derived>
{
  protected:
    // No friend declaration here!
    int do_foo() { return 0; }
};
```

Normally, you access a protected function declared in a base class from its child. The challenge is to access it the other way around.

The first step is obvious. The only place for our interception point where a protected function can be accessed is a descendant of `Derived`:

```
struct BreakProtection : Derived
{
    static int foo(Derived& derived) {
       /* call do_foo here */ }
};
```

An attempt to write

```
return derived.do_foo();
```

inside `BreakProtection::foo` fails because it's forbidden according to the standard, paragraph 11.5:

> *When a friend or a member function of a derived class references a protected nonstatic member of a base class, an access check applies in addition to those described earlier in clause 11. Except when forming a pointer to member (5.3.1), the access must be through a pointer to, reference to, or object of the derived class itself (or any class derived from that class) (5.2.5).*

The function can only be accessed through an object of type `BreakProtection`.

Well, if the function can't be called directly, let's call it indirectly. Taking an address of `do_foo` is legal inside `BreakProtection` class:

```
&BreakProtection::do_foo;
```

There is no `do_foo` inside `BreakProtection`, therefore, this expression is resolved as `&Derived::do_foo`. Public access to a pointer to protected member function has been granted! It's time to call it:

```
struct BreakProtection : Derived
{
  static int foo(Derived& derived)
  {
    int (Derived::*fn)() =
       &BreakProtection::do_foo;
    return (derived.*fn)();
  }
};
```

[concluded at foot of next page]

# How to Quantify Quality: Finding Scales of Measure

by Tom Gilb

**Abstract.** 'Scales of measure' are fundamental to the definition of all scalar system attributes; that is, to all the performance attributes (such as reliability, usability and adaptability), and to all the resource attributes (such as financial budget and time). A defined scale of measure, allows you to numerically quantify such attributes.

'Scales of measure' form a central part of Planguage, a specification language and set of methods, which I have developed over many years.

This paper describes how you can develop your own *tailored* scales of measure for the specific system attributes, which are important to your organization or system. You cannot rely on being 'given the answer' about how to quantify. You will lose control over your current vital system performance concerns if you cannot, or do not, quantify your critical attributes.

## Scales of Measure and Meters

Scales of measure (Scales) are essential to *quantify* system attributes. A Scale specifies an operational definition of 'what' is being measured and it states the units of measure. All estimates or measurements are made with reference to the Scale.

The practical ability to *measure* where you are on a Scale (that is to be able to establish the numeric level) is also important. A Meter (sometimes known as a 'Test') is a practical method for measuring. A Scale can have several Meters.

Tag: <assign a tag name to this Scale>.
Version: <date of the latest version or change>.
Owner: <role/email of who is responsible for updates/changes>.
Status: <Draft, SQC Exited, Approved>.
Scale: <specify the Scale with defined [qualifiers].>.
Alternative Scales: <reference by tag or define other Scales of interest as alternatives and supplements>.
Qualifier Definitions: <define the scale qualifiers, like 'for defined [Staff]', and list the options, like {CEO, Finance Manager, Customer}.>.
Meter Options: <suggest Meter(s) appropriate to the Scale>.
Known Usage: <reference projects & specifications where this Scale was actually used in practice with designers' names>.
Known Problems: <list known or perceived problems with this Scale>.
Limitations: <list known or perceived limitations with this Scale>.

**Figure 1: Draft template**

## Finding and Developing Scales of Measure and Meters

The basic advice for identifying and developing scales of measure (Scales) and meters (Meters) for scalar attributes is as follows:

1. Try to re-use previously defined Scales and Meters.
2. Try to modify previously defined Scales and Meters.

---

[continued from previous page]

For better encapsulation, the `BreakProtection` can be moved to the private section of `Base` class template. The final solution is:

```
// Library code
template<class DerivedT>
class Base
{
  private:
    struct accessor : DerivedT
    {
        static int foo(DerivedT& derived)
        {
            int (DerivedT::*fn)()
               = &accessor::do_foo;
            return (derived.*fn)();
        }
    };
  public:
    DerivedT& derived() {
       return static_cast<DerivedT&>(*this); }
    int foo() { return accessor::foo(
       this->derived()); }
};
// User code
struct Derived : Base<Derived>
  protected:
    int do_foo() { return 1; }
};
```

Note that the user code is slightly shorter and cleaner than in the first solution. The library code has similar complexity.

There is a downside to this approach, though. Many compilers don't optimize away function pointer indirection even if it's called in-place:

```
return (derived.*(&accessor::do_foo))();
```

The main strength of CRTP over virtual functions is better optimization.

CRTP is faster because there is no virtual function call overhead and it compiles to smaller code because no type information is generated. The former is doubtful for the second solution while the latter still holds. Hopefully, future versions of popular compilers will implement this kind of optimization. Also, it's less convenient to use member function pointers, especially for overloaded functions.

*Alexander Nasonov*
alnsn@yandex.ru

## References

[1] James O. Coplien. "Curiously Recurring Template Patterns", *C++ Report*, February 1995.

[2] David Vandevoorde, Nicolai M. Josuttis. "*C++ Templates: The Complete Guide*". http://www.informit.com/articles/article.asp?p=31473

[3] Boost libraries. http://www.boost.org.

[4] ISO-IEC 14882:1998(E),Programming languages - C++.

Tag: Ease of Access.
Version: August 11, 2003.
Owner: Rating Model Project (Bill).
Scale: Speed for a defined [Employee Type] with defined [Experience] to get a defined [Client Type] operating successfully from the moment of a decision to use the application.
Alternative Scales: None known yet.
Qualifier Definitions:
Employee Type: {Credit Analyst, Investment Banker, …}.
Experience: {Never, Occasional, Frequent, Recent}.
Client Type: {Major, Frequent, Minor, Infrequent}.
Meter Options: EATT: Ease of Access Test Trial. "This tests all frequent combinations of qualifiers at least twice. Measure speed for the combinations."
Known Usage: Project Capital Investment Proposals [2001, London].
Known Problems: None recorded yet.
Limitations: None recorded yet.

**Figure 2: Use of the template**

3. If no existing Scale or Meter can be reused or modified, use common sense to develop innovative, homegrown quantification ideas.
4. Whatever Scale or Meter you start off with, you must be prepared to learn. Obtain and use early feedback, from colleagues and from field tests, to redefine and improve your Scales and Meters.

## Reference Library for Scales of Measure

'Reuse' is an important concept for, sharing experience and saving time when developing Scales. You need to build reference libraries of your 'standard' scales of measure. Remember to maintain details supporting each 'standard' Scale, such as Source, Owner, Status and Version (Date). If the name of a Scale's designer is also kept, you can probably contact them for assistance and ideas.

Figure 1 is a draft template with <hints>, for specification of scales of measure in a reference library. Figure 2 is an example of the use of this template.

## Reference Library for Meters

Another important standards library to maintain is a library of 'Meters.' 'Off the shelf' Meters from standard reference libraries can save time and effort since they are already developed and are more or less 'tried and tested' in the field.

It is natural to reference suggested Meters within definitions of specific scales of measure (as in the template and example above). Scales and Meters belong intimately together.

## Managing 'What' You Measure

It is a well-known paradigm that you can manage what you can measure. If you want to achieve something in practice, then quantification, and later measurement, are essential first steps for making sure you get it. If you do not make critical performance attributes measurable, then it is likely to be less motivating for people to find ways to deliver necessary performance levels. They have no clear targets to work towards, and there are no precise criteria for judgment of failure or success.

## Practical Example: Scale Definition

'User-friendly' is a popular term. Can you specify a scale of measure for it?

Here is my advice on how to tackle developing a definition for this quality.

1. If we assume there is no 'off-the-shelf' definition that could be used, then you need to start describing the various aspects of the quality that are of interest.

   There are always many distinct dimensions to qualities such as usability, maintainability, security, adaptability and their like [Gilb 2003]. (Suggestion: Try listing about 5 to 15 aspects of some selected quality that is critical to your project.)

   For this example, let's select 'environmentally friendly' as the one of many aspects that we are interested in, and we shall work on this below.
2. Invent and specify a Tag: 'Environmentally Friendly' is sufficiently descriptive. Ideally, it could be shorter, but it is very descriptive left as it is. We indicate a formally defined concept by capitalizing the tag.

   Tag: Environmentally Friendly.

   Note, we usually don't explicitly specify 'Tag: ' but this sometimes makes the tag identity clearer.
3. Check there is an Ambition statement, which briefly describes the level of requirement ambition. 'Ambition' is one of the defined Planguage parameters.

   Ambition: A high degree of protection, compared to competitors, over the short-term and the long-term, in near and remote environments for health and safety of living things.
4. Ensure there is general agreement by all the involved parties with the Ambition definition. If not, ask for suggestions for modifications or additions to it. Here is a simple improvement to my initial Ambition statement. It actually introduces a 'constraint'.

   Ambition: A high degree of protection, compared to competitors, over the short-term and the long-term, in near and remote environments for health and safety of living things, **which does not reduce the protection already present in nature**.
5. Using the Ambition description, define an initial Scale that is somehow quantifiable (meaning – you can meaningfully attach a number to it). Consider what will be sensed by the stakeholders if the level of quality changes. What would be a visible effect if the quality improved? My initial, unfinished attempt, at finding a suitable Scale captured the ideas of change occurring, and of things getting better or worse:

   Scale: The percentage (%) change in positive (good environment) or negative directions for defined [Environmental Changes].

   However, I was not happy with it, so I made a second attempt. I refined the Scale by expanding it to include the ideas of specific things being effected in specific places over given times:

Scale: The percentage (%) destruction or reduction of defined [Thing] in defined [Place] during a defined [Time Period] as caused by defined [Environmental Changes].

This felt better. In practice, I have added more [qualifiers] into the Scale, to indicate the variables that must be defined by specific things, places and time periods whenever the Scale is used.

6. Determine if the term needs to be defined with several different scales of measure, or whether one like this, with general parameters, will do. Has the Ambition been adequately captured? To determine what's best, you should list some of the possible sub-components of the term (that is, what can it be broken down into, in detail?). For example:

Thing: {Air, Water, Plant, Animal}.
Place: {Personal, Home, Community, Planet}.

This example means: 'Thing' is defined as the set of things: Air, Water, Plant and Animal (which, since they are all four capitalized, are themselves defined elsewhere).

Or alternatively, instead of just the colon after the tag, '=' or the more explicit Planguage parameter, 'Consists Of' can be used to make this notation more immediately intelligible to novices in reading Planguage:

Thing: = {Air, Water, Plant, Animal}.
Place: Consists of {Personal, Home, Community, Planet}.

Then consider whether your defined Scale enables the performance levels for these sub-components to be expressed. You may have overlooked an opportunity, and may want to add one or more qualifiers to that Scale. For example, we could potentially add the scale qualifiers '.... *under defined [Environmental Conditions] in defined [Countries]...*' to make the scale definition even more explicit and more general.

Scale qualifiers (like ...'*defined [Place]*'...) have the following advantages:

- they add clarity to the specifications
- they make the Scales themselves more reusable in other projects

Environmentally Friendly:
Ambition: A high degree of protection, compared to competitors, over the short-term and the long-term, in near and remote environments for health and safety of living things, which does not reduce the protection already present in nature.
----Some scales of measure candidates – they can be used as a complementary set ---
Air: Scale: % of days annually when <air> is <fit for all humans to breath>.
Water: Scale: % change in water pollution degree as defined by UN Standard 1026.
Earth: Scale: Grams per kilo of toxic content.
Predators: Scale: Average number of <free-roaming predators> per square km, per day.
Animals: Scale: The percentage (%) reduction of any defined [Living Creature] who has a defined [Area] as their natural habitat.

**Figure 4: Alternative scales**

- they make the Scale more useful in this project: specific benchmarks, targets and constraints can then be specified for any interesting combination of scale variables (such as, 'Thing = Air').

7. Start working on a Meter – a specification of how we intend to test or measure the performance of a real system with respect to the defined Scale. Remember, you should first check there is not a standard or company reference library Meter that you could use. Try to imagine a practical way to measure things along the Scale, or at least sketch one out. My example is only an initial rough sketch defined by a {set} of three rough measurement concepts. These at least suggest something about the quality and costs with such a measuring process.

   Meter: {scientific data where available, opinion surveys, admitted intuitive guesses}.

   The Meter must always explicitly address a particular Scale specification. It will help confirm your choice of Scale as it will provide evidence that practical measurements can feasibly be obtained on the given scale of measure.

Environmentally Friendly:
Ambition: A high degree of protection, compared to competitors, over the short-term and the long-term, in near and remote environments for health and safety of living things, which does not reduce the protection already present in nature.

Scale: The percentage (%) destruction or reduction of defined [Thing] in defined [Place] during a defined [Time Period] as caused by defined [Environmental Changes].
========== Benchmarks ==========
Past [Time Period = Next Two Years, Place = Local House, Thing = Water]: 20% <- intuitive guess.
Record [Last Year, Cabin Well, Thing = Water]: 0% <- declared reference point.
Trend [Ten to Twenty Years From Now, Local, Thing = Water]: 30% <- intuitive. "Things seem to be getting worse."
========== Scalar Constraint ==========
Fail [End Next Year, Thing = Water, Place = Eritrea]: 0%. "Not get worse."
========== Targets ==========
Wish [Thing = Water, Time = Next Decade, Place = Africa]: <3% <- Pan African Council Policy.

Goal [Time = After Five Years, Place = <our local community>, Thing = Water]: <5%.

**Figure 3: Benchmarks, targets and constraints**

8. Now try out the Scale specification by trying to use it to specify some useful levels on the Scale. Define some reference points from the past (Benchmarks) and some future requirements (Targets and Constraints). See Figure 3, at the bottom of the previous page, for an example.

   If this seems unsatisfactory, then maybe I can find another, more specific, scale of measure? Maybe use a 'set' of different Scales to express the measured concept better? See examples below.
   Here is an example of a single more-specific Scale:

   > Scale: % change in water pollution degree as defined by UN Standard 1026.

   Figure 4 shows an example of some other and more-specific set of Scales for the 'Environmentally Friendly' example. They are perhaps a complimentary set for expressing a complex *Environmentally Friendly* idea.

   Many different scales of measure can be candidates to reflect changes in a single critical factor.

   Environmentally Friendly is now defined as a 'Complex Attribute,' because it consists of a number of 'elementary' attributes: {Air, Water, Earth, Predators, Animals}. A different scale of measure now defines each of these elementary attributes. Using these Scales we can add corresponding Meters, benchmarks (like Past), constraints (like Fail), and target levels (like Goal), to describe exactly how Environmentally Friendly we want to be.

**Level of Specification Detail**. How much detail you need to specify, depends on what you want control over, and how much effort it is worth. The basic paradigm of Planguage is you should only elect to do what pays off for you. You should not build a more detailed specification than is meaningful in terms of your project and economic environment. Planguage tries to give you sufficient power of articulation to control both complex and simple problems. You need to scale up, or down, as appropriate. This is done through common sense, intuition, experience and organizational standards (reflecting experience). But, if in doubt, go into more detail. History says we have tended in the past to specify too little detail about requirements. The result consequently has often been to lose control, which costs a lot more than the extra investment in requirement specification.

## Language Core: Scale Definition

Now let's discuss the specification of Scales in more detail, particularly the use of qualifiers.

**The Central Role of a Scale within Scalar Attribute Definition.** The specified Scale of an elementary scalar attribute is used (re-used!) within all the scalar parameter specifications of the attribute (that is, within all the benchmarks, the constraints and the targets). In other words, a Scale parameter specification is the heart of a specification. Scale is essential to support all the related scalar level parameters: for example Past, Record, Trend, Goal, Budget, Stretch, Wish, Fail and Survival.

Each time a different scalar level parameter is specified, the Scale specification dictates what has to be defined numerically and in terms of Scale Qualifiers (like 'Staff = Financial Manager'). And then later, each time a scalar level parameter definition is read, the Scale specification itself has to be referenced to 'interpret' the meaning of the corresponding scale level specification. So the Scale is truly central to a scalar definition. For example, 'Goal [Staff = Financial Manager]: 23%.' only has meaning in the context of the corresponding scale: for example 'Scale: % of defined [Staff] attending the meeting', Well-defined scales of measure are well worth the small investment to define them, to refine them, and to re-use them.

**Specifying Scales using Qualifiers.** The scalar attributes (performance and resource) are best measured in terms of specific times, places and events. If we fail to do this, they lose meaning. People wrongly guess other times, places and events than you intend, and cannot relate their experiences and knowledge to your numbers. If we don't get more specific by using qualifiers, then performance and resource continues to be a vague concept, and there is ambiguity (which times? which places? which events?).

Further, it is important that the set of different performance and resource levels for different specific time, places and events are identified. It is likely that the levels of the performance and resource requirements will differ across the system depending on such things as time, location, role and system component.

> Decomposing complex performance and resource ideas, and finding market-segmenting qualifiers for differing target levels is a key method of competing for business.

**Embedded Qualifiers within a Scale.** A Scale specification can set up useful 'scale qualifiers' by declaring embedded scale qualifiers, using the format 'defined [<qualifier>]'.

It can also declare default qualifier values that apply by default if not overridden, 'defined [<qualifier>: default: <User-defined Variable or numeric value>]'. For example, […default: Novice].

**Additional Qualifiers.** However, embedded qualifiers should not stop you adding any other useful additional qualifiers later, as needed, during scale-related specification (such as Goal or Meter). But, if you do find you are adding the same type of parameters in almost all related specifications, then you might as well design the Scale to include those qualifiers. A Scale should be built to ensure that it forces the user to define the critical information needed to understand and control a critical performance or resource attribute. This implies that scale qualifiers serve as a checklist of good practice in defining scalar level specifications, such as Past and Goal.

Here is an example of how locally defined qualifiers (see the Goal specification below) can make a quality specification more specific. In this example we are going to see how a requirement can be conditional upon an event. If the event is not true, the requirement does not apply.
First, some *basic definitions* are required (Note that 'Basis', 'Source' and 'State' are Planguage parameters):

> Assumption A: Basis [This Financial Year]: Norway is still not a full member of the European Union.
> EU Trade: Source: Euro Union Report "EU Trade in Decade 2000-2009".
> Positive Trade Balance: State [Next Financial Year]: Norwegian Net Foreign Trade Balance has Positive Total to Date.

Now we apply those definitions below:

> Quality A:
> Type: Quality Requirement.
> Scale: The percentage (%) by value of Goods delivered that are returned for repair or replacement by consumers.
> Meter [Development]: Weekly samples of 10,
> [Acceptance]: 30 day sampling at 10% of representative cases,
> [Maintenance]: Daily sample of largest cost case.
> Fail [European Union, Assumption A]: 40% <- European Economic Members.
> Goal [EU and EEU members, Positive Trade Balance]: 50% <- EU Trade.

The Fail and the Goal requirements are now defined partly with the help of qualifiers. The Goal to achieve 50% (or more, is implied) is only a valid plan if 'Positive Trade Balance' is true. The Fail level requirement of 40% (or worse, less, is implied) is only valid if 'Assumption A' is true. All qualifier conditions must be true for the level to be valid.

## Principles: Scale Specification

1. **The Principle of 'Defining a Scale of Measure'**
   If you can't define a scale of measure, then the goal is out of control.
   *Specifying any critical variable starts with defining its units of measure.*
2. **The Principle of 'Quantification being Mandatory for Control'**
   If you can't quantify it, you can't control it.[1]
   *If you cannot put numbers on your critical system variables, then you cannot expect to communicate about them, or to control them.*
3. **The Principle of 'Scales should control the Stakeholder Requirements'**
   Don't choose the easy Scale, choose the powerful Scale.
   Select scales of measure that give you the most direct control over the critical stakeholder requirements. Chose the Scales that lead to useful results.
4. **The Principle of 'Copycats Cumulate Wisdom'**
   Don't reinvent Scales anew each time – store the wisdom of other Scales for reuse.
   *Most scales of measure you will need, will be found somewhere in the literature, or can be adapted from existing literature.*
5. **The Cartesian Principle**
   Divide and conquer said René – put complexity at bay.
   *Most high-level performance attributes need decomposition into the list of sub-attributes that we are actually referring to. This makes it much easier to define complex concepts, like 'Usability', or 'Adaptability,' measurably.*
6. **The Principle of 'Quantification is not Measurement'**
   You don't have to measure in order to quantify!
   *There is an essential distinction between quantification and measurement. "I want to take a trip to the moon in nine picoseconds" is a clear requirement specification without measurement." The well-known problems of measuring systems accurately are no excuse for avoiding quantification.*

1 Paraphrasing a well-known old saying.

> Be clear about one thing. Quantification is not the same as Estimation and Measurement.

   *Quantification allows us to communicate about how good scalar attributes are or can be – before we have any need to measure them in the new systems.*
7. **The Principle of 'Meters Matter'**
   Measurement methods give real world feedback about our ideas.
   *A 'Meter' definition determines the quality and cost of measurement on a scale; it needs to be sufficient for control and for our purse.*
8. **The Principle of 'Horses for Courses'**[2]
   Different measuring processes will be necessary for different points in time, different events, and different places.[3]
9. **The Principle of 'The Answer always being '42' '**[4]
   Exact numbers are ambiguous unless the units of measure are well-defined and agreed.
   *Formally defined scales of measure avoid ambiguity. If you don't define scales of measure well, the requirement level might just as well be an arbitrary number.*
10. **The Principle of 'Being Sure About Results'**
    If you want to be sure of delivering the critical result – then quantify the requirement.
    *Critical requirements can hurt you if they go wrong – and you can always find a useful way to quantify the notion of 'going right' to help you avoid doing so.*

## Conclusions

This paper has tried to show how to define scales of measure for system attributes. It has also introduced the pragmatic detail available in Planguage for such specification and, for exploiting scales of measure to define benchmarks, targets and constraints.

Scales of measure are an essential means towards quantifying and getting control of your critical system attributes.

*Tom Gilb*
tom@gilb.com

## References

Gilb, Tom, *Principles of Software Engineering Management*. Addison-Wesley, 1988, 442 pages, ISBN 0-201-19246-2. See particularly page 150 (Usability) and Chapter 19 Software Engineering Templates.

Gilb, Tom and Graham, Dorothy, *Software Inspection*. Addison-Wesley, 1993, ISBN 0-201-63181-4, 471 pages.

Gilb, Tom, *Competitive Engineering*, Elsevier 2005 This book defines Planguage.)

Gilb, Tom. Various free papers, slides, and manuscripts on `http://www.Gilb.com/`. The manuscripts include:

- *Quantifying Quality* (Book manuscript draft Summer 2004, available from `Tom@Gilb.com` by request if not on website yet.)
- *Requirements Engineering* (about 500 slides giving examples and theory.) `http://www.Gilb.com/courseinfo`

Version April 15 2003 for INCOSE June Wash DC, updated Dec 14 2004. Paper accepted as a talk at INCOSE 2003, Washington DC, and published in the CD Proceedings.

2 'Horses for courses' is a UK expression indicating something must be appropriate for use, fit for purpose.

3 There is no universal static scale of measure. You need to tailor them to make them useful.

4 Concept made famous by Douglas Adams in *The Hitchiker's Guide to the Galaxy.*

# "Here be Dragons"

**by Alan Griffiths**

*"The use of animals in maps was commonplace from the earliest times. Man's deepest fears and superstitions concerning wide expanses of uncharted seas and vast tracts of 'terra incognita' were reflected in the monstrous animals that have appeared on maps ever since the Mappa Mundi." (Roderick Barron in "Decorative Maps")*

*For many developers C++ exception handling is like this - a Dark Continent with poor maps and rumours of ferocious beasts. I'm Alan Griffiths and I'm your guide to the landmarks and fauna of this region.*

In order to discuss exception safety we need to cover a lot of territory. The next section identifies the "exception safe" mountains in the distance. Please don't skip it in the hope of getting to the good stuff - if you don't take the time to get your bearings now you'll end up in the wastelands.

Once I've established the bearings I'll show you a well-trodden path that leads straight towards the highest peak and straight through a tar pit. From experience, I've concluded that everyone has to go down this way once. So I'll go with you to make sure you come back. Not everyone comes back; some give up on the journey, others press on deeper and deeper into the tar pit until they sink from sight.

On our journey I'll tell you the history of how the experts sought for a long time before they discovered a route that bypasses that tar pit and other obstacles. Most maps don't show it yet, but I'll show you the signs to look out for. I'll also show you that the beasts are friendly and how to enlist their aid.

If you look into the distance you'll see a range of peaks, these are the heights of exception safety and are our final destination. But before we proceed on our trek let me point out two of the most important of these peaks, we'll be using them as landmarks on our travels…

## The Mountains (Landmarks of Exception Safety)

The difficulty in writing exception safe code isn't in writing the code that throws an exception, or in writing the code that catches the exception to handle it. There are many sources that cover these basics. I'm going to address the greater challenge of writing the code that lies in between the two.

Imagine for a moment the call stack of a running program, function `a()` has called function `b()`, `b()` has called `c()`, and so on, until we reach `x()`; `x()` encounters a problem and throws an exception. This exception causes the stack to unwind, deleting automatic variables along the way, until the exception is caught and dealt with by `a()`.

I'm not going to spend any time on how to write functions `a()` or `x()`. I'm sure that the author of `x()` has a perfectly good reason for throwing an exception (running out of memory, disc storage, or whatever) and that the author of `a()` knows just what to do about it (display: "Sorry, please upgrade your computer and try again!").

The difficult problem is to write all the intervening functions in a way that ensures that something sensible happens as a result of this process. If we can achieve this we have "exception safe" code. Of course, that begs the question "what is 'something sensible'?" To answer this let us consider a typical function `f()` in the middle of the call stack. How should `f()` behave?

Well, if `f()` were to handle the exception it might be reasonable for it to complete its task by another method (a different algorithm, or returning a "failed" status code). However, we are assuming the exception won't be handled until we reach `a()`. Since `f()` doesn't run to completion we might reasonably expect that:

1. `f()` doesn't complete its task.
2. If `f()` has opened a file, acquired a lock on a mutex, or, more generally; if `f()` has "allocated a resource" then the resource should not leak. (The file must be closed, the mutex must be unlocked, etc.)
3. If `f()` changes a data structure, then that structure should remain useable - e.g. no dangling pointers.

In summary: If `f()` updates the system state, then the state must remain valid. Note that isn't quite the same as correct - for example, part of an address may have changed leaving a valid address object containing an incorrect address.

I'm going to call these conditions the basic exception safety guarantee, this is the first, and smaller of our landmark mountains. Take a good look at it so that you'll recognise it later.

The basic exception safety guarantee may seem daunting but not only will we reach this in our travels, we will be reaching an even higher peak called the strong exception safety guarantee that places a more demanding constraint on `f()`:

4. If `f()` terminates by propagating an exception then it has made no change to the state of the program.

Note that it is impossible to implement `f()` to deliver either the basic or strong exception safety guarantees if the behaviour in the presence of exceptions of the functions it calls isn't known. This is particularly relevant when the client of `f()` (that is `e()`) supplies the functions to be called either as callbacks, as implementations of virtual member functions, or via template parameters. In such cases the only recourse is to document the constraints on them - as, for example, the standard library does for types supplied as template parameters to the containers.

## Destructors That Throw Exceptions

Exceptions propagating from destructors cause a number of problems. For example, consider a `Whole` that holds pointers to a `PartOne`, a `PartTwo`, and a `PartThree` that it owns (ie it must delete them). If the destructors of the parts propagate exceptions, we would have trouble just writing a destructor for `Whole`. If more than one destructor throws, we must suppress at least one exception while remembering to destroy the third part. Writing update methods (like assignment) under such circumstances is prohibitively difficult or impossible.

There many situations where an exception propagating from a destructor is extremely inconvenient - my advice is not to allow classes that behave in this manner into your system. (If forced to, you can always 'wrap' them in a well behaved class of your own.)

If you look at the standard library containers, you'll find that they place certain requirements on the types that are supplied as template parameters. One of these is that the destructor doesn't throw exceptions. There is a good reason for this: it is hard to write code that manipulates objects that throw exceptions when you try to destroy them. In many cases, it is impossible to write efficient code under such circumstances.

In addition, the C++ exception handling mechanism itself objects to destructors propagating exception during the "stack unwinding" process. Indeed, unless the application developer takes extraordinary precautions the application will be terminated in a graceless manner.

There is no advantage in allowing destructors to propagate exceptions and a whole host of disadvantages. It should be easy to achieve: in most cases all a destructor should be doing is destroying other objects whose destructors shouldn't throw, or releasing resources - and if that fails an exception won't help.

Apart from the practicalities what does an exception from a destructor mean? If I try to destroy an object and this fails what am I supposed to do? Try again?
Destructors that throw exceptions? Just say no.

If we assume a design with fully encapsulated data then each function need only be held directly responsible for aspects of the object of which it is a member. For the rest, the code in each function must rely on the functions it calls to behave as documented. (We have to rely on documentation in this case, since in C++ there is no way to express these constraints in the code.)

We'll rest here a while, and I'll tell you a little of the history of this landscape. Please take the time to make sure that you are familiar with these two exception safety guarantees. Later, when we have gained some altitude we will find that there is another peak in the mountain range: the no-throw exception safety guarantee - as the name suggests this implies that `f()` will never propagate an exception.

## A History of This Territory

The C++ people first came to visit the land of exceptions around 1990 when Margaret Ellis and Bjarne Stroustrup published the *Annotated Reference Manual* [1]. Under the heading "experimental features" this described the basic mechanisms of exceptions in the language. In this early bestiary there is an early description of one of the friendly beasts we shall be meeting later on: it goes by the strange name of RESOURCE ACQUISITION IS INITIALISATION.

By the time the ISO C++ Standards committee circulated *Committee Draft 1* in early 1995 C++ people were truly living in exception land. They hadn't really mapped the territory or produced an accurate bestiary but they were committed to staying and it was expected that these would soon be available.

However, by late 1996 when *Committee Draft 2* was circulated the difficulties of this undertaking had become apparent. Around this time there came a number of reports from individual explorers. For example: Dave Abrahams identified the mountains we are using as landmarks in his paper *Exception Safety in STLPort* [2] although the basic exception safety guarantee was originally dubbed the "weak exception safety guarantee".

Some other studies of the region were produced by H Muller [3], Herb Sutter [4] and [5]. A little later came a sighting of another of the friendly beast that we will meet soon called ACQUISITION BEFORE RELEASE. This beast was first known by a subspecies named it COPY BEFORE RELEASE and was identified by Kevlin Henney [6] it is distinguished by the resources allocated being copies of dynamic objects.

By the time the ISO C++ Language Standard was published in 1998 the main tracks through the territory had been charted. In particular there are clauses in the standard guaranteeing the behaviour of the standard library functions in the presence of exceptions. Also, in a number of key places within the standard, special mention is made of another friendly beast - SWAP in its incarnation as the `std::swap()` template function. We will be examining SWAP after our detour through the tar pit.

Since the publication of the ISO standard more modern charts have been produced: the author in an early version of this article [7]. A similar route is followed by Bjarne Stroustrup [8]. Herb Sutter [9] takes a different route, but the same landmarks are clearly seen.

OK, that's enough rest, we are going to take the obvious path and head directly towards the strong exception safety guarantee.

## The Tar Pit

It is time to consider an example function, and for this part of the journey I have chosen the assignment operator for the following class:

```
class PartOne   { /* omitted */ };
class PartTwo   { /* omitted */ };
class Whole
{
public:
    // ...Lots omitted...
    Whole& operator=(const Whole& rhs);
private:
    PartOne*    p1;
    PartTwo*    p2;
};
```

Those of you that have lived in the old country will know the classical form for the assignment operator. It looks something like the following:

```
Whole& Whole::operator=(const Whole& rhs)
{
  if (&rhs != this)
  {
    delete p1;
    delete p2;
    p1 = new PartOne(*rhs.p1);
```

```
      p2 = new PartTwo(*rhs.p2);
    }
    return *this;
  }
```

If you've not seen this before, don't worry because in the new land it is not safe. Either of the new expressions could reasonably throw (since at the very least they attempt to allocate memory) and this would leave the `p1` and `p2` pointers dangling. In theory the "delete" expressions could also throw - but in this article we will assume that destructors never propagate exceptions. (See: "destructors that throw exceptions".)

The obvious solution to the problems caused by an exception being propagated is to catch the exception and do some clean up before throwing it again. After doing the obvious we have:

```
Whole& Whole::operator=(const Whole& rhs)
{
  if (&rhs != this)
  {
    PartOne* t1 = new PartOne(*rhs.p1);
    try
    {
      PartTwo* t2 = new PartTwo(*rhs.p2);
      delete p1;
      delete p2;
      p1 = t1;
      p2 = t2;
    }
    catch (...)
    {
      delete t1;
      throw;
    }
  }
  return *this;
}
```

Let's examine why this works:

1. An exception in the first new expression isn't a problem - we haven't yet allocated any resources or modified anything.
2. If an exception is propagated from the second new expression, we need to release t1. So we catch it, delete t1 and throw the exception again to let it propagate.
3. We are assuming that destructors don't throw, so we pass over the two deletes without incident. Similarly the two assignments are of base types (pointers) and cannot throw an exception.
4. The state of the Whole isn't altered until we've done all the things that might throw an exception.

If you peer carefully through the undergrowth you can see the first of the friendly beasts. This one is called ACQUISITION BEFORE RELEASE. It is recognised because the code is organised so that new resources (the new `PartOne` and `PartTwo`) are successfully acquired before the old ones are released.

We've achieved the strong exception safety guarantee on our first attempt! But there is some black sticky stuff on our boots.

## Tar!

There are problems lying just beneath the surface of this solution. I chose an example that would enable us to pass over the tar pit without sinking too deep. Despite this, we've

## Standard Algorithms and User Defined Template Classes

The `std::swap()` template functions are one example of an algorithm implemented by the standard library. It is also an example of one where there is a good reason for C++ users to endeavour to provide an implementation specific to the needs of the classes and class templates that they develop. This need is particularly significant to developers of extension libraries - who would like to ensure that what they develop will both work well with the library and with other extension libraries.

So consider the plight of a developer who is writing a template class `Foo<>` that takes a template parameter `T` and wishes to SWAP two instances of `T`. Now `Foo` is being instantiated with a fundamental type, or with an instance of any swappable type from the standard library the correct function can be resolved by writing:

```
using std::swap;
swap(t1, t2);
```

However, the primary template `std::swap()` that will be instantiated for other types is guaranteed to use copy construction and assignment unless an explicit specialisation has been provided (and this is impractical if `T` is a specialisation of a template class). As we have seen, copy construction and assignment probably won't meet the requirements of SWAP. Now this won't always matter, because a language mechanism "Argument-dependent name lookup" might introduce into overload resolution a function called `swap()` from the namespace in which `T` is declared, if this is the best match for arguments of type `T` then it is the one that gets called and the templates in `std` are ignored.

Now there are three problems with this:

1. Depending on the context of the above code Koenig Lookup produces different results. (A library developer might reasonably be expected to know the implications of a class member named "swap" and how to deal with them - but many don't.) Most C++ users will simply get it wrong - without any obvious errors when only the `std::swap()` templates are considered.
2. The standard places no requirements on functions called "swap" in any namespace other than `std` - so there is no guarantee that `bar::swap()` will do the right thing.
3. In a recent resolution of an issue the standards committee has indicated that where one standard function/template function is required to be used by another then the use should be fully qualified (i.e. `std::swap(t1, t2);`) to prevent the application of Koenig Lookup. If you (or I) provide `yournamespace::swap()` the standard algorithms won't use it.

Since the standards committee is still considering what to do about this I can't give you a watertight recommendation. I am hoping that in the short term a "technical corrigenda" will permit users to introduce new overloads of such template functions in `std`. So far as I know this technique works on all current implementations - if you know of one where it doesn't please let me know. In the longer term I am hoping that the core language will be extended to permit the partial specialisation of template functions (and also that the library changes to use partial specialisation in place of overloading).

For those that follow such things this is library issue 226 (`http://anubis.dkuug.dk/JTC1/SC22/WG21/docs/lwg-active.html`).

incurred costs: the line count has doubled and it takes a lot more effort to understand the code well enough to decide that it works. If you want to, you may take some time out to convince yourself of the existence of the tar pit - I'll wait. Try the analogous example with three pointers to parts or replacing the pointers with two parts whose assignment operators may throw exceptions. With real life examples things get very messy very quickly.

Many people have reached this point and got discouraged. I agree with them: routinely writing code this way is not reasonable. Too much effort is expended on exception safety housekeeping chores like releasing resources. If you hear that "writing exception safe code is hard" or that "all those `try...catch` blocks take up too much space" you are listening to someone that has discovered the tar pit.

I'm now going to show you how exception handling allows you to use less code (not more), and I'm not going to use a single `try...catch` block for the rest of the article! (In a real program the exception must be caught somewhere - like function `a()` in the discussion above, but most functions simply need to let the exceptions pass through safely.)

## The Royal Road

There are three "golden rules":

1. Destructors may not propagate exceptions,
2. The states of two instances of a class may be swapped without an exception being thrown,
3. An object may own at most one resource.

We've already met the first rule.

The second rule isn't obvious, but is the basis on which SWAP operates and is key to exception safety. The idea of SWAP is that for two instances of a class that owns resources exchanging the states is feasible without the need to allocate additional resources. Since nothing needs to be allocated, failure needn't be an option and consequently neither must throwing an exception. (It is worth mentioning that the no-throw guarantee is not feasible for assignment, which may have to allocate resources.)

If you look at the *ISO C++ Language Standard*, you'll find that `std::swap()` provides the no-throw guarantee for fundamental types and for relevant types in the standard library. This is achieved by overloading `std::swap()` - e.g. there is a template corresponding to each of the STL containers. This looks like a good way to approach SWAP but introducing additional overloads of `std::swap()` is not permitted by the language standard. The standard does permit to explicit specialisation of an existing `std::swap()` template function on user defined classes and this is what I would recommend doing where applicable (there is an example below). The standards committee is currently considering a defect report that addresses the problem caused by these rules for the authors of user defined template classes. (See: Standard Algorithms and User Defined Template Classes.)

The third rule addresses the cause of all the messy exception handling code we saw in the last section. It was because creating a new second part might fail that we wrote code to handle it and doubled the number of lines in the assignment operator.

We'll now revisit the last example and make use of the above rules. In order to conform to the rule regarding ownership of multiple objects we'll delegate the responsibility of resource ownership to a couple of helper classes. I'm using the `std::auto_ptr<>` template to generate the helper classes here because it is standard, not because it is the ideal choice. (See: "The Trouble With `std::auto_ptr<>`" for reasons to avoid using `auto_ptr<>` in this context.)

```
class Whole {
public:
  // ...Lots omitted...
  Whole& operator=(const Whole& rhs);
private:
  std::auto_ptr<PartOne>    p1;
  std::auto_ptr<PartTwo>    p2;
};
Whole& Whole::operator=(const Whole& rhs)
{
  std::auto_ptr<PartOne> t1(
     new PartOne(*rhs.p1));
  std::auto_ptr<PartTwo> t2(
     new PartTwo(*rhs.p2));
  std::swap(p1, t1);
  std::swap(p2, t2);
  return *this;
}
```

Not only is this shorter than the original exception-unsafe example, it meets the strong exception safety guarantee.

Look at why it works:

1. There are no leaks: whether the function exits normally, or via an exception, `t1` and `t2` will delete the parts they currently own.
2. The swap expressions cannot throw (second rule).
3. The state of the `Whole` isn't altered until we've done all the things that might throw an exception.

Oh, by the way, I've not forgotten about self-assignment. Think about it - you will see the code works without a test for self-assignment. Such a test may be a bad idea: assuming that self-assignment is very rare in real code and that the branch could have a significant cost. Francis Glassborow suggested a similar style of assignment operator as a speed optimisation [10]. Following on from this, Kevlin Henney explored its exception safety aspects in [11], [12] and [6].

We are on much firmer ground than before: it isn't hard to see why the code works and generalising it is simple. You should be able to see how to manage a `Whole` with three `auto_ptrs` to `Parts` without breaking stride.

You can also see another of the friendly beasts for the first time. Putting the allocation of a resource (here a new expression) into the initialiser of an instance of a class (eg `auto_ptr<PartOne>`) that will delete it on destruction is RESOURCE ACQUISITION IS INITIALISATION. And, of course, we can once again see ACQUISITION BEFORE RELEASE.

(Yes, in this case we could use assignment instead of SWAP to make the updates. However with a more complex type SWAP is necessary, as we shall see later. I use SWAP in this example for consistency.)

## The Assignment Operator - a Special Case

Before I go on to deal with having members that may throw when updated, I've a confession I need to make. It is possible, and usual, to write the assignment operator more simply than the way I've

just demonstrated. The above method is more general than what follows and can be applied when only some aspects of the state are being modified. The following applies only to assignment:

```
Whole& Whole::operator=(const Whole& rhs)
{
  Whole(rhs).swap(*this);
  Return *this;
}
```

Remember the second rule: `Whole` is a good citizen and provides for SWAP (by supplying the `swap()` member function). I also make use of the copy constructor - but it would be a perverse class design that supported assignment but not copy construction. I'm not sure whether the zoologists have determined the relationship between SWAP and copying here, but the traveller won't go far wrong in considering COPY AND SWAP as species in it own right.

For completeness, I'll show the methods used above:

```
void Whole::swap(Whole& that)
{
  std::swap(p1, that.p1);
  std::swap(p2, that.p2);
}
Whole::Whole(const Whole& rhs)
:   p1(new PartOne(*rhs.p1)),
    p2(new PartTwo(*rhs.p2))
{
}
```

One further point about making `Whole` a good citizen is that we need to specialise `std::swap()` to work through the `swap()` member function. By default `std::swap()` will use assignment - and not deliver the no-throw guarantee we need for SWAP. The standard allows us to specialise existing names in the `std` namespace on our own types, and it is good practice to do so in the header that defines the type.

```
Namespace std
{
  template<>
  inline void swap(exe::Whole& lhs,
                   exe::Whole& rhs)
  {
    lhs.swap(rhs);
  }
}
```

This avoids any unpleasant surprises for client code that attempts to `swap()` two `Wholes`.

Although we've focused on attaining the higher peak of strong exception safety guarantee, we've actually covered all the essential techniques for achieving either strong or basic exception safety. The remainder of the article shows the same techniques being employed in a more complex example and gives some indication of the reasons you might choose to approach the lesser altitudes of basic exception safety.

## In Bad Weather

We can't always rely on bright sunshine, or on member variables that are as easy to manipulate as pointers. Sometimes we have to deal with rain and snow, or base classes and member variables with internal state.

To introduce a more complicated example, I'm going to elaborate the `Whole` class we've just developed by adding methods that update `p1` and `p2`. Then I'll derive an `ExtendedWhole` class from it that also contains an instance of another class: `PartThree`. We'll be assuming that operations on `PartThree` are exception safe, but, for the purposes of discussion, I'll leave it open whether `PartThree` offers the basic or the strong exception safety guarantee.

```
Whole& Whole::setP1(const PartOne& value)
{
  p1.reset(new PartOne(value));
  return *this;
}
Whole& Whole::setP2(const PartTwo& value)
{
  p2.reset(new PartTwo(value));
  return *this;
}
class ExtendedWhole : private Whole
{
public:
  // Omitted constructors & assignment
  void swap(const ExtendedWhole& rhs);
  void setParts(
    const PartOne& p1,
    const PartTwo& p2,
    const PartThree& p3);
private:
  int       count;
  PartThree body;
};
```

The examples we've looked at so far are a sufficient guide to writing the constructors and assignment operators. We are going to focus on two methods: the `swap()` member function and a `setParts()` method that updates the parts.

Writing `swap()` looks pretty easy - we just swap the base class, and each of the members. Since each of these operations is "no-throw" the combination of them is also "no-throw".

```
void ExtendedWhole::swap(ExtendedWhole& rhs)
{
  Whole::swap(rhs);
  std::swap(count, rhs.count);
  std::swap(body,  rhs.body);
}
```

Writing `setParts()` looks equally easy: `Whole` provides methods for setting `p1` and `p2`, and we have access to `body` to set that. Each of these operations is exception safe, indeed the only one that need not make the strong exception safety guarantee is the assignment to `body`. Think about it for a moment: is this version of `setParts()` exception safe? And does it matter if the assignment to `body` offers the basic or strong guarantee?

```
void ExtendedWhole::setParts(
  const PartOne& p1,
  const PartTwo& p2,
  const PartThree& p3)
{
  setP1(p1);
  setP2(p2);
  body = p3;
}
```

Let's go through it together, none of the operations leak resources, and `setParts()` doesn't allocate any so we don't have any leaks. If an exception propagates from any of the

## The Trouble With std::auto_ptr<>

By historical accident, the standard library provides a single smart pointer template known as `auto_ptr<>`. `auto_ptr<>` has what I will politely describe as "interesting" copy semantics. Specifically, if one `auto_ptr<>` is assigned (or copy constructed) from another then they are both changed - the `auto_ptr<>` that originally owned the object loses ownership and becomes `0`. This is a trap for the unwary traveller! There are situations that call for this behaviour, but on most occasions that require a smart pointer the copy semantics cause a problem.

When we replace `PartXXX*` with `auto_ptr<PartXXX>` in the `Whole` class we still need to write the copy constructor and assignment operator carefully to avoid any `PartXXX` being passed from one `Whole` to another (with the consequence that one `Whole` loses its `PartXXX`).

Another effect of the odd copy behaviour is that `auto_ptr` does not meet the constraints stated in the standard for the `std::swap` template. So, technically the code in the main body isn't guaranteed to work. However, having discussed this issue in the "library" group of the standards committee it is clear that `std::swap` is intended to work with `auto_ptr<>`, and that no "reasonable" implementation would fail.

We encounter a further problem if we attempt to hide the implementation of `PartXXX` from the client code by using a forward declaration: we would also need to write the destructor. If we don't, the one the compiler generates for us will not correctly destroy the `PartXXX`. This is because the client code causes the generation of the `Whole` destructor and consequently instantiates the `auto_ptr<>` destructor without having seen the class definition for `PartXXX`. Technically, this is forbidden by the standard. But, as no diagnostic is required, the effect of this is typically to instantiate the `auto_ptr` destructor. This deletes an incomplete type - unless `PartXXX` has a trivial destructor this gives "undefined behaviour".

Although the standard library doesn't support our needs for a smart pointer very well it is possible to write ones which do. There are a couple of examples in the `arglib` library on my website.

(Both `arg::body_part_ptr<>` and `arg::grin_ptr<>` are more suitable than `std::auto_ptr<>`.) An excellent C++ library (that contains smart pointers) is boost [13].

operations, then they leave the corresponding sub-object in a useable state, and presumably that leaves `ExtendedWhole` useable (it is possible, but in this context implausible, to construct examples where this isn't true). However, if an exception propagates from `setP2()` or from the assignment then the system state has been changed. And this is so regardless of which guarantee `PartThree` makes.

The simple way to support the strong exception safety guarantee it to ensure that nothing is updated until we've executed all the steps that might throw an exception. This means taking copies of sub-objects and making the changes on the copies, prior to swapping the state between the copies and the original sub-objects:

```
Void ExtendedWhole::setParts(
  Const PartOne& p1,
  Const PartTwo& p2,
  Const PartThree& p3)
{
  Whole temp(*this);
  Temp.setP1(p1).setP2(p2);
  Body = p3;
  Whole::swap(temp);
}
```

Once again does it matter if the assignment to body offers the basic or strong guarantee? Yes it does, if it offers the strong guarantee then all is well with the above, if not then the assignment needs to be replaced with COPY AND SWAP vis:

```
PartThree(p3).swap(body);
```

Once again we have attained the highest peak, but this may not be healthy. On terrestrial mountains above a certain height there is a

[concluded at foot of next page]

## The Cost of Exception Handling

Compiler support for exception handling does make the generated code bigger (figures vary around 10-15%), but only for the same code. However, code isn't written the same way without exceptions - for example, since constructors cannot return an error code, idioms such as "two phase construction" are required. I have here a comparable piece of code to the final example that has been handed down the generations from a time before the introduction of exception handling to C++. (Actually I've made it up - but I was around back then and remember working with code like this, so it is an authentic fake.)

```
int ExtendedWhole::setParts(
  const PartOne& p1,
  const PartTwo& p2,
  const PartThree& p3)
{
  Whole tw;
  int rcode = tw.init(*this);
  if (!rcode) rcode = tw.setP1(p1);
  if (!rcode) rcode = tw.setP2(p2);
  if (!rcode)
  {
    PartThree t3;
    Rcode = t3.copy(p3);
    If (!rcode)
    {
      Whole::swap(tw);
      body.swap(t3);
   }
  }
  return rcode;
}
```

To modern eyes the need to repeat this testing and branch on return codes looks very like the tar-pit we encountered earlier - it is verbose, hard to validate code. I'm not aware of any trials where comparable code was developed using both techniques, but my expectation is that the saving in hand-written code from using exceptions significantly outweighs the extra cost in compiler-generated exception handling mechanisms.

Please don't take this as a rejection of return codes, they are one of the primary error reporting mechanisms in C++. But if an operation will only fail in exceptional circumstances (usually running out of a resource) or cannot reasonably be expected to be dealt with by the code at the call site then exceptions can greatly simplify the task.

# Two-thirds of a Pimpl and a Grin

by David O'Neil

This article describes an easy method to reduce compilation dependencies and build times. This method would not be interesting enough to warrant an article on it, except that it is the key to an interesting method of managing project-wide objects, and I have not seen this method mentioned anywhere.

## Background

While watching another religious war starting to erupt over the use of Singletons, I realized that I had not seen the method I use discussed anywhere else, including The Code Project [1]. It has some advantages to it, and, as I haven't seen it mentioned, I figured I'd post it. (Of course, having said that, I'll probably come across an article on the exact same thing tomorrow.)

## The Method

The method itself, as I said, is not very interesting. Simply use smart pointers to hold all non-POD objects stored in your class interface.

Well, there is one interesting thing about it - you cannot use `auto_ptr`'s or many other smart pointers in order to store your objects if you don't include at least an empty destructor inside the unit's `.cpp` file. That is the reason the following uses Alan Griffiths' `grin_ptr` [2]. You could also use `boost::shared_ptr`, and probably some others that I am unaware of.

The reason that you can't use an `auto_ptr` for this purpose is simply that `auto_ptr` must have a complete definition of the forward declared class at the point of destruction, and if you rely upon the default destructor, the forward declaration of the class is the only thing the `auto_ptr` has, so it will generate a 'do-nothing' default destructor for the class being held. This is rarely, if ever, what you want.

(If you are unsure whether a smart pointer can be used for this purpose without creating a destructor in the holding class's `.cpp` file, look in the smart pointer's documentation for a statement saying something to the effect that it can hold and correctly destroy an incomplete type. If it says that, you can safely use it without having to remember to supply a destructor in the `.cpp` file.)

As a quick example of the pattern I am talking about, here is a theoretical implementation of my wallet:

```
//Header file, include guards not shown
#include "arg.h"
//Only uses forward declarations:
class CreditCard;
class BusinessCard;
class DollarBill;
class Wallet {
private:
  // Make it simple - just one of each
  arg::grin_ptr<CreditCard> masterCardC;
  arg::grin_ptr<BusinessCard> businessCardC;
  arg::grin_ptr<DollarBill> dollarBillC;
  // anything else, but if they are classes,
  // wrap them in pointers as above.
public:
  Wallet();
  BusinessCard & businessCard()
    { return *businessCardC.get(); }
  // I really don't want to
  // expose the following two,
  // but this is simply an example...
```

[continued from previous page]

"death zone" where the supply of oxygen is insufficient to support life. Something similar happens with exception safety: there is a cost to implementing the strong exception safety guarantee. Although the code you write isn't much more complicated than the 'basic' version, additional objects are created and these allocate resources at runtime. This causes the program to make more use of resources and to spend time allocating and releasing them.

Trying to remain forever at high altitude will drain the vitality. Fortunately, the basic exception safety guarantee is below the death zone: when one makes a composite operation whose parts offer this guarantee one automatically attains the basic guarantee (as the first version of `setParts()` shows this is not true of the strong guarantee). From the basic guarantee there is an easy climb from this level to the strong guarantee by means of COPY AND SWAP.

## Looking Back

Before we descend from the peak of strong exception safety guarantee and return to our starting point look back over the route we covered. In the distance you can see the well-trampled path that led to the tar pit and just below us the few tracks leading from the tar pit up a treacherous scree slope to where we stand. Off to the left is the easier ridge path ascending from basic exception safety guarantee and beyond that the road that led us past the tar pit. Fix these landmarks in your mind and remember that the beasts we met are not as fierce as their reputations.

*Alan Griffiths*
alan@octopull.co.uk

## References

[1] Ellis & Stroustrup, *The Annotated C++ Reference Manual* ISBN 0-201-51459-1

[2] Abrahams, Dave *Exception Safety in STLPort* http://www.stlport.org/doc/exception_safety.html

[3] Muller, H *Ten rules for handling exception handling successfully* C++ Report Jan.'96

[4] Sutter, Herb *Designing exception-safe Generic Containers* C++ Report Sept.'97

[5] Sutter, Herb *More exception-safe Generic Containers* C++ Report Nov-Dec.'97

[6] Henny, Kevlin *Creating Stable Assignments* C++ Report June'98

[7] Griffiths, Alan *The safe path to C++ exceptions* EXE Dec.'99

[8] Stroustrup, Bjarne *The C++ Programming Language* (3rd Edition) appendix E "Standard Library Exception Safety" (this appendix dies not appear in early printings, but is available on the web at http://www.research.att.com/~bs/3rd_safe.pdf)

[9] Sutter, Herb *Exceptional C++* ISBN 0-201-61562-2

[10] Glassborow, Francis, "The Problem of Self-Assignment" in *Overload 19* ISSN 1354-3172

[11] Henney, Kevlin "Self Assignment? No Problem!" in *Overload 20* ISSN 1354-3172

[12] Henney, Kevlin "Self Assignment? No Problem!" in *Overload 21* ISSN 1354-3172

[13] boost http://www.boost.org/

```
  CreditCard & masterCard()
    { return *masterCardC.get(); }
  DollarBill & dollarBill()
    { return *dollarBillC.get(); }
  // anything else...
};
//Implementation file
#include "Wallet.h"
#include "CreditCard.h"
#include "BusinessCard.h"
#include "DollarBill.h"
Wallet::Wallet() :
  masterCardC(new MasterCard(/*any args*/)),
  businessCardC(new BusinessCard(/*any
args*/)),
  dollarBillC(new DollarBill()) { }
// And anything else...
```

(Feel free to 'new' me some more dollar bills. :) )
Anyway, as you can see, nothing to get excited over, until you think about it for a second. We have just entirely eliminated all external dependencies except the `arg.h` file from the `Wallet` header. If the implementation to `CreditCard`, `BusinessCard`, or `DollarBill` changes, the only units that need to be recompiled in the project are the `Wallet` unit and the unit that you changed. This is a big saving over having 'hard objects' in the class's header file. In that case, every unit that `#included` `Wallet.h` would be recompiled anytime the implementation to `CreditCard`, `BusinessCard`, or `DollarBill` changed.

The saving with the above method is not as good as a full pimpl implementation, as a full pimpl implementation enables you to recompile `CreditCard`, `BusinessCard`, or `DollarBill`, without the `Wallet` unit or any other unit in the project needing to be recompiled. (Of course, changing the interface to a pimpl can be a PITA, and will require more to be recompiled at that time.)

The method I have just outlined is simpler than the pimpl pattern, as it does not require you to create an intermediary `PimplHolder` class. You do, however, have to use `->` notation to access all of the smart pointer held objects from within the class they are held in, unless you create a local reference to them within the function using them.

## The 'Interesting Use'

The above method can be used to easily manage project-wide objects. This method, when used in a global, can quite often be used as a quasi-Singleton manager. Doing so will often simplify some aspects of your overall design, and it will do so without increasing compilation dependencies.

Please do not take this to mean that I don't like Singletons. The pattern I am about to show you does not replace Singletons. There is nothing in this pattern to keep you from instantiating multiple copies of the objects held in this container. This pattern simply makes it very easy to manage and use project-wide objects, and it may be an appealing alternative if you do not really care to mess with figuring your way around Singleton instantiation order dependencies.

Also, do not take this to mean that I am a huge proponent of globals. This pattern allows me to minimize the use of globals to two or three for my entire project, and I am happy with that. I do store quite a few variables within the global objects, though, and as these variables are defined within the header file of the global, whenever their interface changes, or I add another item to the global, every unit that `#includes "Globals.h"` will be recompiled at that time. If your project takes considerable time for a rebuild of such a nature, you will want to carefully atomize your globals, maybe even to the point of making each object (like `TextureManager` in the following code) into its own global item. I outline a wrapper that will simplify this for you in the addendum at the end of this article.

Let me give a simple example of this 'interesting use'. The two changes to the previous example that are needed are to change it so that it holds things commonly held in Singletons, and make the class into a global. The example I gave while the religious war was raging was the following:

```
// Header file (minus include guards again)
#include "arg.h"
class TextureManager;
class LoggingSystem;
class ObjectManager;
// ...
class Globals {
private:
  arg::grin_ptr<TextureManager>
textureManagerC;
  arg::grin_ptr<LoggingSystem> loggerC;
  arg::grin_ptr<ObjectManager> objectManagerC;
  // ...
public:
  Globals();
  TextureManager & textureManager()
    { return *textureManagerC.get(); }
  LoggingSystem  & logger()
    { return *loggerC.get(); }
  ObjectManager  & objectManager()
    { return *objectManagerC.get(); }
  // ...
};
// Implementation file:
#include "TextureManager.h"
#include "LoggingSystem.h"
#include "ObjectManager.h"
Globals::Globals() :
  textureManagerC(new TextureManager()),
  loggerC(new LoggingSystem()),
  objectManagerC(new ObjectManager())
  /* and any other stuff */ { }
// Here is a sample of a 'main' file:
#include "Globals.h"
Globals gGlobals;
// The following #include is only
// so we can access 'doSomething'.
// We don't need it for the global creation.
#include "TextureManager.h"
int main() {
  gGlobals.textureManager().doSomething();
  //...
  return 0;
}
```

And that is it, although if you need to pass in initialization parameters, in order to pass them to one of your classes, you will need to implement `gGlobals` as a pointer, and initialize it after whatever parameter it is dependant upon is obtained. The best option is to implement it through the use of an `auto_ptr` (in

which case auto_ptr has no problems):

```
Globals * gGlobals;
int main() {
  // Do whatever in order to get your 'args'
  // ... and finally
  std::auto_ptr<Globals> tGlobals(
     new Globals(/*args*/));
  gGlobals = tGlobals.get();
  //...
}
```

Using the method outlined above, you can explicitly control your object creation order, and very easily overcome the issues that arise when trying to control multiple Singletons with inter-singleton creation order dependencies. In addition, this method has a simpler syntax than Singletons. Singletons require something like:

```
SingletonManager::getInstance().
    textureManager().doSomething()
```

in order to use them from a Singleton manager. The above method boils down to:

```
gGlobals.textureManager().doSomething()
```

But the truly interesting part is that using this technique, if you only modify the TextureManager.cpp file, it will be the only file recompiled at recompilation. If you modify the TextureManager.h file, only units that explicitly `#include "TextureManager.h"` will be recompiled. This will include the `Globals` unit, but will not include every file that `#includes "Globals.h"`.

It is worth reading the last paragraph again, and looking at the code, until you understand that this system is not exposing any of the other objects being managed by the `Globals` unit to any unit that is not `#include`-ing the sub-unit you wish access to. You can `#include "Globals.h"` in every .cpp file in your program, but they won't link to `TextureManager` until you explicitly `#include "TextureManager.h"` as well as Globals.h in the unit you want to access the `TextureManager` from. There are no other compilation dependencies to be aware of, and the `Globals` unit does not impose any more overhead than a few forward declarations, the class declaration of `Globals` itself, and a few bits for the `grin_ptr`'s internals.

The secrets to this whole technique: using only forward declarations and a capable smart pointer.

I hope that you find this technique useful, and wish you happy coding.

## Addendum

If you do atomize your globals, and do not wish to use Singletons, you can modify the previous method to instantiate your globals within `main`, and completely control your instantiation and destruction order:

```
Globals * gGlobals;
TextureManager * gTextureMan;
// ...
int main() {
  std::auto_ptr<Globals> tGlobals(
     new Globals(/*args*/));
  gGlobals = tGlobals.get();
  std::auto_ptr<TextureManager>
     tTextureMan(new TextureManager());
  gTextureMan = tTextureMan.get();
  // And, if you want, you can even destroy
  // them in any order. Just manually call
  // 'release' on the pointers in the order
  // you want at the end of 'main', rather
  // than relying upon the auto_ptr's
  // destructors.
}
```

You could even create a class to manage these atomized globals. Doing so would overcome the previous objection to long build times. I envision something of the following nature:

```
// GlobalManager.h w/o include guards
class TextureManager;
class OtherGlobals;
class GlobalManager {
private:
  arg::grin_ptr<TextureManager>
     textureManagerC;
  arg::grin_ptr<OtherGlobals> otherGlobalsC;
  //...
};
//GlobalManager.cpp
#include "TextureManager.h"
TextureManager * gTextureMan;
#include "OtherGlobals.h"
OtherGlobals * gOtherGlobals;
GlobalManager::GlobalManager() {
  textureManagerC.reset(new TextureManager());
  gTextureMan = textureManagerC.get();
  otherGlobalsC.reset(new OtherGlobals());
  gOtherGlobals = otherGlobalsC.get();
  // ...
}
//Main unit:
#include "GlobalManager.h"
int main() {
  // Automatically instantiate all
  // globals in one fell swoop:
  std::auto_ptr<GlobalManager>
     globals(new GlobalManager());
  // ...
}
```

Using this method, all of your globals will automatically be instantiated for you in a manner that only forces your main unit to recompile if you add more globals. You no longer have the 'hiding' that took place in the earlier pattern I discussed, but you have a simple method of controlling your global class instantiation order. You can even explicitly control the destruction order, if you desire, by creating a destructor for the global organizer class, and calling 'release' upon the smart pointers in the order you want the objects to be released.

Hopefully, the above discussion has given you more options when it comes to implementing global objects. As always, use what works for you, and keep the religious wars to a minimum :)

*David O'Neil*
david@randommonkeyworks.com

## References

[1] *The Code Project*, http://www.codeproject.com/
[2] Alan Griffiths, 1999, *Ending with the Grin*, http://www.octopull.demon.co.uk/arglib/TheGrin.html