contents

credits & contacts

Editor:

John Merrells,
merrells@acm.org
808 East Dana St,
Mountain View,
CA 94041, U.S.A

Advertising:

Pete Goodliffe, Chris Lowe
ads@accu.org

Membership:

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

Readers:

Ian Bruntlett
IanBruntlett@accu.org

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

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

Membership fees and how to join:

Basic (C Vu only): £15
Full (C Vu and Overload): £25
Corporate: £80
Students: half normal rate
ISDF fee (optional) to support Standards work: £21
There are 6 journals of each type produced every year.
Join on the web at www.accu.org with a debit/credit card, T/Polo shirts available.
Want to use cheque and post - email membership@accu.org for an application form.
Any questions - just email membership@accu.org.

Editorial - On Writing

I am sometimes asked how one goes about writing an article for Overload. I usually rattle off an email with a few random thoughts about getting the text down and editing it into shape. This editorial is my attempt to properly address the topic.

But, why write?

In The Elements of Style [1], Strunk and White describe the rewards of writing as:

> "[The writer] will find it increasingly easy to break through the barriers that separate him from other minds, other hearts – which is, of course, the purpose of writing, as well as its principle reward."

Well quite, but people write either because they enjoy it, or because they know that it's good for them. I'm definitely in the latter group. Words do not flow from my fingertips in the same way that code does, but I know that I have benefited from improving the quality of my writing. Strunk and White continue:

> "…the act of composition, or creation, disciplines the mind; writing is one way to go about thinking, and the practice and habit of writing not only drain the mind but supply it, too."

The distinguishing quality of a senior engineer is their ability to communicate ideas clearly and efficiently. I've seen talented engineers limit their progression, unable to project themselves beyond their immediate workgroup, because they do not take up opportunities to write or conduct presentations.

Selecting a topic

Selecting a topic can be the hardest part of writing an article. We all have a vast array of thoughts spinning around inside our heads, but it can be hard to pin down one that we think is interesting enough to present to others.

Don't judge your ideas too harshly. Something you think of as simplistic and well known will turn out to be multi-faceted and interesting under further examination.

Don't try to cover too much. It can be overwhelming to write about the architecture of an entire system, or even a hundred lines of code. Some of the best Overload articles are those that carefully examine a pattern, an idiom, or even a single phrase or keyword.

People naturally want to write about their successes, but we actually learn more from our failures. Learning to examine and share our failures is an important developmental milestone for us all, both as individuals and as professionals.

'Write about what you know', is the most common advice given to prospective authors. Indeed, draw from your own work experiences. Base your writing on problems you are trying to solve, and the solutions that could be deployed.

Conversely, I find that 'write about what you don't know' to also be true. A difficulty with writing about what you know, is that the topic material can be so ingrained in your being that it is no longer at the forefront of your mind. The process of researching and documenting a new topic can be easier than the introspection required to dredge up the reasoning for the everyday assumptions under which you operate.

Overall I'd say that the journey is more interesting than the destination. For me, the process of solving a problem is more interesting than a statement of the solution. For example, when I interview engineers I look for people who know how to go about solving a problem, rather than people who know the solutions to problems.

Audience

In all writing it is important to consider your audience. This is simple for Overload; your readers are people just like you. I have in mind professional software engineers who are self-educating but busy people. They are seeking a forum of peers in which to share their thoughts, learn from others, and discuss ideas.

Planning

Inexperienced writers often skip this important stage. Would we start coding before designing? Perhaps that's not the best analogy, but a simple plan can keep an article on track. Without a plan, the tendency is to produce a wayward collection of random paragraphs. The editing process then retrospectively imposes a plan, which rarely works out well.

Developing

This is the process of collecting the material that will make up your article. I personally have most trouble with this stage. I'm unable to capture all my thoughts on a topic in a single session. I read background material and build up a sheaf of hand-written notes before I reach for the keyboard. Jotting down potential sub-topics, key phrases, and supportive material like examples, references, and quotations helps me a lot.

Avoid starting by writing code. The text is more important than the code. The article should be about the writing the code, not what the completed code looks like.

Organizing your material

Given a plan and a collection of notes you can now develop an outline of the article. The outline pulls together related material, showing how ideas are grouped and related to each other. This process helps bring balance to the article by ensuring sufficient coverage for each point.

Rough draft

At this stage it is important for you to adopt a state of mind where getting any text down is more important than getting the perfect text down. Forget grammar, punctuation, and spelling, just get started. Possible ways to approach a draft are to:

- **Start at the beginning** – Writing the introduction is a natural place to start and planning the route will ease the journey. I often start with the introduction, but with the explicit assumption that I'll throw most of it away in the first revision. This seems to help me get going.
- **Start at the end** – Writing the conclusion first makes a clear statement about the destination.
- **Start in the middle** – Start with the part of the document that you feel most confident about.
- **Throw one draft away** – Just assuming that the first draft is to be thrown away can help grease the writing wheels. When the draft is done you may decide it's good enough not to bother starting from a blank page again.
- **Develop alternative drafts** – Writing multiple drafts from alternative perspectives can help you to find the best way to approach a topic.

The difficulty with getting started is psychological; you must be in the right frame of mind to write. Any number of factors can contribute: time pressure, the location, or distractions. Write where and when you feel comfortable. Schedule time specifically for writing. Turn down the ringer on the phone. Close your office door (if you are so blessed). Go home. Go to work. Go to the library. Shut down your email client. Above all just focus on the present and set yourself achievable goals.

My favourite technique is to send myself an email. I feel totally uninhibited writing email messages. I can crank out a paragraph in about thirty seconds, a paragraph that might take me an hour with a word processor.

Editing the rough draft

Having completed the rough draft it's best to take a break from writing, so that you can return to the text with a fresh mind. I find a good night's sleep works for me, others may prefer a couple of days, or even a week. The rough draft should be edited for substance rather than language. Don't waste time fixing up the text for publication, concentrate on which points should remain, and which should be eliminated.

Now is a good time to think about the length of the article. Overload magazine has no minimum or maximum article length restrictions, but we usually serialize articles longer than five pages. We rarely receive short submissions, which is unfortunate as they are very useful when composing an issue.

Revising the first draft

Check the draft against the plan. Has the objective has been achieved? Is the message clear, and has the main point been adequately addressed?

Test the draft against the outline, revise the organization to group ideas together and put them in the proper order. Balance the main points of the article so that they get equal attention, making sure that there is enough supporting material for each, not too much, and not too little. A hard, but important, step is removing content that does not contribute to the main point. For example, Allan Kelly makes excellent use of sidebars to further develop material that is surplus, yet still supportive to the main point.

Revising the second draft

The second revision of the draft focuses on the text of the article; the paragraphs, sentences, and words.

- **Revise paragraphs** – Each paragraph should express one point. They should vary in length, typically between 30 and 150 words, not all long, and not all short. Use short paragraphs for emphasis, and long paragraphs for descriptive or discursive text.
- **Revise flow** – Ensure that the narrative of the article flows smoothly from paragraph to paragraph.
- **Revise sentences and words**– Use a variety of sentence constructions. Vary sentence length; short sentences for impact, long sentences for descriptions. Rely on your inner ear to find sentences that don't scan well. Rework any that sound awkward, imprecise or wordy. Spike Milligan said, "Clichés are the handrail of the mind". Substitute them for fresh, interesting descriptions. Prefer the active voice to the passive voice.
- **Revise for tone** – Everyone writes in their own voice, but articles written for Overload should be a mixture of instructive, confident, explorative, and friendly; you should avoid preachy, arrogant, fancy, or overly academic and formal tones.
- **Revise the introduction** – Check that it introduces the topic of the article. The introduction can also be used to grab the reader's attention, possibly by asking a question, or making a blunt statement.
- **Check the conclusion** – The conclusion can be used to pull together all the parts of the article, it can rephrase the main argument, or it can reaffirm the importance of the topic. Above all it should convey a sense of completion.

There are three excellent books that I often refer to during this stage of the writing process. The Elements of Style [1] is a short classic that I mean to reread more often than I do. Essential English Grammar [2] is a short guide that makes up for my not having paid any attention to my English teacher. And, Bugs in Writing [3] is an instructional and beautifully presented book written specifically for engineers who write.

Revising the third draft

The final revision is for overall style, grammar, punctuation, and spelling. Read the text slowly, carefully, and aloud. Let your ear guide you. If some text doesn't sound quite right, rework it. You can also ask someone else to read and comment on your draft. Select a reader who will provide you with constructive and affirmative feedback.

Introduce titles and sub-headings to clearly identify topics. They can also pull the reader into the article as they browse through the magazine.

Use font effects sparingly; reserve them for emphasis to clarify meaning.

Submission to Overload

Articles should be submitted to the Overload editor after the second or third revision. An editorial board, comprised of the editor and a number of readers, manages the content of the magazine. The editor distributes the article to readers, who read the article for technical correctness and relevance. The readers return their comments to the author, who revises the article once again and resubmits the final draft to the editor. The editor performs the final proof reading before passing the article on to the production editor for inclusion in the next issue.

You'd write, only...

- **You don't have time** – You should regard writing as an investment in yourself. You take time to educate yourself to keep your technical skills fresh and relevant. Writing is another important skill you should nurture. Try asking for some work time to write an article. Enlightened management will recognize your increased value to the organization. In order words, writing an article for Overload will get you a promotion!
- **You're not so good at writing** – The hardest part is getting started, but by developing a regular writing habit you will improve with practice. We'll help you get through the writing process by discussing topics, approaches and editing drafts.
- **People won't want to read what you write** – To publish seems to be calling for attention, inviting others to judge you. But the Overload audience is friendly and supportive and I've only ever received positive comments for my writing in Overload.
- "**Bjarne is smarter than me"** – Probably, but we publish articles at all levels. An issue full of language extension proposals would not be fun for any of us.
- **You just don't feel good about this** – Writing is much like public speaking, it takes time to gain confidence. Start with a small audience and build from there. Challenge yourself.
- **You mean to, but you can't get started**– Take each stage at a time. It's hard to sit down at your desk with the goal of writing a book. A more manageable goal to set is to write a page, or even a single paragraph. As each milestone is achieved you build success upon success until you reach the greater goal.
- **You don't have anything to say** – Trust me, you do.

You have no choice

I hope this editorial has persuaded you that there are real benefits from writing for publication, and that writing an article for Overload or CVu is something that you can and should strive to do.

John Merrells

References

[1] Strunk and White, *The Elements of Style*, Allyn and Bacon.
[2] Gucker, *Essential English Grammar,* Dover.
[3] Lyn Dupré, *Bugs in Writing*, Addison-Wesley.

Copy Deadline

All articles intended for publication in *Overload 53* should be submitted to the editor by January 1st 2003, and for *Overload 54* by February 14th 2003. **Note earlier than usual deadline for *Overload 54* - this is to allow us to produce the April journals in time for the conference.**

From Mechanism to Method – Good Qualifications

by Kevlin Henney

Introduction

When it is not necessary to change, it is necessary not to change.
Lucius Cary, Viscount Falkland, 1610-1643

Change. In every day life it is seen as something either to embrace and face or to resist and fear. It is either as good as a rest or something that leopard spots simply do not do. It is also, when it comes to matters of state, at the heart of procedural programming, making it a principle and principal concern for C++ developers: events cause functions to be executed; objects are created and destroyed; variables are assigned.

But as a path of control flows through our code not everything is about change. In places the flow is smooth and unchanging, and importantly so. It is important that things we regard as constants remain so, giving rise to the oxymoron *constant variable*. It is important that some functions do not change the objects they are called on or with. It is important that some function results do not allow subsequent changes.

Change Management

In C++ the responsibility of documenting and enforcing the absence of change is given to `const`, and that of communicating asynchronous and unpredictable change is given to `volatile`, by far the lesser of the two qualifiers. In combination, the apparently oxymoronic `const volatile` leaves many developers bemused, but makes some sense when applied to references or pointers that offer read-only semantics to asynchronously changing values.

Ringing the Changes

A simple dictionary class, which allows you to look up a string value based on a unique string key, demonstrates common applications of `const` with respect to member functions, references, and pointers:

```
class dictionary {
public:
  bool empty() const;
  size_t size() const;
  const std::string *lookup(
           const std::string &) const;
  void insert(const std::string &,
            const std::string &);
  void erase(const std::string &);
  void clear();
  ...
private:
  ...
  typedef std::map<std::string,
                std::string> map;
  map content;
};
std::ostream &operator<<(ostream &,
                   const dictionary &);
std::istream &operator>>(istream &,
                   dictionary &);
...
```

With the exception of the `lookup` function, the function names and semantics correspond to those in the standard library [ISO1998]. Being able to read this interface with respect to mutability helps you determine some of the expected behavior of the class.

Don't Change the Spots

In some cases we can allow change behind the scenes with `mutable`, supporting discreet change on data members even under the rule of `const`. Sometimes referred to as the *anti-`const`*, `mutable`'s role is to support the occasional discrepancy between a `const`-correct class public interface and its underlying physical implementation. Rather than modify the interface – and therefore affect the class user – to reflect optimizations such as caching, `mutable` allows the interface to remain stable and implementation details that do not affect the usage to remain encapsulated.

Let us assume that in using the `dictionary` class we discover that there is a good chance that we look up a given key many times in a row. We could try to optimize this by keeping a cache. Preservation of the class's perceived interface and functional behavior is assisted by `mutable`:

```
class dictionary {
  ...
  mutable map::const_iterator last_lookup;
};

const std::string *dictionary::lookup(
          const std::string &key) const {
  if(last_lookup == content.end() ||
             last_lookup->first != key)
    last_lookup = content.find(key);
  return last_lookup != content.end()
            ? &last_lookup->second : 0;
}
```

`mutable` has helped bridge any discrepancy between physical and logical `const`-ness. However, note that this solution is not appropriate in an environment where dictionary objects are shared between threads. Between each of these two implementation options the type perceived by the class user has remained stable.

Substitutability and Mutability

What is an object's type? Is it the class from which it is created? Is it the functions that can be applied to it, whether global or member? Is it its usage? Is it its capabilities and behavior? Is it the classification that groups similar objects together? In truth, *type* can mean any one of these, depending on the context. You can see that in some way they are all related – or at least can be – to one another.

Of Types and Classes

If we restrict the notion of type to be the declared class of an object and the functions that work on it, we may have a syntactic notion of type, but we are short of a model of usage – sure I can write code against it that compiles, but what am I expecting at runtime? In the dictionary example, we can see how to write code that compiles, but what result are we expecting from a call to dictionary::lookup? If we say that a class defines the expected behavior from the member and global functions that can be applied its instances, we can equate the syntax and semantics of the class directly with a notion of type that satisfies most possible definitions of the word.

What about template type parameters? These are constrained by syntax usage but not by class. That is, after all, the idea of templates: They are generic and not locked into specific class hierarchies. In the STL, the compile-time requirements for syntax usage are supplemented by operational requirements. For instance, an object that is *CopyConstructible* [ISO1998] satisfies a set of requirements that goes beyond the simple syntax of a copy constructor, so that std::list<int> is *CopyConstructible* whereas std::auto_ptr<int> is not. These syntactic and semantic requirements together form an equally valid concept of type.

What seems to be common across these notions of type is that a type names and describes a particular model of usage and external behavior for an object. In the case of a class, the type name exists explicitly in the source code, the usage is defined by the functions in its interface (according to the *Interface Principle* [Sutter2000]), and the behavior is described outside of the compiled class interface (comments, unit tests, external documentation, word of mouth, class author's head, etc.). In the case of a template type parameter, the type name is not truly in the source code, the usage is defined by expression syntax, and the behavior is again implied and beyond the code.

So, how do const and volatile qualifiers relate to our notion of type? OK, if we're being practical: How does the const qualifier affect our notion of type? When const is applied to an object, whether directly on a value declaration or indirectly via a pointer or reference, it changes our view of that object. To be precise, it restricts what we can do with it. In other words it affects the model of usage, and hence the type. For instance, a plain int supports assignment operations, such as =, +=, and ++, whereas a const int does not. A std::string supports modifier functions such as clear and append, whereas a const std::string supports only query functions. Therefore, the typical class public interface is really two interfaces: The interface for const qualified objects and the interface for non-const qualified objects.

Of Type Hierarchies and Class Hierarchies

Relating classes together with derivation forms a class hierarchy. From the perspective of an external user of the class hierarchy – as opposed to someone implementing or extending it – only public derivation is of interest. What is the best advice for forming inheritance relationships? Substitutability or, to be more precise, the Liskov Substitution Principle (LSP) [Liskov1987]:

> A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a *subtype* is one whose objects provide all the behavior of objects of another type (the *supertype*) plus something extra. What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

In a nutshell, class hierarchy should follow type hierarchy. This recommendation is more detailed than the more common *is-a* or *is-a-kind-of* recommendation, which creates a taxonomy ensuring that each derived class *is a kind of* its base class. LSP is more detailed because it considers explicitly the behavior of the types involved.

Notice that – without actually having it officially stated – the implicit conversion in C++ from pointers or references from derived to base underpins an assumption that LSP is followed. The compiler knows nothing of the expected semantics of your classes, but it knows in code that where a base instance is expected a derived instance can be supplied. You drop LSP at your own risk.

There is another assumption that LSP is a recommendation only for organizing inheritance between classes in OO systems [Coplien1992, Sutter2000]. Notice that, if you read the recommendation carefully, there is no mention of classes. LSP is about relationships between types. Substitutability as defined applies not only to class hierarchies, but also to other notions of type based on models of usage [Henney2000a]: conversions [Henney2000b], overloading [Henney2000c], templates (so there is no need for a generic variant of LSP [Sutter2000]), and mutability.

We can relate mutability directly to const and non-const, and substitutability to the relationship between them: For a given class, a non-const object is a subtype of a const object because it may be used wherever the const version is expected. The non-const type also supports the interface of the const type. Pointer and reference conversions work in the way that you would expect: To go from non-const to const is implicit, whereas to go the other way, against the grain, requires an explicit conversion, a const_cast. Compare this to the implicit derived to base conversion with inheritance, and the explicit static_cast (or safer dynamic_cast) to go the other way.

Returning to the dictionary class, we can take some artistic and linguistic license to consider it to be two types with a subtyping relationship between them:

```
class const dictionary { // not legal C++
public: // const member functions only
  bool empty() const;
  size_t size() const;
  const std::string *lookup(
              const std::string &) const;
  ...
};
// globals taking const dictionary
// references only
std::ostream &operator<<(ostream &,
                     const dictionary &);
...

class dictionary : public
     const dictionary { // not legal C++
public: // additional non-const
        // member functions
  void insert(const std::string &,
              const std::string &);
  void erase(const std::string &);
  void clear();
  ...
};
// additional globals taking non-const
// dictionary references
std::istream &operator>>(istream &,
dictionary &);
...
```

From this, it is clear that when we see `const dictionary` in code we are looking at the *as-if* type represented by the first fragment, and when we see plain `dictionary` in code it is the second fragment, which builds on the first.

Specialization

Where there are subtypes there is specialization. Specialization can be with respect to extension, i.e. the subtype extends the interface of the supertype with more operations. It can also be with respect to constraints, i.e. the subtype's operations are more specific with respect to behaviour or result types.

In a class hierarchy classes typically acquire more operations the further down the hierarchy you descend. The guarantees of behavior can also become more specific. For example, if a base class function guarantees that its pointer result is null or non-null, an overridden version in a derived class can satisfy substitutability by never returning null. Conversely, if a base class function requires that its pointer argument must never be null, a derived class version can legitimately liberalize this to also accommodate null. The specialization of result also applies to return type: Assuming that the return type is a class pointer or reference, an overridden function can redeclare the return type to be a derived class pointer or reference.

This much is standard in OO: Runtime polymorphism offers us the method for such specialization, and `virtual` functions the mechanism. What of `const` and non-`const`? There is no concept of runtime polymorphism related to mutability. However, overloading offers us a compile-time variant of overriding: We can overload with respect to`const`-ness. In this compile-time view of polymorphism (the foundation of generic programming in C++) selection is performed with respect to `const`-ness for member functions on their target object and functions in general with respect to their arguments.

Given two member functions of the same name and similar signature, differentiated only by `const`-ness, the `const` version will be the only viable option for `const` access to an object. For non-`const` access, both functions are in theory available, but the compiler will select the more specific version, i.e. the non-`const` one. The most common reason for such *overriding* is to specialize the return type, e.g. `operator[]` on strings should allow scribble access for non-`const` strings and read-only access for `const` strings. In our `dictionary` class, a more STL-based approach to lookup demonstrates this approach:

```
class dictionary {
public:
  typedef map::const_iterator
                      const_iterator;
  typedef map::iterator iterator;
  ...
  const_iterator find(
               const string &) const;
  iterator find(const string &);
  ...
};
```

Viewing this again in terms of `const`-based type and subtype gives us the following interfaces:

```
class const dictionary { // not legal C++
public:
  const_iterator find(const string &)
const;
  ...
};

class dictionary : public const
           dictionary { // not legal C++
public:
  iterator find(const string &);
           // more specialized 'override'
  ...
};
```

Conclusion

In C++, `const` divides the novice from the experienced: on one side lies a source of confusion; on the other a means of clarification. Explicit annotation of modifier from query functions can benefit a system, and this is a concept that can be expressed in C++ using type qualifiers. Thus `volatile` and `const` – as well as mutable – are unified under the heading of change, even if the names are not as well chosen as they might be.

Qualification relates to the notion of type in terms of usage and behavior, and with it subtyping and all its accumulated practices and understanding. One valuable property of subtyping is substitutability. Although it is often clear from the context, we sometimes need to clarify what kind of substitutability we are referring to, i.e. substitutability with respect to *what*? In the case of `const` it is substitutability with respect to change.

Kevlin Henney
kevlin@curbralan.com

References

[Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

[Henney2000a] Kevlin Henney, "From Mechanism to Method: Substitutability", *C++ Report* 12(5), May 2000, also available from `http://www.curbralan.com`.

[Henney2000b] Kevlin Henney, "From Mechanism to Method: Valued Conversions", *C++ Report* 12(7), May 2000, also available from `http://www.curbralan.com`.

[Henney2000c] Kevlin Henney, "From Mechanism to Method: Function Follows Form", *C/C++ Users Journal C++ Experts Forum*, November 2000, `http://www.cuj.com/experts/1811/henney.html`.

[ISO1998] *International Standard: Programming Language - C++*, ISO/IEC 14882:1998(E), 1998.

[Liskov1987] Barbara Liskov, "Data Abstraction and Hierarchy", *OOPSLA '87 Addendum to the Proceedings*, October 1987.

[Sutter2000] Herb Sutter, *Exceptional C++*

This article was originally published on the C/C++ Users Journal C++ Experts Forum in January 2001 at
`http://www.cuj.com/experts/1901/henney.htm`

Thanks to Kevlin for allowing us to reprint it.

Implementing the Observer Pattern in C++ - Part 1

by Phil Bass

Introduction

The Observer design pattern is described in the "Gang of Four" book [1] as a method of propagating state changes from a Subject to its Observers. The key feature of the pattern is that Observers register with the Subject via an abstract interface. The existence of the registration interface decouples the Subject from its Observers. This makes the Observer pattern well suited to a layered architecture in which a lower-level Subject passes information up to its Observers in the layer above.

This idea is so important that I have formulated it as a design principle:

Use the Observer Pattern to pass information from a lower layer to the layer above.

With this in mind I developed some C++ library components intended to support this use of the pattern. These components have been used extensively in my work at Isotek. They served us well for nearly two years, but recently we began to find situations in which they failed to deliver the ease-of-use we expected. The library itself seemed fine, but unexpected complexities started to arise in classes using the library components.

In this article I shall describe the Isotek library, illustrate the situations in which it is awkward to use and begin to explore ways of tackling its limitations. I hope the library will be of interest as a partial solution to the problem of implementing the Observer pattern. A complete solution, however, is left as an exercise for the reader - because I don't know what it is!

Library Overview

The Observer support library defines a Subject as any object that publishes Events. A Subject notifies its Observers of state changes by generating appropriate Events. A Subject may publish all state changes through a single Event, or provide separate Events for different sorts of state change. For example, a Button might publish a single Event that signals both button-pressed and button-released state changes, or it might provide both a button-pressed Event and a button-released Event.

An Observer registers with a Subject by attaching a suitable function (or function object) to an Event, and unregisters by detaching the function. The library supports functions taking one argument or none. In principle it could be extended to use functions with more arguments but, so far, we have not felt the need.

Conceptually, an Event is a container of polymorphic functions. Any function (or function object) that can be called with an argument of a particular type can be inserted into this container. So, for example, the following callback functions can all be inserted into an Event that generates an `int` value:

```
void f(int);  // natural signature
int  g(int);  // different return type
void h(long); // implicit argument
              //       conversion
```

Here is some sample code showing the use of the `Event<>` template:

```
#include <iostream>
#include "Event.hpp"
using namespace std;

// A simple Subject.
struct Button {
  enum State { released, pressed };

  Button() : state(released) {}
  void press() {
    stateChanged.notify(state=pressed);
  }
  void release() {
    stateChanged.notify(state=released);
  }

  State        state;
  Event<State> stateChanged;
};

ostream& operator<<(ostream& os,
          const Button::State& state) {
  return os << (state == Button::pressed
                ? "down" : "up");
}

// A callback function.
void output(Button::State state) {
  cout << "New state = "
       << state
       << endl;
}

// A sample program
int main() {
  Button button;

  cout << "Initial state = "
       << button.state << endl;

  button.stateChanged.attach(output);

  button.press();
  button.release();

  return 0;
}
```

The `Button` class is a Subject. It publishes a single state-changed Event. When the `Button::press()` function is called the button goes into the `Button::pressed` state and it publishes the state change by calling `Event<>::notify()`. Similarly, calling `Button::release()` causes another change of state and this, too, is published by calling `Event<>::notify`.

In this simple example a global function is attached to the button's state changed event. There are no Observer objects.

The Event<> Template Declaration

A slightly simplified version of the `Event<>` class template in the library is shown below.

```
template<typename Arg>
class Event {
public:
  // Iterator type definition.
  typedef ... iterator;

  // Destroy an Event.
  ~Event();

  // Attach a simple function to an
  // Event.
  template<typename Function>
  iterator attach(Function);

  // Attach a member function to an
  // Event.
  template<class Pointer,
           typename Member>
  iterator attach(Pointer, Member);

  // Detach a function from an Event.
  void detach(iterator);

  // Notify Observers that an Event
  // has occurred.
  void notify(Arg) const;

private:
  ...
};
```

The template takes a single type argument and the `notify()` function takes an argument of this type. Although not shown here, the library provides a specialisation for `Event<void>` in which the `notify()` function takes no argument.

There are two member template `attach()` functions. The first accepts a simple function; the second takes a pointer to an object and a pointer to a member function of that object. Both `attach()` function templates create a callback function object (stored on the heap) and insert a pointer to the callback into an internal list. This makes it possible to attach a non-member function, static member function, function object or member function to the `Event<>`. The only restriction is that the function takes an argument convertible to the `Arg` type of the Event.

The `detach()` function destroys the callback object specified by the iterator argument and removes the callback pointer from the list.

The `notify()` function simply iterates through the internal callback list calling each in turn, passing the parameter value (if any).

Finally, the destructor destroys any callback objects that are still attached.

The Event<> Template Definition

The Event implementation uses the External Polymorphism design pattern. The `Event<>` classes store a list of pointers to a function object base class and the `attach()` functions create callback objects of derived classes. The callbacks contain function pointers or function objects provided by the client code. The client-supplied objects need have no particular relationship to each other. In particular, there is no requirement for the client object types to be classes or to derive from a common base class. The callback classes perform the role of an Adapter (see [1]), in effect, adding run-time polymorphism to the client's function types.

The implementation described here is slightly simpler than the one in the Isotek library, but it does illustrate all the essential features of the library implementation. Note that the "`std::`" prefix has been omitted here to save space.

```
// Event class template
template<typename Arg>
class Event {

  // Function object base class.
  struct AbstractFunction;

  // Concrete function object classes.
  template<typename Function>
          class Callback;

public:
  // Iterator type definition.
  typedef list<
          AbstractFunction*>::iterator
          iterator;

  ...

private:
  // List of function objects.
  list<AbstractFunction*> callback;
};
```

The library uses the standard list class, which ensures that iterators are not invalidated by insertions/deletions and callback functions can be removed efficiently in any order. Both considerations are important for Subjects that can make no assumptions about their Observers.

`AbstractFunction` is a simple abstract base class with a pure virtual function call operator. The `Callback` classes are concrete classes derived from `AbstractFunction`. They store a function (as a pointer or function object) and implement the virtual function call operator by calling the stored the function. The same basic mechanism is used in Andrei Alexandrescu's generic functions [2]. The `Callback` template, however, is less sophisticated.

```
// The function object classes

// Abstract Function.
template<typename Arg>
struct Event<Arg>::AbstractFunction {
  virtual ~AbstractFunction() {}
  virtual void operator()(Arg) = 0;
};
```

```
// Callback class template.
template<typename Arg>
template<typename Function>
class Event<Arg>::Callback
           : public AbstractFunction {
public:
  explicit Callback(Function fn)
           : function(fn) {}

  virtual void operator()(Arg arg) {
     function(arg);
  }

private:
  Function function;
};
```

The `attach()` functions create a callback on the heap and insert a pointer to its base class into the list of function objects. In principle, only the single-argument `attach()` function is required; the two-argument version is provided for convenience. In practice, the client code attaches a member function much more frequently than a simple function, so there is considerable value in the convenience function.

```
// The attach() functions

// Attach a simple function to an Event.
template<typename Arg>
template<typename Fn>
Event<Arg>::iterator
              Event<Arg>::attach(Fn fn) {
  return callback.insert(callback.end(),
                  new Callback<Fn>(fn));
}

// Attach a member function to an Event.
template<typename Arg>
template<class P, typename M>
Event<Arg>::iterator
         Event<Arg>::attach(P pointer,
                             M member) {
  return attach(
       bind1st(mem_fun(member),pointer));
}
```

The `detach()` function destroys the callback and erases its pointer from the list.

```
// Detach a callback from an Event.
template<typename Arg>
void Event<Arg>::detach(
                iterator connection) {
  delete(*connection);
  callback.erase(connection);
}
```

Notifying observers is simply a case of calling each callback in the list with the supplied parameter (if any). The code in the library and presented here uses `std::for_each()` to iterate through the list. The function object required as the third parameter of `for_each()` is built from the `AbstractFunction`'s function call operator using `std::mem_fun()` and `std::bind2nd()`.

```
// Notify Observers that an Event has
// occurred.
template<typename Arg>
void Event<Arg>::notify(Arg arg) const {
  typedef AbstractFunction Base;

  for_each(callback.begin(),
    callback.end(), bind2nd(mem_fun(
                &Base::operator()),arg));
}
```

The final part of the `Event<>` template definition is its destructor. It just iterates through the callback list destroying any callbacks that remain. Again, the code uses `for_each()` and a simple function object is defined for use as its third parameter.

```
// Delete function object.
struct delete_object {
  template<typename Pointer>
  void operator()(Pointer pointer) {
    delete pointer;
  }
};
// Destroy an Event.
template<typename Arg>
Event<Arg>::~Event() {
  for_each(callback.begin(),
    callback.end(), delete_object());
}
```

So Far So Good, But...

When I first wrote the `Event<>` template I was aware that copying an Event could cause problems. Each Event contains a list of pointers that implicitly own the callback they point to. Copying this list produces two pointers for each callback, one in the original list and one in the copy. Destroying one of the lists destroys all the callbacks. Destroying the second list leads to disaster when the code attempts to destroy each of the callbacks again.

At the time, it wasn't clear to me how this situation should be handled. Should the copying of Events be prohibited or should more appropriate copy semantics be defined? In the end I left the issue un-addressed on the assumption that any problems would quickly surface and the hope that specific cases would throw more light on it. It seems I was wrong on both counts!

The real problems only surfaced when we wanted to store objects containing Events in standard containers. I shall describe that scenario in part 2.

Phil Bass

References

[1] Gamma, Helm, Johnson and Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN 0-201-63361-2.
[2] Andrei Alexandrescu, *Modern C++ Design*, Addison-Wesley, ISBN 0-201-70431-5.

Organising Source Code

by Allan Kelly

We've all seen it, one directory containing lots and lots of files. Where to start? Where is `main`? How do they fit together? This is the classic *Topsy system*, someone ran the wizard once and just kept on adding more and more files.

While it is simple to put all our files in one place we are missing an opportunity to communicate information about the system. The original designer may know these five files form a discrete group, and another six over there form another module but this isn't obvious, that information has been lost, recovering it takes time.

Consequently our understanding slows, and changes to the system are delayed – the software is resisting change. We need to split the system into comprehensible, logical, cohesive modules; we can then use the directory tree structure to convey the system structure.

Dividing our system across several directories has other advantages. It becomes easier for multiple developers to work on the system at the same time, and it becomes easier to transfer modules between multiple projects.

The directory structure of a project is closely linked with the source code control system employed – one mirrors the other. We cannot consider the layout of files without talking about the source code control too – once we commit files to source code control it becomes more difficult to move them to other directories or rename them.

Libraries, files, and directories represent the most physical manifestation of source code short of a printout. Consequently they form a key part of our overall strategy. Neglecting these aspects leads to blob-like software that lacks cohesion.

Splitting the system into modules

Many programming courses start by teaching their students the need to divide systems into modules. *Modularization* has moved beyond buzz-word status, we take for granted that it is a *good thing* and all systems should exhibit it. Some of the benefits cited usually include:

- *Comprehensibility*: while modularization helps us understand smaller elements of a system we need to be able to integrate this knowledge. Integrative knowledge is more difficult to express.
- *Division of labour*: if is easier for multiple developers to work on a modularised system then a monolithic one. We also get *specialisation of labour* where experts in one aspect of the system can work on one module, and other experts on other modules.
- Modularization should help focus our minds on cohesion, coupling, dependencies, division of tasks and such, this should all make the system easier to change.

Reuse is often cited as another advantage of modularization, given the current debate on reuse I won't cite it as an automatic benefit. However, if we wish to share elements between projects then there must be some division of the source code.

But what are our modules? *Module* and *modularization* are such overloaded words we're never really sure what they mean. *Component* has similar problems.

Individual files can be a module, but that is too fine grained for most purposes. And if a module is a file why use the word module? And what difference does it make?

One of our usual objectives in defining modules is that we wish to practice *information hiding*. It seems to me that the correct level to define our modules is the level at which we can actively hide some information, that is, hide some implementation.

Once our C or C++ code is compiled to object code we can hide the implementation since we only need distribute the header files and the object file. Still, the object file has a one-to-one relationship with the source file so we're not hiding much.

We need a bigger unit to hide in. When we bundle many object files together we get a static library. This is more promising. Our code can interface to the library by including one or more header files and we shouldn't need to care whether the library is made up of one file, two files, or 25.

Static libraries are simple to create and use. Once compilation and linking are complete then static libraries present no additional overheads or requirements, we can have as many of them as we want at no additional run-time cost. Hence, they are well suited to be building blocks when decomposing a system into discrete chunks.

Dynamic link libraries are more complicated, by no means are they simply "static libraries which have been linked differently." We must consider run-time issues, where are the libraries found? How do we find them? Do we have the right version? Dynamic libraries have their place but they should not be the basic building block.

When we create a static library we want to hide a secret inside the library. The secret is *implementation detail*. We want the library to represent an idea, and we want to hide the realisation of the idea from the rest of the system. To this end, the library needs to be highly cohesive, that is, it needs to express its ideas fully but no more than need be, it should not have lots of bells-and-whistles. The library also needs to pay attention to what it depends on, how connected it is to other modules in the system, that is, we want to minimise coupling.

We can't reason about the cohesion and coupling of every file, class and function in a system, that would take forever. While individual developers may consider these forces within the library module at a system level we would be overwhelmed by such details.

Static library modules represent the basic ideas from which a system is built. Since each one contains a complete idea we should expect to have many static libraries in our system. Many is good, it shows that your ideas are discrete and can be expressed individually.

Although you can cram more than one idea into a library you usually know when you are doing so. It is pretty obvious when the library is called "Logging" and you are putting database update functions in that something is wrong.

It is also possible to fracture an idea and split it across multiple libraries, but again it is pretty obvious. You quickly notice that library "Logging" always requires library "LogMessage" and something isn't quite right here.

Good systems are decomposed into many distinct static libraries – we should prepare for and encourage this. On top of the libraries we will find at least one application which results in an executable program. It may only comprise one file, a main.cpp, with the bulk of the code farmed out to static libraries.

You may well find that your project produces several applications, when this happens you can benefit from good modularization. There is no need for each application to provide its own logging system, you use the logging library.

Is this reuse? Well, that depends on your definition of reuse. I would argue that you are producing a family of programs with common characteristics for which you use common code. In time you may transfer some of this code to other projects.

How do we encourage modularization? Well, we start by providing a structure into which we can modularize our project. Since we will be writing files, we need a directory tree to place them in.

The Directory Tree

In the early days of a project we may like to work light, especially if there is just one developer on the project. But very quickly a project crosses a line, usually when a second developer starts work, or you decide that you could pull in code from a previous project. Once you've crossed this line you need to structure the work area of the project, that is, the directory layout.

Obviously we want a logical directory structure but we also want one we can add to. We need to be able to create sub-directories for new modules, and we don't want to get overwhelmed by directories. It is better to have many small modules, in many directories, than several "catch all" modules in a few directories. Above all, we need to give ourselves space.

We want to use the directory tree to partition the system into recognisable chunks: all the database files in one directory, all the logging files in another, and so on. When someone comes new to the system each chunk is clearly defined. Of course, they still have to understand the insides of each chunk, and how they fit together but you are not faced with one overwhelming mass of files.

The directory structure we use should map directly into the structure used in our source code control system. The two are closely intertwined, and although I've tried to separate the rationale for each I can't, there is one hierarchy and it should apply in the directory tree and in the source code control tree.

Some control systems allow us to break this link and check files out to different locations. On occasions this can be useful but if you find you need to do this regularly you should consider why. Your structure is lacking something. Even with the best intentions breaking the link becomes troublesome, in the long run it is better to come up with a solution which does not break the link between directory hierarchy and source code hierarchy.

Where is the root?

All trees need to be rooted somewhere and software trees are no different. If we always refer to our directories by relative paths we need not care where they are rooted. However, experience shows that this eventually breaks down, at sometime we need to refer to the absolute location of files. It is a lot easier to reason about a path like `/home/allan/develop/lib/include/logging` than to reason about `../../../include/logging` – try shouting the latter across the room.

In the Unix world there is only one ultimate root, `/`, but we all have local roots, e.g. `/home/allan`. Usually our trees are rooted in our home directory but not always. Typically we set an environment variable to the point where our tree is rooted and append from there, so we get `PROJECT_ROOT=/home/allan/develop`, and `$PROJECT_ROOT/lib/include/logging`.

(I was confused for far too long over Unix environment variables, sometimes they just wouldn't work for me. What I was failing to appreciate is that there are two types: shell variables, and exported variables.)

In the one-root-per-disc world of Microsoft things are a little more complex. Traditionally I would use the `subst` command to create a virtual drive on my machine, this I could point wherever I liked – it is worth putting this command in a start up script.

```
subst w: d:\develop
```

Thus, each developer can have their directory tree where they like, their C: drive, or D:, at the root, or within another tree.

More recently I've moved over to the Unix way of doing things even in the Microsoft world. As it happens .dsp project files are happy to pick up environment variables so you can use the same technique as in Unix.

Unfortunately, Microsoft has made environment variables a lot more hassle under Windows than Unix. Unix is simple: set them in your shell .rc file and change them at the command line. The .rc file can be copied, e-mailed and edited easily. Under Windows, you need to go fiddling in the control panel, and the location seems to move slightly with each version of Windows.

This may seem like a lot of unnecessary work but it pays for itself if you ever need to maintain two different development trees on the same machine, say a maintenance version 2.1.x and the new development 3.0.

The External Tree

It is it increasingly unusual to find a system that doesn't use any third party code or libraries. Whether these are commercial

Source Code Control

The fact that the directory structure outlined above maps well to a source code control system is no accident. Which came first? The two are closely intertwined, I've tried to separate the rationale between the two before – and failed, the two are the same.

Some source code control systems, for example, Visual Source Safe, allow you to put your source files in a check-out directory which is different to the source control directory. For each Source Safe repository folder containing source files, there is a *working directory,* but you are free to have a working directory hierarchy that is completely different to the folders directory. This doesn't work, there are not two hierarchies, there is one, why should the hierarchy under control be different to the one on your hard disk?

Normally, left to our own devices we usually keep the two hierarchies in synch. On occasions though we face pressures that may lead us to break the link. This frequently happens when it is necessary to share files between two groups. Even with the best intentions this becomes troublesome, in the long run it is better to come up with a solution that does not break the link between directory hierarchy and source code hierarchy.

Putting code in a source code control system does remove some flexibility. I've yet to find a system which is good at renaming and moving files. Suppose we have a static library `Toys.lib`, and suppose we decide to make it an optional DLL, `Toys.dll`, so we need to change its position in the tree. Most of the files can be used "as is" but it is surprisingly difficult to move them, we may end up adding them as completely new files and losing their history.

In fact, this opens up a more perplexing question: just how should we treat those files? Sometimes we may be happy to treat them as new, erase the long history that is getting in the way, sometimes we may want to explicitly state the relationship.

The problem can be particularly bad when a file changes over time, quite legitimately the code comes to represent something which isn't reflected in the name. Should we change the name? The current name may be confusing but to change it would lose its history.

Worse still is what happens when we delete files. We want them gone for good reason, but they must exist somehow so we can reference what happened.

I can't provide fixed answers for these questions. Each case is different, influenced by your source control system, your environment and your team's attitude.

libraries like RogueWave, Open Source projects like Xerces or future standards like Boost we are increasingly dependent on code we have not created.

It is important to differentiate between in-house code, which we have created and own the rights to, and code which is external to our organisation. The development cycle for these two sources of code is very different. Our own code is changing according to a cycle which we dictate, the third party code changes whether we want it to or not, of course, we decide if and when we accept these changes so in the meantime the code is static.

Once we've decided on our root, and how we refer to it, we need to split our directory tree to show what is ours and what is not. Typically this means we create an *External* directory tree which contains third party code, e.g. `/home/allan/develop/external`, while our own code goes in a separate tree such as `/home/allan/develop/ProjectFire`. (Sometimes the External tree is called "third party" tree, but this gets confused, is it: 3rdParty, or ThirdParty, or 3Party, where are the capitals?)

Sometimes we find one external tree can be used for several projects. This may mean we have multiple versions of the same product in the tree, say Xerces 1.6 and Xerces 1.7. We have two solutions here: one is to include everything in one external tree and reference what we need, the second is multiple external trees, we may have `/home/allan/develop/external/v2.1` and `/home/allan/develop/external/v3.0`. Which way you jump depends on your project requirements.

It is not normally a good idea to try and delta one version of an external product on top of a previous version. The complications of new files, removed files and renamed files usually make this a lot of effort for little reward. It is simpler to just allow multiple versions in the tree.

On my current project I have environment variables for almost everything, so I have one external tree and within that I can point `XERCES_ROOT` to the version of Xerces I need.

The other half of the development tree is your own source code. You'll probably find this is dwarfed by the third party code, I normally find that even on my biggest projects a Zip file of our code will happily fit on a floppy disc but the third party stuff needs its own CD.

Of course, life isn't quite this straightforward, you may well have code from elsewhere in your organisation, perhaps this is supplied by the *infrastructure team*, or the *client application group* or the New York office. Sometime you'll want to share the tree with them, sometimes you'll want to treat them as external code, or maybe you split your tree three ways: external, enterprise and team. It depends on how much you need to isolate yourself from these developers.

Align libraries with namespaces

If you're following my guidelines from above the chances are you've got at least one executable application, several DLLs and lots of static libraries. Each application, DLL and library needs its own space – that is: its own directory. Only by giving them their own directories can you explicitly spell out what belongs where.

Where static libraries are concerned you need to split the files into two directories. Most, of the code in a static library is contained in .cpp files (or .c, or .cxx or whatever.) This is implementation detail. You want to put this out of the way – out of sight, out of mind. However, the interface to the library is going to be expose in a set of .h files, these need to be put somewhere publicly visible, somewhere obvious, somewhere well known.

Traditionally we do this by creating a lib subdirectory, inside this we would have:

```
/lib
    /include
    /Logging
    /AccessControl
```

(Although most OS's now allow you to have spaces in directory and filenames they are still best avoided, they complicate matters.)

In our `lib/Logging` directory we would put all files implementing our logging system, the files exposing the public interface would be put in the `lib/include` directory where we can all find them. Similarly, `lib/AccessControl` contains the implementation for our access system, and the public interface files are also put in `lib/include`.

Using one include directory we quickly fill it with a lot of disparate files which adds nothing to our structural information. We could leave them with the implementation files – but now we can't tell what is a private, implementation only header file, and what is a public interface file.

Alternatively, we could put them all in separate directories but this could mean we end up with lots and lots of `-I` options on the link line, imaging:

```
gcc -I $PROJECT_ROOT/libs/include/Logging
    -I $PROJECT_ROOT/libs/include/AccessControl
    ....
```

Well who cares what it looks like? Does it really matter? No, but, each time you add a new library you need to change your makefile to specify the new include directory.

A better solution is to specify one root include directory, and within our code specify the actual library we're interested in, hence we get

```
gcc -I $PROJECT_ROOT/libs/include
```

and

```
// main.cpp
#include "Logging/LogManager.hpp"
#include "AccessControl/AccessOptions.hpp"
```

The real power of this comes when we align with namespaces. So, each library has its own namespace and the namespace corresponds to the sub-directory. Continuing the above example we get:

```
...
int main(int argc, char* argv[]) {
  Logging::Logger log;
  AccessControl::User user(argv[1]);
  ...
}
```

Looking back to the source tree, we should also think about balancing it is little bit, it now seems a little lop-sided, so we get:

```
/lib
    /include
        /AccessControl
        /Logging
        /Utils
    /source
        /AccessControl
        /Logging
        /Utils
```

The extra level of directories may seem surplus but actually helps space the project quite well. When you put this altogether you get a very powerful technique for managing your files and module structure, it really pays off.

A couple of points to note however: firstly, not all header files will go in the `/include` directories. Some don't represent an interface to the library, they are not intended for public use so they should only exist in the `/source` directories. It is good to make a clear distinction between what is available to the general public and what is considered local implementation detail.

Second, I've taken to prefixing my header guards with the namespace name as well, so:

```
  #ifndef ACCESSMGR_HPP
```

becomes:

```
  #ifndef ACCESSCONTROL_ACCESSMGR_HPP
```

Once you're free of thinking up completely unique filenames you quickly find several `Factory.hpp` files appearing, it doesn't really matter because one will be `Logging/Factory` and one will be `AccessControl/Factory`. However, some debuggers (OK, I'm thinking of one from Redmond) can have problems telling the files apart.

If you already think of namespaces as program modules this technique may be obvious, if you're still thinking of namespaces as a convenient way to avoid name clashes then you haven't realised their full potential yet. A namespace is a C++ language module; a static library is the natural corollary.

Lastly, and fairly obviously, this doesn't quite apply to template files where the implementation must be exposed in the same file as the interface. I'll still tend to stick with my solution and place them in the `lib/include/xxx` directory because nowhere else really makes sense. Hopefully, in time, the C++ compiler vendors will resolve this problem, but in the meantime systems where the majority of the code is templates are fairly rare.

DLLs

DLLs should be treated just like static libraries, that is, I give them a DLL directory and split this into source and includes. DLLs are different from static libraries, the linkage rules are different, they are used for different reasons – some people regard use of one over the other as fairly arbitrary but with experience you come to see them as two very different beasts.

One of my personal rules is to avoid subtlety in design. Let's call it Kelly's Law of Software Subtlety:

Subtlety is bad
If it is different make it obvious - Write it BIG

Since DLLs are different to libraries put them somewhere else.

As your system grows you can easily find you have 10 or more static libraries to manage and perhaps another 3 or 4 DLLs. If we separate out DLLs, we give ourselves more space. There is a natural difference here so use it.

We need to add another `-I` to our compile line, but it is just one more, the same namespace alignment scheme can be used for DLLs provided you don't need to specify C linkage for any of your functions.

Applications

Even if your objective is to produce just one final application, say, `server.exe`, the chances are you will end up with more than one executable, even if all but one are trivial. Executables are the top of the food chain when it comes to source code so it pays to put them there - straight off the project root.

If however, your project is going to produce many applications you may want to avoid littering the project root with lots of directories. In this case create a `project_root/app` directory and create a sub-directory for each on there.

There is no need to split header files from source files because the application does not expose its internals like this. If you find you need to access application files, and you see things like `#include "../otherapp/widget.hpp"` appearing in your code it is an indication that there is some shared functionality that belongs in a library where it is readily accessible to all. The directory tree is highlighting a refactoring needed in your system.

On some projects you may find that one or more applications are large enough to warrant being broken into libraries by themselves. If so then don't hesitate, go for it!

Each one becomes a mini-project in its own right, apply the tree design I've just outlined to each application. You'll probably find some common libraries shared between them all, they stay where they are, but for a large application it should have its own library structure. This is just recursion, apply the pattern over again taking the application's directory as your new root.

Putting it together

You'll probably find that you have other stuff which needs to be in the tree, makefiles, documentation and such. Where these belong to a component, say documentation for your logging library, then place the documents same directory as the library. Where they are common, say system documentation, place them in a docs directory from the root – if necessary divide it.

In the case of makefiles you'll find that some are common and need to be placed in a well known place in the tree, but most are specific to individual elements – indeed each library, dll, application, should have its own.

Your source tree should now be looking something like this:

```
  $project_root
      /apps
          /repairtool
          /server
          /testtool
      /dlls
          /include
              /BlueWidgets
              /RedWidgets
          /source
              /BlueWidgets
              /RedWidget
          /docs
      /libs
          /include
              /AccessControl
              /Logging
              /Utils
          /source
              /AccessControl
              /Logging
              /Utils
      /make
```

At first sight this may seem a bit excessive, but the important point is it gives you space, it gives you organisation, you are free to do what you like in any sub-directory and it won't interfere with another. This model will scale, we are building in depth for extendibility.

[concluded at foot of next page]

File Format Conversion Using Templates and Type Collections

by Richard Blundell

A recent project involved upgrading some files from an old format to a new one (and possibly back again), as a result of changes to the data types being stored. Several possible implementations were considered. The final solution made use of template methods, and type-collection classes, and supported forward and backward file format conversion with no code duplication and minimal overhead.

Requirements

Many years ago, one of our projects was converted from a 16-bit application running on Windows 3.11 to a 32-bit one running on Win32. Most of the code was ported at the time, but some changes were not made because they would have required a change to the file formats. 16-bit identifier values were being stored in a file. Changing the file format was seen as too much of an upheaval (especially at a time when so many other changes were being made). And besides, 16-bits should be enough for anyone...

Time passed. Suddenly, 16-bits were no longer enough everyone. The file format needed to be upgraded. Discussions were had, and the following requirements emerged:

1 The old version of the software would not be required to read the new file format (i.e. no forwards compatibility – see [2]).
2 The new version of the software was required to use the new format (obviously) but only had to recognise the old format, and prompt the user to upgrade (i.e. limited backwards compatibility – see [2]).
3 An upgrade utility would convert from the old format to the new format. A ‘downgrade’ facility would be a ‘nice-to-have’ (just in case users were running both software versions on site and upgraded the wrong site by mistake) but was not a necessity.
4 The interfaces of the data classes should be changed as little as possible.
5 Any solution should support future changes (we don’t want to have to re-implement everything when it comes to 64-bits).

Initial suggestions

Support for the new format in the software, and for both formats in the upgrade utility, required old and new versions of the persistence code for the data types involved, as well as some form of user-interface for the upgrade utility, and logic for converting the files as a whole. Suggestions were put forward for tackling the serialisation issues:

1 Copy the old serialisation source code to the upgrade tool project, modify the original code to use the new format so the application can read and write the new files, and include this modified code in the upgrade utility as well. The upgrade utility would therefore have code for both the old and new formats, and the application would have only the new code.

[continued on next page]

External tree and source control

Nor do I have a good answer for the question “Do we check in the external tree?” There are three possible answers here:

1 Check nothing in: the code has come from elsewhere, you can always get it again, download it, install from CD. Maybe you want to burn some CDs with downloads on so you can always get old versions. Of course, what happens when you fix a bug in external code? I once found a fault in the ACE library, I devised a fix and contributed it back but it was several months before a version of ACE was released with my fix. In the meantime we checked-in my fix to our ACE copy in CVS and carried on working.
2 Check in source code: if you have the source code you can recreate the binaries, this will save people having to locate several dozen different resources. And it helps when you make a change to external code. However, each developer needs their own copies of the binaries - access over a network to a common store can substantially slow compile time - but it can be quite time consuming to build lots of third party libraries.
3 Check in source and binaries: source control systems aren’t really built for binaries and they rapidly bloat when you check in everything, and if we do then developers need even bigger hard discs to get trees containing lots of stuff they don’t actually want.

Nor does the external tree contain all the third party products in your system. Do you check in your compiler? Perl? Your OS?

Source control is not confined to your source code control system, it should encompass all the elements needed by your system: compilers, OS, patches, etc, etc. It is unrealistic to put everything under source control so you need to look elsewhere.

The best solution I know to this dilemma is:

1 Only check in source code, this means you can tinker with Boost, ACE, or whatever as you need to.
2 Build the binaries under clean conditions – as you would your own source but place it somewhere generally available, say a Samba drive. This is a kind of cache and ensures that everyone is using the same thing and saves everyone rebuilding it. Some people may care to place some of this stuff on their local hard disc.
3 Burn CD copies of all third party product used to build your system, be they libraries, compilers or whatever, and put them in a safe, well known location.

The objective is: to be able to take a new machine and only use what is in the cupboard and source code control be able to build your entire system.

Conclusion

Our directory structure affects our source code. At the simplest level this is shown in our makefiles, makefiles are the glue that links the two, makefiles explain how to integrate the system. Directory trees always have an effect on code, rather than hide this detail we should use it to our advantage. The directory tree can encode important details about the system structure and organisation, and its extension points.

Much comes down to your work environment, your schedule, your team, but these issues are important. The ideas I’ve laid out come up again and again, each team will have slightly different needs but all solutions exhibit a general similarity.

Above all you need to actively organise your source code, external tools and resources. These things don’t just happen. This organisation is an integral part of your software development strategy and helps communication.

Allan Kelly
Allan.Kelly@bigfoot.com

2 Append methods supporting the new formats to all the affected data classes. The application would use the new format only, and the upgrade utility would use both.
3 Modify the serialisation methods to handle both formats, determining which one to use with some form of flag or version number.

Drawbacks

The first suggestion set warning bells ringing left right and centre. Every time I have ever copied code around it has come back to haunt me. When the same, or similar, code is in two places you have twice as much code to manage. Changes need to be made in two places instead of one, which is highly error-prone. Furthermore, people inevitably forget about one or other of the copies, and so it gets out of date, it doesn't get built properly, documentation stagnates, and it causes endless confusion to new team members when they stumble across it. Re-use good; copy-and-paste bad!

However, criticisms were levelled at the second suggestion too. The application would need to cart around both old and new serialisation code, despite only ever using the new code. Small classes would find the majority of their source code comprising multiple persistence methods. Changes and fixes would still need to be made to both versions. Even if they sit right next to each other in the source file it is easy to miss one when editing the code through a tiny keyhole source code window[1].

Finally, the third suggestion leads to spaghetti serialisation code, with huge conditional blocks based on ever-more complicated version dependencies. In later versions you have a mess of if blocks checking for file formats that have not been supported for years [2]. As with the previous suggestion lean classes become fat with persistence methods.

Types, typedefs and templates

In our project we were making no changes other than the types and sizes of various data values. Instead of a version flag, why not parameterise the persistence methods on the relevant types? This way we can support a whole raft of file formats using different types all with the same code. Simple wrapper methods can then be written to forward to the parameterised method with the correct types.

As a rather trivial example, consider the code for a class that stores an array of id values (see [1]).

```
// id_array.h
class id_array {
  ...
  short m_size; // should be plenty...
  short *m_ids; // should be wide enough
};

// id_array.cpp
void id_array::extract(out_file &f) const
{
  f << m_size;  // raw write, 16-bits
  for (short i = 0; i != m_size; ++i)
    f << m_ids[i];
}
```

```
void id_array::build(in_file &f) {
  short size;
  f >> size;  // raw read of 16-bits
  resize(size);
  for (short i = 0; i != size; ++i)
    f >> m_ids[i];
}
```

This class is limited in a number of ways for our purposes. `m_ids` only holds 16-bit short ints, and we wish to extend this to 32-bit regular ints (on our platform). We also wish to be able to hold lots of these, so a 16-bit container size is also insufficient. Finally, an unsigned type would be more appropriate for the size of the container.

Our first step is to parameterise our persistence methods:

```
template <typename T>
void id_array::extractT(out_file &f)
                                const {
  T size = m_size;
  f << size;

  for (T i = 0; i != m_size; ++i) {
    T value = m_ids[i];
    f << value;
  }
}

template <typename T>
void id_array::buildT(in_file &f) {
  T size;
  f >> size;
  resize(size);

  for (T i = 0; i != size; ++i) {
    T value;
    f >> value;
    m_ids[i] = value;
  }
}
```

As you can see, there is very little change to the code. The two methods are prefixed with a template declaration containing the type required. This type is then used inside the methods. One point worth noting here is that the type must be used in any overloaded function calls rather than the data members from the class itself. Writing `f << m_size;` will output `m_size` as the type defined in the class itself, rather than the required type `T`. Hence you must write `T size = m_size; f << size;` instead. Easy to overlook, that one (he says from experience :-)[2].

Explosion of types

It soon becomes clear that, strictly, we should have parameterised the class both on the capacity and the contained type, because these are not necessarily the same. Thus, our class is now parameterised on two types:

1 which is all the space you seem to be left with, these days, in between the project windows, watch windows, output windows, toolbars, palette windows, etc., of the modern IDE.

2 But fortunately one that is easy to spot when the automated unit tests, which of course you wrote first, fall over.

```
template <typename Count, typename T>
void id_array::extractT(out_file &f) const{
  Count size = m_size;
  f << size;
  for (Count i = 0; i != m_size; ++i) {
    T value = m_ids[i];
    f << value;
  }
}
template <typename Count, typename T>
void id_array::buildT(in_file &f) {
  Count size;
  f >> size;
  resize(size);
  for (Count i = 0; i != size; ++i) {
    T value;
    f >> value;
    m_ids[i] = value;
  }
}
```

More complicated data structures may have even more types, and when you have many such low-level data types you can end up with a huge number of types and a huge number of different parameters to each method. It gets nasty very quickly.

Classes of types

What we really want is to be able to say, "My old file format used types `t1, t2, …, tn`, whereas in my new format I use types `T1, T2, ..., Tn`." It would be nice to be able to group these relevant types together so you can just say "new format" or "old format" rather than "short, unsigned short, int and short" to one method and something else to another. Enter the class as a method of naming things as a group:

```
// format_types.h
class old_types {
public:
  typedef short count_t;
  typedef short my_id_t;
  ... // lots more follow, if nec.
};
class new_types {
public:
  typedef size_t count_t;
  typedef int my_id_t;
  ... // lots more...
};
```

Now, rather than passing in as many parameters as each class requires, persistence methods can be parameterised solely on a single format type. These methods then pull out whatever named types they require from the file format 'types class':

```
template <typename Format>
void id_array::extractT(out_file &f) const{
  Format::count_t size = m_size;
  f << size;
  for (Format::count_t i = 0;
                          i != size; ++i) {
    Format::my_id_t value = m_ids[i];
    f << value;
  }
}
```

```
template <typename Format>
void id_array::buildT(in_file &f) {
  Format::count_t size;
  f >> size;
  resize(size);
  for (Format::count_t i = 0;
                          i != size; ++i) {
    Format::my_id_t value;
    f >> value;
    m_ids[i] = value;
  }
}
```

Forwarding functions

We did not want to alter the interfaces of the data classes more than necessary. In particular, we wanted persistence from our main application to work exactly as before. To achieve this we created one more typedef for the types currently in use:

```
// format_types.h
// current_types points to new_types
//   now (not old_types)
typedef new_types current_types;
...
```

and wrote forwarding functions to call the buildT() and extractT() template methods with the correct types:

```
// id_array.h
class id_array {
public:
  // these are the original method names
  void extract(out_file &f) const;
  void build(in_file &f);
  // these are new forwarding methods
  void extract_old(out_file &f) const;
  void extract_new(out_file &f) const;
  void build_old(in_file &f);
  void build_new(in_file &f);

private:
  // These are the implementations
  template<typename Format>
  void extractT(out_file &f) const;
  template<typename Format>
  void buildT(in_file &f);
};
```

We then implemented these forwarding methods:

```
void extract(out_file &f) const {
  extractT<current_types>(s);
}
void build(in_file &f) const {
  buildT<current_types>(s);
}
void extract_old(out_file &f) const {
  extractT<old_types>(s);
}
...  // etc.
```

These are all just one-liners, making it trivial to implement and maintain.

New formats

If a new format is required in the future (64-bits, etc.) supporting it is simple:

1 Add code to the unit test class to check that the new format works OK.
2 Add a new types class, `really_new_types`, containing the relevant typedefs.
3 Add one-line forwarding methods to each class to pass this types class in.
4 Update `current_types` to point to the new types class, `really_new_types`.
5 Build and check that your unit tests pass, to ensure the single persistence methods are sufficiently generalised to support the new types.

If you want you can omit step 3 and expose public templated serialisation methods. That way, clients can use any file format they choose by calling the method with the correct types class. We did not do this, (a) to control access to the different formats more closely, and (b) because our compiler, Visual C++ 7 (the latest .NET version) requires template methods to be implemented inline, which we did not want to do. Some of our persistence methods were quite involved. Implementing them in the header files could have introduced additional compilation dependencies from extra #include directives being required.

Our workaround involved declaring a private friend helper class at the top of each data class:

```
// id_array.h
class id_array {
  class persister;
  friend class id_array::persister;
public:
...
};
```

Class `persister` then simply had two methods: the two template persistence methods moved from the main class:

```
// id_array.cpp
class id_array::persister {
public:
  template<typename Format>
  static void extractT(const id_array &a,
                       out_file &f) {
    ...  // inline because of VC++7
  }
  template<typename Format>
  static void buildT(id_array &a,
                     out_file &f) {
    ...  // inline because of VC++7
  }
};
```

The use of this private helper class allowed us to move the inline implementations of these template methods out of the header file. Making it a nested class avoided name clashes because we were not polluting the scope of our data classes with additional names (and therefore each class could use the same nested class name, `persister`). The forwarding methods within each data class could now simply forward to the static methods of class `persister`, passing in a reference to themselves:

```
// id_array.cpp
void id_array::extract(out_file &f) const {
  persister::extractT<current_types>(
                                *this, f);
}
...  // etc.
```

Alas we were not quite in the clear yet. Another weakness of VC++7 is that it does not support explicit specification of template parameters for template methods/functions. We had to work around this one as well by passing in a dummy object to each method and letting the compiler sort out which function to call:

```
// id_array.cpp
class id_array::persister {
public:
  template<typename Format>
  static void extractT(const Format &,
                       const id_array &a,
                       out_file &f) {
    ...
  }
  ...
};
...

void id_array::extract(out_file &f) const {
  id_array::persister::extractT(
              current_types(), *this, f);
}
...
```

Conclusion

Classes were used as a scope to package up the whole set of types, used when serialising to a given file format, into a ‘types class’. A typedef was provided to allow `current_types` always to refer to the primary types class, and hence the current file format. Template serialisation methods were used to localise a single serialisation algorithm for each class in a single place to aid implementation and maintenance. One-line (non-template) forwarding methods were used to provide an easy interface to the current, old, and new file formats. And finally the use of a private nested friend class and dummy template function parameters allowed us to work around various weaknesses in the Microsoft C++ compiler and to move our templated persistence methods out of the header files.

None of these choices were rocket science, but the end result was a seamless implementation of multi-format persistence with very little overhead, either overall (just the format classes were needed) or in each of the persisted classes.

Richard Blundell

References

[1] Blundell, R.P., “A Simple Model for Object Persistence Using the Standard Library,” *Overload 32*, June 1999
[2] Blundell, R.P., “Automatic Object Versioning for Forward and Backward File Format Compatibility,” *Overload 35*, January 2000

Developing a C++ Unit Testing Framework

by John Crickett

Introduction

Testing is an important part of all software development, yet a part that is so often overlooked or skimped on. Why is this? Perhaps it's because testing software is not considered exciting, or perhaps it's because it's not trivial, and if we're honest with ourselves it's impossible to write a set of tests that's perfect, the only way of knowing a test works is if it shows that the software doesn't. Passing all the tests does not mean the software is perfect, it may mean your tests just aren't good enough.

As the title suggests I'm going to look solely at unit testing, as it's currently something I'm focused on, having adopted Extreme Programming (XP) [1]. So what is unit testing? To us it's testing individual software units to prove that they work as specified. In our case, tests may well form part of that specification, as XP is pro Test First programming, whereby we write our tests before writing the code, and we stop writing the code when all the tests pass. As such tests are both a way of ensuring quality and a way of measuring the status of our development. It's always worth remembering (and reminding the boss) that the sooner problems are found, the cheaper they are to fix. Tests formalise the testing process, they move it from just running the debugger and seeing what happens, to a structured, repeatable process. To encourage regular running of the tests, they should be quick and easy to use, in other words, fully automated. This will help you to get unit testing accepted as part of your personal and company development culture.

Prerequisites

Ok, so you see a benefit to unit testing, or at least I assume you do otherwise you'd probably have stopped reading by now. What do we need to begin developing effective unit tests? Happily not much, consider the following example:

```
bool isZero(long value) {
  return value == 0;
}
```

Which we can test with the following code:

```
#include <iostream>

int main() {
  if ((isZero(0)) && (!isZero(1))) {
    std::cout << "Passed." << std::endl;
  }
  else {
    std::cout << "Failed." << std::endl;
  }
}
```

However not all code is as simple to test, nor do we want to have to repeat the basic features that all tests share. The ability to report the success or failure of the test is the most obvious feature that we'll be requiring again. We'll also probably want to test the function with more than two inputs, and a massive conditional statement is not clean, maintainable and readable. It would be very nice to know which part of the test failed, and which resulted in an error. It would be useful if we were able to run the tests on application start-up, choosing to run all the tests or a specified subset. Finally, unlike many C++ testing frameworks, we won't assume that the user will only ever test classes.

Building our Framework

Let's start by changing the code to allow for our first prerequisite, which is to easily allow testing with multiple inputs. Firstly we'll add a function to determine if the test succeeded, and record the result:

```
void Test(bool eval, bool& result) {
  if (result) {
    result = eval;
  }
}
```

I've chosen to implement it this way so a pass after a failure will not overwrite the failure, in other words if `result` is already false we don't lose the failure when the next test passes. We can then change `main` to the following:

```
int main() {
  long const max_test = 100;
  bool result = false;

  Test(isZero(0), result);

  for (int i = 1; i < max_test; ++i) {
    Test(!isZero(i), result);
  }

  if (result) {
    std::cout << "Passed." << std::endl;
  }
  else {
    std::cout << "Failed." << std::endl;
  }
}
```

Which allows us to test the function for potentially all possible positive values of a long (if we changed `max_test` to be `std::numeric_limits<long>::max()`). Ok, so we're now able to run multiple tests, but should it fail, it would be very helpful to know which part failed. So how could we do that? Well, we could stop on the first failure, but we probably don't want to stop the entire unit test, so it's time to break our test down a little and practice some of our procedural/object-oriented design. We can start by changing the result to a structure as follows:

```
struct test_result {
  bool passed;
  unsigned long line;
  std::string file;
};
```

We will then change the test function to set these additional values:

```
void Test(bool eval,
          const char* file,
          unsigned long line,
          test_result& result) {
  result.passed = eval;

  if (!result.passed) {
    result.file = file;
    result.line = line;
  }
}
```

We'll also need to call the function differently (I've changed it to only print out failures, to reduce the clutter, and introduced a failed test) so our final program becomes:

```
#include <iostream>
#include <string>

// . . . code . . .

int main() {
  long const max_test = 100;
  test_result results[max_test];

  Test(!isZero(0), __FILE__, __LINE__,
       results[0]); // fails 0 is zero!

  for (int i = 1; i < max_test; ++i) {
    Test(!isZero(i), __FILE__, __LINE__,
         results[i]);
  }

  for (int i = 0; i < max_test; ++i) {
    if (!results[i].passed) {
      std::cout << "Test failed in file "
        << results[i].file << " on line "
        << results[i].line << std::endl;
    }
  }
}
```

Ok, so far so good, however it's rather tedious to have to add `__FILE__`, `__LINE__` to each call, and not terribly pretty either, so I'm going to pull them out, and use a macro (don't look so horrified) to save us the effort. We'll call the macro `ASSERT_TEST()`, just because that's a common naming style in testing frameworks. We'll define it as so:

```
#define ASSERT_TEST(condition, result)
Test(condition, __FILE__, __LINE__, result)
```

However, having decided to use a macro we can now use a bit of magic to get the actual code that failed and print that as part of our diagnostics, so here's the new macro, with the `test_result` structure changed to accommodate the new information and the `Test()` function renamed to `assertImpl()`:

```
#include <iostream>
#include <string>

struct test_result {
  bool passed;
  unsigned long line;
  std::string file;
  std::string code;
};

bool isZero(long value) {
  return value == 0;
}

void assertImpl(bool eval, char* code,
               char* file, unsigned long line,
               test_result& result) {
  result.passed = eval;

  if (!result.passed) {
    result.file = file;
    result.line = line;
    result.code = code;
  }
}

#define ASSERT_TEST(condition, result) \
assertImpl(condition, #condition, \
__FILE__, __LINE__, result)

int main() {
  long const max_test = 100;
  test_result results[max_test];

  ASSERT_TEST(!isZero(0), results[0]);
      // fails 0 is zero!

  for (int i = 1; i < max_test; ++i) {
    ASSERT_TEST(!isZero(i), results[i]);
  }
  for (int i = 0; i < max_test; ++i) {
    if (!results[i].passed) {
      std::cout << "Test "
                << results[i].code
                << " failed in file "
                << results[i].file
                << " on line "
                << results[i].line
                << std::endl;
    }
  }
}
```

Having used our little bit of macro magic (for those that feel it's voodoo read [2] page 90), it's time to start thinking about how we can scale this up.

Refactoring into Classes

At this stage it is worth reviewing our requirements for a C++ testing framework:

- We need to know if a test failed.
- If a test failed we want the code for the test, the name of the file it is in, and the line it is on.
- We want to be able to test multiple conditions in each test, failure of any single condition is a failure for the test.
- Failed test(s) should not stop the rest of the test running.
- We want a report on the test results, after all tests have run.
- We may need to be able to set up some data before a test, and destroy it afterwards.
- We should cope with exceptions.
- The testing framework must be easy to use, as part of which we will implement our code in the `test` namespace.

We might also want the following information at some stage in the future:

- Duration of each test.
- Free/Used memory before and after the test.
- Log results to file.

Ok, lets take these one at a time starting with the easiest, the test result. We've already solved this the easy way with our `test_result` structure. So we'll dive right in and do the whole lot:

```
class TestResultCollection {
public:
  void error(const std::string& err);
  void fail(const std::string& code,
         const char* file, size_t line);
  unsigned long failedCount() const;
  void reportError(std::ostream& out) const;
  void reportFailures(std::ostream& out) const;

private:
  class TestResult {
  public:
    explicit TestResult(
              const std::string& code,
              const char* file, size_t line);
    void report(std::ostream& out) const;

  private:
    std::string code_;
    std::string file_;
    unsigned long line_;
  };

  typedef std::list<TestResult>
                          results_collection;
  typedef results_collection::iterator
                          iterator;
  typedef results_collection::const_iterator
                          const_iterator;

  results_collection results_;
  std::string error_;
};
```

To save space I'm not going to detail each function here, as a full description is available in the code [3].

As each test class may evaluate multiple expressions we're going to need to store more than one result for each test, as such we're going to use the class `TestResultCollection` to provide the interface for a test's results. It in turn will store a `TestResult` class for each failure, or error (we don't record passes; they are determined by the absence of a failure).

An error is defined as an exception that escapes the test code, or a failure of the `setUp()` or `tearDown()` methods, which are explained later. A failure is an expression that evaluates to false inside an `IS_TRUE()`, or an expression that evaluates to true within an `IS_FALSE()`. `IS_TRUE` and `IS_FALSE` are explained later.

Next we need a base class for tests, which will define our common interface to each test:

```
class Testable {
public:
  Testable(const std::string& name);
  virtual ~Testable() = 0;
  virtual bool setUp();
  virtual void run();
  virtual bool tearDown();
  virtual std::string name();
};
```

The constructor requires a name argument, this name will be returned by the member function `name()` and should be a human friendly name for this test class, as it is only used for reporting.

The four member functions have been designed to allow overriding in order to allow the client to perform appropriate action for each class. `setUp()` should be used to prepare any data required for the test, the body of the tests should be in `run()`, and `tearDown()` should tidy up any resources allocated in `setUp()`. The function `name()` returns the name provided to the constructor. Each function has a default implementation provided.

The class also contains the protected member `test_out_`, intended to allow tests to write out a stream of data during the test. Note however that it is implemented via a `std::ostringstream`, and as such is not printed to the screen immediately.

As we can only ever have one instance of the `TestCollection`, it is implemented as a Singleton [4]. This has the additional benefit of allowing us to be able to register each test with the collection through the public static method `TestCollection::Instance()`. The `Testable` class's default constructor is implemented as so:

```
Testable::Testable(const std::string& name)
             : name_(name) {
  TestCollection::Instance().addTest(this);
}
```

Now any class deriving from `Testable` is automatically registered with the testing framework. The great benefit of this is that it allows us to add a test without changing **any** of the code already in the test build; all we need to do is add the new test's implementation file to our build list (or makefile). I believe this is important as I've several times seen code go untested as someone has forgotten to add the call to the test suite to the test driver code.

The header testable.h also contains the following macros:

```
#define IS_TRUE(exp)  test::isTrue(exp, \
#exp, __FILE__, __LINE__)
#define IS_FALSE(exp) test::isFalse(exp,\
#exp, __FILE__, __LINE__)
```

which call the following helper functions, ensuring we capture the line of code being tested, and the details of the file and line where the test can be found:

```
void isTrue(bool val, const char* code,
            const char* file, size_t line);
void isFalse(bool val, const char* code,
             const char* file, size_t line);
```

These functions evaluate the result of the test, and ask the test collection to log a failure if they are not true or false, respectively.

Our tests are then gathered up into the main body of the framework; the class TestCollection. The main body of the testing framework is contained in the function TestCollection::run() which looks like this:

```
void TestCollection::run() {
  const iterator end = tests_.end();
  run_number_ = 0;

  for (iterator current = tests_.begin();
    current != end; ++current, ++run_number_) {
    try {
      test::Testable& test = *current->first;
      test::TestResultCollection& test_result
                     = *current->second;
      if (test.setUp()) {
        try {
          test.run();
        }
        catch(...) {
          error("Error occurred while
                       running test");
          test.tearDown();
          continue;
        }

        if (test_result.failedCount() == 0) {
          ++pass_count_;
        }
        else {
          ++fail_count_;
        }

        if (!test.tearDown()) {
          error("Error occurred while
                tearing down the test");
        }
      }
      else {
        error("Setup failed");
      }
    }
    catch (std::exception& e) {
      error(e.what());
    }
    catch (...) {
      error("Unexpected error");
    }
  }
}
```

The main purpose of the function is to iterate through the list of tests, calling setUp(), run() and tearDown(), for each test in turn, catching any exceptions thrown, and checking that we only run tests that have been successfully set up, or recording tests that have failed to properly tear themselves down.

The final thing the test framework does is call TestCollection::report() which iterates through the TestResultCollection, reporting any passed tests, or failed tests along with any associated failures or errors.

Reading the Code

The actual code supplied on my website contains a few more comments than are displayed in this article, and is documented with Doxygen [5], so you can generate HTML, RTF, Latex, or man pages from it.

Using the Testing Framework

To demonstrate the use of the unit testing framework, I've chosen to use a modified version of the calculator Bjarne Stroustrup presents in The C++ Programming Language. I've modified it slightly, changing it into a class, and making it perhaps a little more reusable and testable.

The Calculator is intended to compute the result of simple formulae, possibly reusing the result of an earlier expression to calculate a more complex one such as: (a + b) * (c - 0.147) + d;

Calculator is a simple class defined as follows:

```
class Calculator {
public:
  Calculator();
  double evaluate(const std::string expression);
// ... private members ...
};
```

A nice simple interface for us to test! So we'll create a CalculatorTest.cpp file and we can begin writing our test:

```
#include "Testable.h"
#include "Calculator.h"
```

We need the definition of Testable, as we're going to inherit from it, and we need the definition of Calculator, as that's what we're going to test. Next we need to define our unit test class:

```
class CalculatorTest
                  : public test::Testable {
public:
  explicit CalculatorTest(const std::string&
       name) : Testable(name) {}
// ... rest of class follows ...
```

Remember that we need to pass the human readable form of our test name down to the base class. Next we'll want to provide the code to set up and tear down any classes or data we'll need for the test, in this case I've decided to dynamically allocate the Calculator here:

```
virtual bool setUp() {
  try {
    calc_ = new Calculator;
  }
  catch(std::bad_alloc& e) {
    std::cerr << "Error setting up test: "
              << e.what() << std::endl;
    return false;
  }
  return true;
}

virtual bool tearDown() {
  delete calc_;
  return true;
}
```

The `teardown()` method assumes `delete calc_;` will always succeed. Last but not least we need to implement the `run()` method:

```
virtual void run() {
  testBasics();
  testVariables();
  testCompound();
}
```

in which I've chose to break my tests down into related groups and run each group in turn. So let's look at the simple tests in `testBasics()`:

```
void testBasics() {
  double result = calc_->evaluate("1 + 1");
  IS_TRUE(equal_double(2.0, result));

  result = calc_->evaluate("1 + 1");
  IS_TRUE(equal_double(2.0, result));

  result = calc_->evaluate("3 - 1");
  IS_TRUE(equal_double(2.0, result));

  result = calc_->evaluate("1 * 2");
  IS_TRUE(equal_double(2.0, result));

  result = calc_->evaluate("6 / 3");
  IS_TRUE(equal_double(2.0, result));
}
```

Now we've gotten to the meat of it. We know `calc_` must be valid for the framework to have called `run()` on our test class, so we can start using it, and what simpler test for a calculator than 1 + 1, so we create a double called `result` to store the result of the `evaluate()` call:

```
double result = calc_->evaluate("1 + 1");
```

Then we use the `IS_TRUE()` macro to compare the result to 2, our expected answer:

```
IS_TRUE(equal_double(2.0, result));
```

Finally after our class declaration we need to create a single instance of the class, we can do this as so:

```
static CalculatorTest
                 the_test("Calculator Test");
```

You may prefer to place it in an anonymous namespace, as well as or instead of the static. We can then run the testing framework and hopefully we'll see something that looks like this:

```
C++ Testing Framework v1.0 Copyright 2001
Crickett Software Limited
The latest version can be downloaded from
http://www.crickett.co.uk
-----------------------------------------
Ran 1 test(s)
Passes:   1
Failures: 0
Errors:   0
-----------------------------------------
Test: Calculator Test
Output:
No failures.
No errors.
-----------------------------------------
```

If we deliberately introduce a failure, say:

```
IS_TRUE(equal_double(7.0,
                  calc_->evaluate("1 + 1")));
```

Then the framework will produce the following results:

```
C++ Testing Framework v1.0 Copyright 2001
Crickett Software Limited
The latest version can be downloaded from
http://www.crickett.co.uk
-----------------------------------------
Ran 1 test(s)
Passes:   0
Failures: 1
Errors:   0
-----------------------------------------
Test: Calculator Test
Output:
Failures:
Failed IS_TRUE(equal_double(7.0,
  calc_->evaluate("1+1"))) at line 68 in file
  d:\dev\c++\testingframework\calculatortest.cpp
No errors.
-----------------------------------------
```

Which gives us the location (file and line number) of the failed test, and the actual code for the test that failed. I suggest that you, dear reader, experiment with the supplied examples, perhaps undertaking the following exercises:

1 Extend the tests; there is scope for more comprehensive testing.
2 Extend `Calculator` to support function calls, for example sin, cos, and tan. Also write suitable tests for each.
3 Add another class to the project, and a suitable test class.

Logging

Sometimes it's just not that easy to test results directly, or we really might just want to log some text during the test, so the framework allows the test to log data. This is provided by the protected member variable `test_out_`, and all you need to do to record text during the test is treat it exactly as you would `std::cout`. The text is then reported in the Output section of the tests report.

Taking It Further

This article has introduced the design and development of the C++ unit testing framework we use at Crickett Software. It's continually evolving as we need it to, and the latest version will normally be available from our website.

In a future article we will look at how to use unit testing to improve the quality of your code, by catching tests earlier, and automating as much testing as possible. I'll also look at test first design. Any questions on the article are most welcome to `john@crickett.co.uk`.

John Crickett

Thanks To

Jon Jagger (`www.jaggersoft.com`) for feedback on an earlier testing framework and some suggestions that got me started on this incarnation of a C++ unit testing framework.
Mark Radford (`www.twonine.co.uk`) for comments on both the framework and this article.
Paul Grenyer (`www.paulgrenyer.co.uk`) for comments on several drafts.

References

[1] Extreme Programming – `http://www.xprogramming.com`
[2] *The C Programming Language* by Kernigan & Richie, 1988.
[3] C++ Unit Testing Framework – available from `http://www.crickett.co.uk`
[4] Singleton – `http://rampages.onramp.net/~huston/dp/singleton.html`
[5] Doxygen, an open source C++ documenting tool, similar to JavaDoc – `http://www.doxygen.org`

Other Interest

JUnit – a Java unit testing framework – `http://www.junit.org`, covered recently in CVu by Alan Griffiths
XP Unit Testing Frameworks – `http://www.xprogramming.com/software.htm`
XP Unit Testing – `http://www.xprogramming.com/xpmag/expUniTestsat100.htm`

Letters to the Editor

Comments on "Applied Reading - Taming Shared Memory" by Josh Walker in Overload 51.

I really enjoyed to see an article about shared memory in Overload 51. For one, it (re-)promoted a solution for a frequent problem (managing common access to shared objects) that is generally much more appropriate than the usual solution "multi-threading". The overwhelming number of publications (books and articles) about multi-threading had the effect that most developers think that multi-threading is the proper solution to the sharing problem (and even for problems that share virtually no resources). Therefore, it is really good to see an article about a long-known solution that is in my opinion nearly always more appropriate.

Then, I enjoyed the article because I'm now working for quite a while on a library to make the use of shared memory easier. Unfortunately, it turned out that using standard containers or strings inside of shared memory is virtually impossible. Not that there is a fundamental problem with containers in shared memory. Though it is not completely easy, it is possible to provide `shm_ptr` as a full blown pointer type (including pointer arithmetics). And providing a wrapper around `shm_allocator` that implements the STL allocator requirements is pretty easy. But unfortunately, the C++ standard gives library implementors the leeway to ignore the type `Allocator::pointer` and instead just assume that it is a raw pointer. And though the standard explicitely encourages library implementors not to use this leeway, I didn't yet come across an implementation that actually worked when `Allocator::pointer` was not a raw pointer. (Probably I just looked in the wrong places.)

What is really sad is that when I looked at the implementation, it seemed that there is no real reason for that, but just a general sloppiness by using "`T*`" instead of "`Allocator::pointer`". (The main reason for this is probably historical as well as an unwillingness to really think about the requirements of "`Allocator::pointer`" and document them as an implementation defined enhancement.) So, to make shared memory really comfortable, you need to provide your own replacements for the standard containers and string. Or you ensure that the shared memory segment is mapped to the same address in all processes, but this is an option that is rarely available.

Some final notes: Whether shared memory is actually the proper solution for the problem at hand I'm not so sure. If only, because it's pretty hard to implement a clean and safe message queue. But then, if a different solution would have been chosen, we wouldn't have got this excellent article. And a minor issue with the presented code: Why is the `size` parameter of the `shm_segment` constructor of type "`int`", while the return of `get_size()` and the internal `size_` member are of type "`unsigned int`". The proper type would probably be "`size_t`".

Detlef Vollmann
`<dv@vollman.ch>`

Response to Detlef Vollmann's comments

I found Detlef's comments on using standard containers in shared memory very interesting. It is unfortunate that current implementations fail to support this natural and useful extension. I hope this will change as libraries and compilers mature.

As I mentioned in the article, the choice of shared memory was influenced by my own skill set. It has so far proved an adequate solution, though I agree that it may not be the ideal one. Of course, it was also a useful topic for exposition.

As Detlef keenly points out, the `shm_segment` members and methods concerned with size should use `size_t`. The use of `signed int` is just sloppiness on my part.

Finally, let me thank Detlef for his comments. I hope more readers will send their feedback on Overload articles.

Josh Walker
`joshwalker1@earthlink.net`