contents

credits & contacts

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

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

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

Readers:
Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

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 - The Internet's Coming Silent Spring

Lawrence Lessig received a standing ovation for his keynote presentation at Usenix 2002 in Monterey California. Lessig is a professor of law at Stanford Law School, and a founder of the Stanford Center for Internet and Society. He is a cyberspace lawyer [1]. Usenix is a venerable and aging conference for operating systems academics and open source contributors. They are beardy Unix hackers [2]. I am beardless, and unixless, but I attended anyway as part of the yearly company beano.

Lessig has a gift for oration. His legal training and his many courtroom appearances have provided him with a highly polished presentation style. The pace of his delivery is measured, with a poetic meter that is hard to identify. The slides that accompanied his presentation were the most un-powerpoint-ey powerpoint slides I've ever seen, and were so numerous, and so smoothly transitioned, that they took on a filmic quality. His voice, his presence, and the visuals, combined to form a compelling platform for his message.

His talk, entitled 'The Internet's Coming Silent Spring', presented the argument that the Internet is being undermined by those that are threatened by the neutral and unrestrained innovation fostered by its network architecture.

Stories of the allowed

He introduced his topic by referring to a number of historical examples.

Firstly, he described the invention of FM radio by Edwin Armstrong in 1934. AM radio was plagued with static. FM offered higher fidelity, and greater range, but FM threatened the established broadcasting industry, in the form of RCA. RCA used the Federal Communications Commission (FCC) against Armstrong, by harassing him with patent infringement suits, and forcing him to switch frequencies. Twenty-three years to the day after patenting FM, Armstrong put on his hat, coat, scarf, and gloves, and walked out his apartment window, falling13 floors to his death. FM radio was not allowed.

Secondly, Lessig referred to the invention of Packet Switching by Paul Baran of Rand in 1964. Packet Switching was designed as a decentralized communications network that could survive nuclear attack. But, AT&T said: It "will never possibly work" and we'll be "damned if we'll allow the creation of a competitor to ourselves." The Internet was not allowed.

Thirdly, Lessig described the website that gave instructions on how to teach a Sony Aibo to dance Jazz. Sony invoked the Digital Millennium Copyright Act to have the code forcibly removed. Jazz was not allowed.

Lastly, Lessig told the tale of the Brothers Grimm and Walt Disney. The Grimm's unpleasant stories had passed into common property by the time Walt started making animated films. He quite legally borrowed from the stories liberally. In 1790 an author of a work owned rights over that work for 14 years. In 1832 that term was increased to 42 years, in 1909 to 56, in 1962 to 59, and then extended often until 1976 when it stood at 75 years. In 1998 the Sonny Bono Copyright Term Extension Act (aka the Mickey Mouse Protection Act) increased it to 95 years. Thus, no one shall do to Disney what Disney did to the Brothers Grimm.

Societies and Architecture

Lessig went on to describe the architecture of the Internet as a simple network, with smart applications. The benefit being that network users don't have to ask AT&T's permission in order to exchange information. The end-to-end nature of the network architecture allows anyone to do anything with anyone. Furthermore, the number of innovators increases from one to many, and the kind of innovations change from those that benefit the network owner, to those that benefit the network users. For Example, the Internet was innovated by Vint Cerf and Robert Kahn. The World Wide Web was innovated by CERN of Switzerland. ICQ was innovated by Israelis. HotMail was innovated by Indians. What do they have in common? They were kids and non-Americans, and most importantly they were not the owners of the network. These innovations were the consequence of the architecture.

Within society policies are agreed upon, and then codified within a body of law. By analogy the policies of the Internet are its architecture, and the law of the Internet are the protocols and the code. The architectural policies are implemented within the design of the protocols, and within the actual implementations of those protocols.

The Internet is the content that is delivered, the code that implements the protocols, and the physical media it runs over; the cables and the spectrum. The code at the core of the network is being corrupted by the influence of

the corporations that control the content and the physical media. The core is being corrupted whilst policy makers do nothing. They do nothing because of the framing of the debate.

Stolen Property

The debate is being framed in terms of property rights. Corporations own the **property**, and their property is being **stolen**.

For example, innovators are broadcasting cable content over the Internet. Just as the cable industry broadcasts network television content. That's "blood sucked from our veins" say the cable bosses. It's their property, and it's being stolen.

Another example is the Recording Industry Association of America (RIAA). They would rather have all consumer devices modified to prevent copying, than provide a digital distribution channel. Napster is theft!

Some historical context helps to highlight the shift in policy that has occurred. At the start of the 20th century there was a thriving sheet music business. Sheet music could not be copied and resold, as it was protected by copyright. An innovator created piano rolls, which were not a copy of the music, but a new form of distribution, so therefore the rolls were not infringing the right of any copyright holder. Piano rolls 'napsterized' sheet music. Similarly cable napsterized broadcast television, and the VCR napsterized the film industry.

In the past the law was fitted to the technology. Today the technology must fit to the law of the past. The shift is clear from the Napster case. The judge ruled that Napster must prove that their system was 100% free of copyrighted material. By comparison the innovators of the VCR only had to show that there were legitimate legal uses for their device. Would the VCR exist if the manufacturers had to prove that no one would ever use the device to infringe copyright holders rights? No.

This policy change has occurred because corporations wish to minimize competition. They want to be in the position of dictating which new technologies should be allowed.

The corporations want policy put into the code to protect their property.

Lessig believes that policy makers must be persuaded to acknowledge this policy change. The barrier is that everyone supports the protection of property owner's rights. Theft is after all wrong. But, reframing the debate can break down the barrier.

Is it property, and should it be owned? Should the Ford Motor company own the highways? How well would General Motors cars work on Ford Highways? The architecture of the Internet should be common property and should not be owned, as neutral platforms build innovation.

Is the property being stolen? Creativity has always built upon the past. The rights of the owner must be balanced with the rights of society. This is the precept upon which the patent system and the copyright system are based. Creativity breeds innovation.

The Internet should be free, free in the way Richard Stallman means free. We, the innovators of the Internet, should have the freedom to innovate, to tinker, to copy.

Silent Spring

Finally Lessig explained the meaning of the title for his talk. The book 'Silent Spring' was written by Rachel Carson in the early sixties. She single-handedly took on the chemical industry to stop the uncontrolled use of pesticides. A compelling chapter of the book describes a town where all life has been 'silenced' by the insidious effects of DDT. Lessig's ominous parallel is that if the attacks upon the core of the Internet are not defended, then the Internet will be silenced.

Code and Other Laws of Cyberspace

In closing, if you'd like to pursue more of the thoughts and writings of Professor Lessig you'll find information on his website [1] and in his recent book [4], an excellent review of which was written by Mike Godwin [5].

John Merrells
merrells@acm.org

References

[1] http://cyberlaw.stanford.edu/lessig/
[2] http://www.usenix.org/
[3] http://aibopet.com/
[4] *Code and Other Laws of Cyberspace*, Lawrence Lessig, Basic Books.
[5] http://www.oreilly.com/news/lessig_0100.html Review of 'Code and Other Laws of Cyberspace', by Mike Godwin.

Copy Deadline

All articles intended for publication in *Overload 51* should be submitted to the editor by September 1st, and for *Overload 52* by November 1st.

Alternatives for Partial Template Function Specialisation

By Oliver Wigley

Whilst template classes can be partially and wholly specialised, template functions cannot. Alexandrescu[1] presents a technique to simulate partial template function specialisation so that a uniform interface is preserved, and calls to the template function(s) can be made in a generic way. Key to the solution is a mapper type which the client code must use in the function calls. This extra 'type-to-type' mapper rather clutters the interface with what you might call an implementation detail. Here is a look at the original proposal, followed by some possible alternatives.

Original Solution

Overloaded template functions are at the heart of the 'specialisation'. A template struct is introduced (`Type2Type`) which serves as an identifier to facilitate function lookup:

```
//from [1]:
template <typename T>
struct Type2Type {
  typedef T OriginalType;
};

template <class T,class U>
T* Create(const U& arg, Type2Type<T>) {
  return new T(arg);
}

template <class U>
Widget* Create(const U& arg,
               Type2Type<Widget>) {
  return new Widget(arg, -1);
}
```

`Create()` 'new's instances of `Widget`s or instances of specific `Widget`-derived things. The `Widget` class constructors have two arguments – the second being an `int` which should be set to -1, and the derived classes' constructors all have just a single argument – hence the need for specialisation. The flavour of `Type2Type` which is passed to `Create()` determines which overload to use. The overload for the `Widget` class requires `Type2Type<Widget>`, and the completely generic version accepts `Type2Type` instances of `Widget`-derived types[2].

```
//test code
#include <assert.h>
struct WidgConfig {int i;};
struct Widget {
  Widget(const WidgConfig& setup,
         int a) { assert(-1==a); }
};

struct SpecialWidget : Widget {
  SpecialWidget(
     const WidgConfig& setup)
     : Widget(setup,-1) {}
};

WidgConfig cfg;
SpecialWidget* psw   = Create(cfg,
Type2Type<SpecialWidget>());
Widget* wi           = Create(cfg,
Type2Type<Widget>());
```

First Alternative

Alexandrescu initially suggests overloading by passing dummy objects of the appropriate `Widget` type rather than the `Type2Type` struct, but then dismisses it as it requires the construction of potentially superfluous objects, incurring the overhead of that construction – and there's also the overhead of a pass-by-value to consider:

```
//from [1]:
template <class T, class U>
T* Create (const U& arg, T/*dummy*/) {
  return new T(arg);
}

template <class U>
Widget* Create (const U& arg,
                Widget/*dummy*/) {
  return new Widget(arg, -1);
}
```

If `Create()` is our only mechanism for getting instances of `T` or `Widget`, then we will have difficulty calling it the first time when we don't yet have an instance to pass. With a little tweaking, however, this does offer a feasible solution without the problem of superfluous object creation. Pointer types could be used to overload the function instead:

```
template <class T, class U>
T* Create (const U& arg, T*/*dummy*/) {
  return new T(arg);
}

template <class U>
Widget* Create (const U& arg,
                Widget*/*dummy*/) {
  return new Widget(arg, -1);
}
```

Pass a pointer of the appropriate type and the correct function is called. No extra constructors or copy constructors are now being called and the template function is effectively specialised. Overloaded lookup can go ahead courtesy of an uncharacteristically welcome `NULL` pointer, so there's no need to worry about supplying a `Widget` instance that we don't have yet:

```
SpecialWidget* pSwi =
  Create(cfg,
    reinterpret_cast<SpecialWidget*>(0));

Widget* pWid =
  Create(cfg,
    reinterpret_cast<Widget*>(0));
```

As an extra, this approach offers new possibilities as we now have the option to pass a valid object, which can be exploited by either overload. Imagine the `Widget` class as a `Window` class:

```
template <class U>
Window* Create (const U& arg,
                Window* parent) {
  if(parent) {
    Window* child =
             new Window(arg, -1, parent);
    return child;
  }
  else
    return new Window(arg, -1);
} // [3]
```

This does offer an alternative style with extended flexibility, and doesn't require the `Type2Type` type.

Second Alternative

Whilst template functions cannot be partially specialised, we can partially specialise template classes. It could help to make use of the functor[4] idiom:

```
template <class T, class U>
struct Create {
  T* operator()(const U& args) {
    return new T(args);
  }
};

template <class T>
struct Create <Widget, T> {
  Widget* operator()(const T& args) {
    return new Widget(args, -1);
  }
};

WidgConfig cfg;
SpecialWidget* psw =
 Create<SpecialWidget,WidgConfig>()(cfg);
Widget* pw =
 Create<Widget,WidgConfig>()(cfg);
```

Although the function calls to `Create` might look rather esoteric, this does also offer a generic interface and dispels the need for extra types to be defined just to solve the original problem:

```
template <class T, class U>
struct Create {
  T* operator()(const U& args) {
    return new T(args);
  }
};

template <class T>
struct Create <Widget, T> {
  Widget* operator()(const T& args) {
    return new Widget(args, -1);
  }
};

SpecialWidget* psw =
 Create<SpecialWidget,WidgConfig>()(cfg);
Widget* pw =
 Create<Widget,WidgConfig>()(cfg);
```

Oliver Wigley
oliver.wigley@teleca.com

Footnotes & References

[1] Andrei Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001 (section 2.5)

[2] Microsoft Visual C++ 6.0 considers the call to `Create(cfg, Type2Type<Widget>)` to be ambiguous, so the code does not port. Explicit template arguments are needed to coax it in the right direction, but consistency in the interface is broken:

```
SpecialWidget* psw =
  Create<SpecialWidget,
         WidgConfig>(cfg,
             Type2Type<SpecialWidget>());
Widget* wi = Create<WidgConfig>(cfg,
             Type2Type<Widget>());
```

See MSDN Knowledge Base article Q240869 for bug description. Also, if we try to specify explicit template arguments for the other overload as well: `Create<SpecialWidget, WidgConfig>(cfg, Type2Type<SpecialWidget>())`, then VC++ is unable to match the overload, generating compiler bug C2665. Strangely, it actually matches the lookup when you call it with `Create<SpecialWidget>(cfg, Type2Type<SpecialWidget>())`.

[3] We still need to help Microsoft Visual C++ 6.0 to know which function we want to call, by providing explicit template arguments, so a generic and portable style is lost.

[4] Bjarne Stroustrup, *The C++ Programming Language 3rd Ed.*, Addison-Wesley, 1997 (Section 18.4)

[5] Microsoft Visual C++ 6.0 lacks support for partial specialisation of template classes. See MSDN Knowledge Base article Q240866. In that case each possible variation (i.e. constructor) for the `Widget` class would have to be fully specialised with a `Create` class of its own:

```
#ifdef __MSVC__
  template <>
  struct Create <Widget, WidgConfig> {
    Widget* operator()(
                const WidgConfig& args) {
      return new Widget(args, -1);
    }
  };
  // and any other complete
  // specialisations needed...
#else
  // Create <Widget, T> implementation
  // as before
#endif
```

STL-style Circular Buffers By Example

by Pete Goodliffe

I've always found that you don't learn anything until you try to actually do it. In this article I provide a practical chance to learn some new C++ knowledge. If you fancy getting your hands dirty here then you might pick up some valuable new techniques. This is a gentle introduction to writing robust STL-like generic containers.

Recently I needed a *circular buffer* to implement some low level logic. These data structures aren't exactly complicated to write, but it got me thinking. That's a dangerous thing at the best of times. I've never written an STL-style container before, and I'd never come across an STL-like circular buffer[1].

With a somewhat gung-ho attitude I began to implement an STL-compatible circular buffer class. I'm not going to say I've got it perfect but I present here my journey of discovery in the hope that it will be useful. You have the chance to cover a lot of the same ground I did since I'll take us on this journey via a few "exercises" – just a little something to get you thinking.

Put as much effort into the exercises as you like. As with so many things, the more effort you put into it, the more you'll get out at the end.

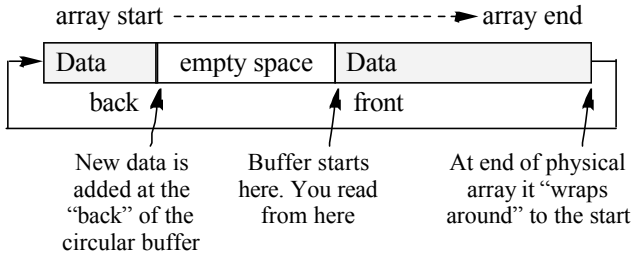
What is a circular buffer?

Before we delve into the murky depths of the STL let's take a quick refresher course in circular buffers.

There are a number of fundamental data structures in computer science. The *array* is about as basic as it comes, its implementation essentially requires a base pointer to some allocated memory, and an index into it. You could perhaps argue a *stack* is even more basic: if you don't want to check for the location of top of the stack you only need a single "current location" pointer. Textbook examples of stack code are generally implemented in terms of an array, though.

Coming hot on the heels of these two faithful friends is the good old circular buffer. It looks like an array but has FIFO consumption semantics[2], can smell really quite like an array, but gives the pretence of being infinite in size. How does it do this? This is where the *circular* bit comes into play. The logical buffer is considered to 'wrap around' the physical array it is implemented inside. The 'head' and 'tail' of the buffer chase each other around the implementation array.

The following diagram may help envision this.



-Circular buffers have a number of uses. For example, device drivers that constantly receive data (like a serial port), and need to buffer it often use circular buffers – acting as a data 'producer' for the client code. It is the client's responsibility to consume the data about as fast as it is produced to ensure that no data is lost. This makes reasoning about the data exchange easier. The circular buffer helps here since the producer (driver) only needs to worry about adding data to the end of the circular buffer, whilst the consumer only needs to worry about reading from the front.

A simple C++ circular buffer class interface would look like this. For the moment, let's presume that the buffer just stores `int`s:

```
class simple_cbuf {
  public:
    enum { default_size = 100; };
    explicit simple_cbuf(size_t size =
                         default_size);
    ~simple_cbuf();
    size_t size() const;
    bool   empty() const;
    int    top() const; /* see below */
    void   pop();
    void   push(int new_value);
  private:
    /* whatever you want */
};
```

Note that here we separate `top` and `pop`. Some implementers might like a single atomic top-and-pop function. However, this approach keeps consistency with the STL `vector`-like interface which will help later on. There is also a practical reason to split the two functions – it makes exception safety issues easier to reason about. More on this later.

Exercise #1
Implement this simple class API. At this stage it's OK to store the data in a dynamically created array[3].

Hopefully that exercise made you wonder what to do in `push` when the buffer is full. If we want to "pretend" to be of infinite size then somewhere along the way some data will have to be thrown away. There are two choices:

- throw away the new data, and leave the `simple_cbuf` state unchanged, or
- throw away the data at the 'front' of the circular buffer and replace it with the new 'end' data.

In practice the latter case is often required – this is the whole point of the circular buffer (the first method is really rather like a fixed size array, after all). It might be nice to provide a policy option to select what behaviour you want. Our final solution will allow us to make this choice with no extra cost.

Did you think about copy construction and copy assignment? Basing our implementation on a dynamically created array will mean that we'll need to provide correct behaviour for these two members. They must be either declared as private and unimplemented, or added to the class' public API.

Random access?

Now here's an interesting usage question, the above minimal interface doesn't allow us to randomly access the buffer. In this respect it just looks like a stack. This isn't too hard to do, though – try exercise two.

Exercise #2
Implement an `operator[]` for the `simple_cbuf`.

This should be fairly simple to do, there's just one thing to watch out for. For maximum usefulness we need to provide *two* versions of `operator[]`; one 'normal' version for non-`const` objects, and one for `const` objects. This allows us to provide assignment into the `simple_cbuf`. You should end up with two functions whose signatures are thus:

```
int       &operator[](size_t);
const int &operator[](size_t) const;
```

This is the canonical form of these functions, although for an `int`–holding container we perhaps don't need to pass a `const int&` from the `const operator[]`, we could get away with just an `int` return type. We'll stick with this because it's good practice.

Now we have something that looks a quite like an array, but with more usual circular buffer FIFO characteristics.

With this kind of behaviour is this data structure genuinely useful? Does it have any benefits over a plain array? Actually it can be very useful, but it would be limited to a smaller set of problems than something like, say, an STL `vector`.

Make it generic

So we now have a pretty complete `simple_cbuf` class, but it only works for `int`s. The final introductory step is to make it generic.

> **Exercise #3**
> Add a line `template<typename T>` to the beginning of the class definition and modify the code accordingly.
> How much of the public API do you need to modify?

This really isn't too onerous. Obviously if you were thinking about separate `cbuf.h` and `cbuf.cpp` files you now want to munge the function definitions into the header file, but it's largely a case of selectively replacing of `int`s with `T`s.

The second question: "How much of the public API do you need to modify?" is worth considering. For a simple integer circular buffer you can pass parameters to `push` by value. However since `T` could now be *any* class type you really want to pass by `const&`. Also should `top` return a `T` by value, or by `const&`? The latter makes more sense, since it prevents clients from being stupid with a temporary object by writing something like:

```
cbuf.top() = T(); // meaningless; no part
    // of cbuf would be modified by this,
    // despite what it looks like
```

Our choice of return type for `operator[]() const` is now also vindicated.

So now we have seen what a circular buffer is, how to use it and how to implement a fairly basic example case. The rest can't be that hard, can it?

Moving towards the STL style

Now we'll start to write an STL style container that's based on the class we've developed above. I'm going to presume at this point that you're familiar with using the STL and its definitions of container class interfaces[4]. What do we need to provide to make this circular buffer class more STL-like?

- standardised `typedef`s like `value_type` etc,
- standardised API names,
- iterators (forward, reverse, random access),
- use allocators for memory operations, and
- to be maximally useful, exception safe (and also exception neutral)

We'll start adding this functionality in stages and consider what we're doing at each point.

Our implementation will start off template based, using a dynamically created array as the internal data storage implementation. So here's the initial framework of our class, not yet worrying about any of the above list of items to include. We'll flesh it out as we go along:

```
template <class T>
class circular_buffer {
  public:
    explicit circular_buffer(
        size_t capacity = 100)
        : array_(new T[capacity]),
          array_size_(capacity),
          head_(0), tail_(0),
          contents_size_(0) {}
    ~circular_buffer()
        { delete [] array_; }
    /* ... */
  private:
    T      *array_;
    size_t  array_size_;
    size_t  head_;
    size_t  tail_;
    size_t  contents_size_;
};
```

Step 1: Some standard type definitions

To work with STL components we need to provide these definitions in the public API of our class[5]:

```
value_type      // The type of the
                //  container's elements.
Pointer         // A pointer to the
                //  element type.
const_pointer   // As above, but const.
Reference       // A reference to the
                //  element type.
const_reference // As above, but const.
size_type       // The type used to index
                //  into the container.
difference_type // The type of the result
                //  of subtracting two
                //  container iterators.
```

> **Exercise #4**
> What should they be?

No rocket science yet. At this stage, my choice was:

```
typedef T          value_type;
     /* T is template param */
typedef T         *pointer;
typedef const T   *const_pointer;
typedef T         &reference;
typedef const T   &const_reference;
typedef size_t     size_type;
typedef ptrdiff_t  difference_type;
```

Now the original framework code can be slightly modified – I'll change the variable and parameter types from `size_t` to `size_type` and for `array_` from `T*` to a `value_type*`. Is this too picky? No, it will aid future maintainability. If one of these definitions needs changing later (hint – it will) then the change can be made in one place and apply to all the code we've written.

Step 2: Appropriate function names and implementations

Now, in the `vector` API we have `front`, `back`, `push_back`, `pop_back` and `clear`. We'd like those (or something akin to them). We'll clearly not provide `pop_back`, instead there will be `pop_front` for good circular buffer measure.

Exercise #5
How would you implement these?
Take care over function signatures to ensure maximal efficiency in the light of template types.

`front` and `back` are pretty simple. There's just one thing to watch out for here, appropriate `const` and non-`const` versions:

```
reference front()
              { return array_[head_]; }
reference back()
              { return array_[tail_]; }
const_reference front() const
              { return array_[head_]; }
const_reference back() const
              { return array_[tail_]; }
```

What happens if you call these when the circular buffer is empty? It's an invalid call and we define this to result in *unspecified behaviour*. We could provide a 'safe' version of the `circular_buffer` class later on that does checking, but we don't want to be burdened by it unnecessarily. It is up to the user to call us in a consistent and valid manner.

The `clear` member function is also reasonably simple:

```
void clear()
  { head_ = tail_ = contents_size_ = 0; }
```

In order to implement the final two functions, I'd define these private helper functions:

```
void increment_tail() {
  ++tail_;
  ++contents_size_;
  if (tail_ == array_size_) tail_ = 0;
}
void increment_head(){
  // precondition: !empty()
  ++head_;
  –contents_size_;
  if (head_ == array_size_) head_ = 0;
}
```

Now `push_back` and `pop_front` can be implemented along the following lines. It's actually a reasonably verbose `push_back` implementation, but will show exactly how we reason about the contents of the circular buffer, and where `head_`, `tail_`, and `contents_size_` fit into the equation:

```
void push_back(const value_type &item) {
  if (!contents_size_) {
    array_[head_] = item;
    tail_ = head_;
    ++contents_size_;
  }
  else if (contents_size_ != array_size_)
  {
    increment_tail();
    array_[tail_] = item;
  }
  else {
    // We always accept data when full
    // and lose the front()
    increment_head();
    increment_tail();
    array_[tail_] = item;
  }
}
void pop_front() { increment_head(); }
```

Again with `pop_front` we'll insist the user only calls the function when it is semantically valid, making our life a lot easier. Don't think we're being unnecessarily lazy here; this is exactly what the standard `vector` does!

For the record, can we coalesce some of that sloppy `push_back` logic above? Indeed we can, if we apply a few other minor modifications to the class:

```
void push_back(const value_type &item) {
  increment_tail();
  if (contents_size_ == array_size_)
    increment_head();
  array_[tail_] = item;
}
```

That looks nicer. To get here we have removed the special case of an empty buffer. To cope with this we need to set the member variables to a different pre-ordained state. The constructor and `clear` should initialise `tail_` and `contents_size_` to zero as before. `head_` should now though be set to `1`.

Step 3: More simple functions

You won't get many points for these simple functions, but they make for a consistent container API.

Exercise #6
Implement the following simple functions:

```
size_type size()     const;
size_type capacity() const;
bool      empty()    const;
size_type max_size() const;
```

They're all self explanatory except for `max_size`. This function returns the largest possible size of the container. A reasonable example implementation is:

```
size_type max_size() const {
  return size_type(-1) /
                 sizeof(value_type);
}
```

An alternative is to use `std::numeric_limits<size_type>::max()`. However, this requires that we include the `<limits>` header file and we'll avoid that for the moment.

Step 4: Random access

We want to implement iterators, and this requires that we provide general access to our circular buffer via `operator[]`. We implemented this in exercise #2, so we need to move this over to our new class. However, to be more thorough and more `vector`-like we can observe that `vector` also provides the `at` function, that provides a checked access, whilst the `operator[]` only provides unchecked access.

Exercise #7
Implement both `const` and non-`const` forms of `operator[]` and `at` for `circular_buffer`.

Now our circular buffer is randomly accessible we can move onwards and upwards...

Step 5: Implement an iterator

In exercise #2 we said that it was debatable whether the `operator[]` functions should have been added to the circular buffer class. Certainly in a traditional circular buffer design you wouldn't have them. However, part of the point of this whole exercise is to learn how to write a generic container that can be iterated over, which will make it compatible with standard library algorithms. Because of this we *should* be able to randomly access data in the circular buffer.

If you hold to the view that buffer data may be read, but may not be written by random access, then you will only implement the `const` version of `operator[]` and only provide a `const_iterator`. However, for the full STL "experience" we'll allow the non-`const operator[]` and provide both mutable types of iterators.

We'll start off by considering how to write a non-`const` forward iterator, and work up from there.

The simplest example of an iterator would be for an array – the iterator type would be a simple pointer. The "end" iterator would be a pointer to one-past-the-end of the array. We can't simply use a pointer to iterate over our circular buffer, so we'll have to define our own class type. For obvious reasons we'll call it `circular_buffer_iterator`.

Exercise #8
What data members does `circular_buffer_iterator` require?

To iterate over a `circular_buffer` effectively we only need to know which the `circular_buffer` is and at what index we currently stand. However, `circular_buffer` is a template type so how can we refer to it? We'll make the iterator a template class too, dependant on the type of the circular buffer. We could make it dependant on the `value_type` of the `circular_buffer`, but we choose not to. Why? Because later on we'll add more template parameters to the container class and that would make the alternative iterator implementation prohibitively complex.

Should we store a pointer or reference to the `circular_buffer`? They are equivalent for these purposes – if a reference became stale the iterator could be considered to have become invalid anyway, using the iterator would then be invalid. However, since we need to implement `operator==` for the iterator class, it's easier to compare pointers, so that's what we'll choose.

This means we start off with a class like this:

```
template<typename T>
      // T is circular_buffer type
class circular_buffer_iterator {
  public:
    typedef T cbuf_type;
    circular_buffer_iterator(
         cbuf_type *b, size_t start_pos)
         : buf_(b), pos_(p) {}
      /* ... */
  private:
    cbuf_type *buf_;
    size_t     pos_;
};
```

There's a lot of operations we need to define for this class. But first, we need to add the appropriate `typedef` and functions into `circular_buffer` class to meet STL requirements[6]:

```
typedef
   circular_buffer_iterator<self_type>
   iterator;
iterator begin()
    { return iterator(this, 0); }
iterator end()
    { return iterator(this, size()); }
```

We'll have appropriately defined `self_type`, naturally.

STL iterators are required to define a number of `typedef`s. These are:

```
iterator_category // See below.
value_type        // Type of element
                  //  iterator 'points
                  //  to'.
size_type         // Container index
                  //  type.
difference_type   // Container difference
                  //  type.
Pointer           // Type of a pointer to
                  //  element.
const_pointer     // As above, but const.
Reference         // Type of a reference
                  //  to element.
const_reference   // As above, but const.
```

Don't they look suspiciously like the `typedef`s in the container class? Does that give you a clue how to implement them?[7] But

what's that `iterator_category` definition for? This is a type as defined by the C++ standard that defines what operations are valid on the iterator. We can choose what kind of iterator to provide, and define it here. The simplest is a *forward iterator*, the most comprehensive a *random access iterator*. For the moment we'll compromise: our iterator will be a *bidirectictional iterator*, so we'll set this to `bidirectional_iterator_tag`[8].

Exercise #9
What operations should an iterator support? How do you implement them?

Obviously, we need to dereference an iterator. That means we need to provide an implementation of `operator*` and `operator->`. They're not hard. The following code does the trick:

```
T &operator*()
        { return (*buf_)[pos_]; }
T *operator->()
        { return &(operator*()); }
```

The other operations on an iterator type are reasonably simple. We don't have to do too much error checking; remember that incrementing an iterator past `end()` is invalid and results in undefined behaviour.

Exercise #10
For a bidirectional iterator we need to implement the follow functions. How would you do this?

```
self_type &operator++()
self_type operator++(int)
self_type &operator--()
self_type operator--(int)
```

These are some simple random access iterator operations that we can also implement here. How will you provide these?

```
self_type operator+(difference_type n)
self_type &operator+=(difference_type n)
self_type operator-(difference_type n)
self_type &operator-=(difference_type n)
bool operator==(const self_type &other) const
bool operator!=(const self_type &other) const
```

What are the differences between the two forms of `operator++` and `operator--`?

Answering the last question first, remember that these two forms are pre- and post-increment/decrement (respectively). Take care over the canonical forms of all of the above functions. As a guide, here are the implementations of `operator++`, `operator+` and `operator+=` (for a suitable definition of `self_type`):

```
self_type &operator++() {
  ++pos;
  return *this;
}
self_type operator++(int) {
  self_type tmp(*this);
  ++(*this);
  return tmp;
}
self_type operator+(difference_type n)
{
  self_type tmp(*this);
  tmp.pos_ += n;
  return tmp;
}
self_type &operator+=(difference_type n)
{
  pos_ += n;
  return *this;
}
```

`operator==` and `operator!=` are made easier since we decided to store pointers to the `circular_buffer` and not references.

Step 6: A `const` iterator

Exercise #11
How different from the above iterator interface is a `const` iterator interface?

The answer is, not much. We'll need to point to a `const circular_buffer`, return `const` references to the buffer data when deferenced, and that's about it. It seems a little bit of a shame to rewrite another class that is almost identical to the one we already have. To prevent too much duplicated code we can factor out the commonality. How best do we do this? Write one class and move out the policy decisions to template arguments.

Unfortunately it's not quite as simple as defining the existing template parameter `T` (circular buffer type) to be a `const circular_buffer`. We need to determine the type of reference to return (i.e. choose between `circular_buffer`'s `reference` or its `const_reference` types). We can make this another parameter, and we're *almost* there. What we've got now is:

```
template<typename T,
       typename elem_type =
       typename T::value_type>
class circular_buffer_iterator {
  public:
    /* ... */
    elem_type &operator*();
    elem_type *operator->();
};
```

Providing a default type for this means that our original `circular_buffer::iterator typedef` is still correct. The rest is the same, even the definitions of these operators. To create a const iterator we just need to instantiate the iterator class with

```
<const circular_buffer, const
circular_buffer::value_type>
```

There is only one problem left with our reverse iterator implementation. You need to be able to convert from a non-`const` iterator to a **`const`** one so we can write code like:

```
circular_buffer::const iterator i =
                                    cbuf.begin();
```

where `cbuf` is a non-**`const`** circular buffer. The path of least resistance for implementing this is to add another template parameter `T_nonconst` (putting it after `T`), which is the same as template parameter `T`, but without any const qualification. We can then write the following few lines and get the desired conversion:

```
circular_buffer_iterator (
      const circular_buffer_iterator<
         T_nonconst, T_nonconst,
         dir, typename
         T_nonconst::value_type>
      &other )
  : buf_(other.buf_), pos_(other.pos_) {}
  friend class
    circular_buffer_iterator<
         const T, T,
         dir, const elem_type>;
```

Naturally, we now add a `typedef` definition for `reverse_iterator` to the `circular_buffer` class. We can now do most normal iterator operations. One more thing to consider for now...

Step 7: A reverse iterator

If we can iterate forwards then we can also iterate backwards. Conventional STL containers provide this functionality, returning *reverse iterators* from their `rbegin` and `rend` member functions. A reverse iterator starts (logically enough) at the last element in the container. Every time you increment it, it will actually step *back* one element. `rend` points to the 'one-past-the-front' element of the container. Providing reverse iterators means we can apply all the standard algorithms backwards as well as forwards without writing any new algorithm code.

> **Exercise #12**
> How would you modify the iterator class we've already made to create a reverse iterator?

There's two ways to do this. The easy way, or the hard way. Choose! I took the high road, but we don't have to when the low road is perfectly acceptable. At first I created my own reverse iterator class in a similar manner to the const iterator; cunning additional template parameters.

However, to make our life that little bit easier the STL provides an iterator adaptor that automatically creates a reverse iterator for you, based on your forward iterator implementation. I noticed that just a little bit too late, although I was pretty happy that my reverse iterator looked so much like the standard library one! Using this STL facility is by far the better option. It is likely to contain fewer bugs (hopefully none), and will more likely behave in a manner our users expect.

So how do we use this STL magic? Just add the following lines to the `circular_buffer typedefs`:

```
typedef
    std::reverse_iterator<iterator>
    reverse_iterator;
typedef
    std::reverse_iterator<const_iterator>
    const_reverse_iterator;
```

That's it. If you really want to understand what's going on in the `reverse_iterator` template class, think how you would implement your own version. Consider how you would represent `rend()` bearing in mind that our index type `(size_type)` is unsigned and the one-past-the-beginning index would be –1. Solve that conundrum and you've done the hard work of a reverse iterator implementation.

What have we achieved?

So far we have created a circular buffer class that looks pretty similar in API and usage conventions to the other STL containers. This means that any C++ programmer can pick the class up and use it pretty much immediately with no steep learning curve. It also means that, thanks to the iterators we have provided, the class can immediately be used with all the standard library algorithms currently available.

Of course, there's still more work we need to do…

Next time

We'll look into working with standard library allocators and knitting up exception safety issues. There are one or two more functions to provide which we'll look at too – look over what we've done and see if you can work out what's left.

Pete Goodliffe
`pete@cthree.org`

Endnotes

1. There may be a good reason; it's a moot point whether such a data structure is a genuinely useful thing. Still, why let that stop us?
2. FIFO: First In First Out. You can only add data to the end of the "array" and consume from the "front". Imagine sending ping pong balls down a thin vertical tube, the order you get them out at the bottom is the order you put them in at the top.
3. If you don't know why this might be a problem later on don't worry – you'll find out soon enough. (Well, in the second part of this series.)
4. If you want to view an online reference of the STL interfaces, there is one publicly available from `http://www.sgi.com/tech/stl/`.
5. Actually, there's more (including iterator definitions) but we'll start here.
6. We'll also have to add a `const_iterator` definition, but we'll come to that in the next section.
7. Whilst I `typedef` these here based on the container `typedef`s, you can alternatively make your life easier by using the `std::iterator` template class definition.
8. Don't worry, we'll make this a full random access iterator in the next article.

Execute Around Method and Proxy Goulash

by Alan Griffiths

A recent design discussion resulted in a solution that has elements with strong similarities to the "Execute Around Method" and "Proxy" patterns while, in both cases moving outside the scope of the usual descriptions of these patterns [Henney2001, GOF1995].

This article presents the scenario we encountered as a pattern story. It notes the similarities and differences to the canonical forms of "Façade", "Execute Around Method" and "Proxy" patterns; and, raises the question "are these still the same patterns – or have they been cooked beyond recognition?"

Intent

To grant access to specific functionality only between paired operations.

Motivation

When updating large amounts of state it may be desirable to ensure that "before" and "after" messages are sent to the owner of that state so that any necessary preparation or cleanup may be applied – or that concurrent operations can be inhibited.

Consider a batch update of the product lines available within a system. A complete list of product lines is supplied by an external source and used to create, amend or delete product lines available within the system. Because the only indication that a product is to be deleted is that no corresponding product line is supplied it is necessary to accumulate information regarding the product lines accessed during the update and to deal with deletions when the update completes. (One of the delights of interworking with client's "legacy systems" is that, in this instance, they cannot provide perfectly simple information – such as positive notification of deletes.)

Typically the product lines are stored as rows within an RDBMS and accessed via a *Broker* class that provides the persistence mechanism. The Broker provides a finer grained and more extensive interface than is required by the needs of the application and access is mediated by a *Director* that acts as a Façade [GOF1995]. (In an EJB based system the Director would be a stateless session bean and a Broker used in place of an entity bean to avoid the cost of unwanted synchronisation.)

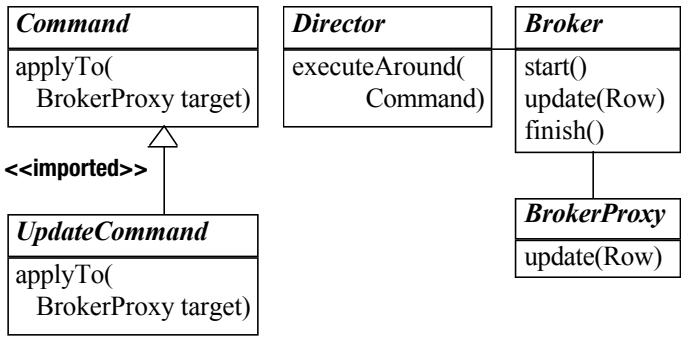
Because Java allows classes to grant privileged access to package members it is possible (and not uncommon) to restrict access to some Broker methods to the package and place both Broker and Director in the same package. This adaptation of the pattern enforces use of the Façade by classes outside the package and allows protocols to be enforced. In particular declaring "start", "update" and "finish" methods with package access will limit the code that calls them to the this package.

In keeping with its role as a Façade the Director should have the responsibility of calling "start" and "finish" methods on the Broker before and after processing of the batch update. However, receipt of the update is not the responsibility of the Director and, in fact, is driven by another part of the system entirely. (It might be possible to wrap the processing of the input as an "iterator" passed to the Director – however this would add complexity to the system.)

This leads us towards a variation of "Execute Around Method" where the paired operations are "start" and "finish" method calls (not resource allocation) and whose target is the Broker (not self invocation on the Resource object as described by Kevlin Henney [Henney2001]). We still have Kevlin's *Command* object – but not only has Resource been split into Director and Broker it is not passed to the Command's applyTo method ("run" in Kevlin's paper).

However, we've not yet resolved the full context – as the implementation of the Command interface still won't have access to the necessary Broker methods. But this is where our variation on Proxy comes into play: we define a BrokerProxy class that, being in the same package as the Broker, has access to the necessary functionality and can implement public forwarding functions. It is this (not a resource) that is passed to the Command's applyTo method by the Director. (To ensure that the Director isn't bypassed construction of the BrokerProxy is given package access.)

This usage of Proxy differs from that described in [GOF1995] because additional functionality is exposed by the Proxy class. Specifically it doesn't substitute for a Broker. (I feel it is far closer to Proxy than to Adapter or Bridge.)



Participants

- **Broker (Broker)** – supplier of the functionality to which access is controlled
- **Façade (Director)** – responsible for enforcing the access control protocol
- **Command (Command)** – declares a usage (applyTo) method for the functionality
- **ConcreteCommand (UpdateCommand)** – user of the access controlled functionality
- **Proxy (BrokerProxy)** – forwards calls to the Broker within the controlled scope

Consequences

The client code in `UpdateCommand.applyTo()` has access to the `Broker.update()` method via the BrokerProxy. The Director is able to ensure that `start()` and `finish()` are invoked at the appropriate points in the `executeAround` method.

Sample code

The following code illustrates the implementation of this dish in Java. We'll assume that a product line comprises its name and price:

```
public class Product {
  private String name;
  private int price;
  public Product(String name, int price){
    this.name = name;
    this.price = price;
  }
```

```
  public String getName() {
    return name;
  }
  public int getPrice() {
    return price;
  }
}
```

A Broker class called `ProductBroker` manages the persistent storage for products. It provides methods for retrieving Product information, and for updating the product table:

```
class ProductBroker {
  // ...
  List listProductNames() {
    // ...
  }
  Product getProduct(String name) {
    // ...
  }
  void startUpdate() {
    // ...
  }
  void update(Product data) {
    // ...
  }
  void finishUpdate() {
    // ...
  }
}
```

Because the broker doesn't provide public access we provide public access to the update method via the `Proxy` class:

```
public class ProductBrokerProxy {
  // ...
  public void update(Product data) {
    broker.update(data);
  }
}
```

The Command interface by which the client code supplies the update logic is:

```
public interface ProductUpdateCommand {
  public void applyTo(
            ProductBrokerProxy target);
}
```

All of this is co-ordinated by the `ProductDirector` façade as follows:

```
public class ProductDirector {
  // ...
  public void updateProducts(
        ProductUpdateCommand command) {
    final ProductBrokerProxy proxy =
              new ProductBrokerProxy();
    synchronized (this) {
      broker.startUpdate();
      command.applyTo(proxy);
      broker.finishUpdate();
    }
  }
}
```

While there appear to be a lot of pieces to the implementation of the final design the client code is clear and does not rely on following a "start/update/finish" protocol for correctness.

```
public class UpdateProducts {
  // ...

  public static void main(String[] args)
  {
    // Set up some example data...
    final Vector data = new Vector();
    data.add(new Product("beans", 27));
    data.add(new Product("chicken",
                                 525));
    // Process the example data...
    final Iterator iter =
                  data.iterator();
    director.updateProducts(
            new ProductUpdateCommand()
    {
      public void applyTo(
            ProductBrokerProxy target)
      {
        while(iter.hasNext())
        {
          target.update(
                  (Product)iter.next());
        }
      }
    });
  }
}
```

Acknowledgements

Thanks to Jason Martin and Andrew Rigley who: brought the motivating example to my attention; allowed me to participate in an interesting design session that led to the above resolution; and, reviewed the draft article. Additional thanks are also due to Jason who supplied the sample code on which the above fragments are based.

Alan Griffiths alan.griffiths@microlise.com
Jason Martin jason.martin@microlise.com
Andrew Rigley andy.rigley@microlise.com

References

[Henney2001] Kevlin Henney, "Another Tale of Two Patterns" *Java Report* March 2001 http://www.two-sdg.demon.co.uk/curbralan/papers/AnotherTaleOfTwoPatterns.pdf

[GOF1995] Gamma, Helm, Johnson, Vlissides, *Design Patterns* ISBN 0-201-63361-2

Even More Java Exceptions

by Jon Jagger

A recap

In Overload 49 Alan Griffiths continued his exploration of the exception safety landscape in the wilds of Java and looked at a problem introduced by Tom Cargill; namely how to make a copy of a *whole* object holding multiple parts in an exception safe manner. In his article Alan ensured that if construction of the whole object failed, the resources of all fully constructed parts were released. He achieved this by careful use of try/finally blocks and statement ordering - essentially, like this:

```
public class Part implements Cloneable {
  ...
  public void dispose();
}

public class Whole {
  ...
  public void copy(Whole other) {
    Part t1 = (Part)other.p1.clone();
    if (t1 != null) {
      try {
        Part t2 = (Part)other.p2.clone();
        if (t2 != null) {
          t1.setParent(this);
          t2.setParent(this);

          // pivot point

          Part swap1 = t1;
          t1 = p1;
          p1 = swap1;

          Part swap2 = t2;
          t2 = p2;
          p2 = swap2;
        }
      }
      finally {
        t2.dispose();
      }
    }
    finally {
      t1.dispose();
    }
  }

  private Part p1, p2; // never null
}
```

A detailed look

`Copy` succeeds atomically (and returns) if nothing before the pivot point throws an exception:

- The cloned parts are held in local variables `t1` and `t2`. The cloned parts are *not* used to set the `p1` and `p2` fields directly.
- The local variables are swapped with the fields *only* if they are not `null`. This maintains the invariant that `p1` and `p2` are never `null` (the calls to `other.p1.clone()` and `other.p2.clone()` do not check `other.p1 != null` or `other.p2 != null`).

`Copy` fails atomically (with an exception) if any of the methods before the pivot point throw an exception:

- The `finally` blocks will dispose of any cloned parts.
- The exception will signal that the copy failed.
- The target of the copy will be unchanged.

A potential problem or two

One potential problem with `copy` as it stands is that if a `clone()` call returns null there will be no exception, `copy` will return, and the target of the copy will remain *unchanged*. This means you don't know the state of the object if the call to `copy` returns (rather than throwing an exception). There are at least two solutions to this problem:

- You could make `copy` return a boolean and return false if any call to `clone()` returns `null`. (The `finally` blocks will still dispose of any successfully cloned parts.)
- You could throw an exception if a call to `clone()` returns `false`.

I prefer the latter option, partly because it fits with the exception-for-failure model, but mostly because it just seems right. We've already established that `other.p1` and `other.p2` are never `null` and `null` simply isn't a clone of something that's not `null`.

There's another copy subtlety worth mentioning. It concerns the calls to `setParent`. These calls nicely illustrate that a whole-part relationship is, in general, a two-way relationship. If you follow the code through carefully, you'll see that once the swaps have taken place, the cloned parts (now referred to by the `p1` and `p2` fields) see the target of the copy (`this`) as their parent (which is fine), but that the old parts (now in `t1` and `t2`) *also* see the target of the copy (`this`) as their parent. In other words, if a Whole holds N parts then N*2 parts will simultaneously think they're in the same Whole. In reality, this probably won't be a problem because half of the parts are about to be disposed of.

An interface: Disposable resource

The strategy I used in my attempt to progress through the jungle is an interface. The `Disposable` interface embodies the Disposal Method pattern – it creates a method you can explicitly call to dispose of a resource at a *known* point in the code (this interface is noticeable by its absence from the Java SDK).

```
public interface Disposable {
  void dispose();
}
```

A specification: Stack of Disposable resources

The next step is to create something that holds a stack of `Disposable` objects. The purpose of this class will be to hold a number of resources. I call it a stack because it effectively plays the part of the control flow stack in C++. That is, a stack in the sense of the opposite of a heap; the stack that, in C++, holds local objects and calls their destructors when they go out of scope. `DisposableStack` is an example of the Composite pattern (although I don't actually require the composite in this article).

```
public final class DisposableStack
implements Disposable {
  public DisposableArray(int
fixedCapacity);
  public void push(Disposable resource);
  public void clear();
  public void dispose();
  ...
}
```

A beginning

Now consider what would happen if `Part` implemented `Cloneable` *and* `Disposable` (I will remove this assumption later):

```
public class Part implements Cloneable,
Disposable {
  ...
}
```

With these pieces of the puzzle in place you can collapse the multiple nested `try`/`finally` blocks in `Whole.copy` (one per part) down to just a single `try`/`finally` block. Notice that each resource is held twice, once in a local variable (eg `t1`, `t2`) and once inside the `DisposableStack` (`resources`). The trickiest part is the last three statements before the `finally` block. These statements ensure that a successful copy (one that passes the pivot point) disposes of the old `Part` resources just swapped from `p1`, `p2` into `t1`, `t2`:

```
public class Whole {
  ...
  public void copy(Whole other) {
    DisposableStack resources =
                new DisposableStack(2);
    try {
      Part t1 = (Part)other.p1.clone();
      push(resources, t1);
      Part t2 = (Part)other.p2.clone();
      push(resources, t2);
      t1.setParent(this);
      t2.setParent(this);
      Part swap1 = t1;
      t1 = p1;
      p1 = swap1;
      Part swap2 = t2;
      t2 = p2;
      p2 = swap2;
      resources.clear();
      resources.push(t1);
      resources.push(t2);
    }
    finally {
      resources.dispose();
    }
  }
  private static void push( DisposableStack
                resources, Part resource) {
    if (resource == null) {
      throw ...
    }
    resources.push(resource);
  }
  private Part p1, p2; // never null
}
```

If a call to `clone()` returns `null`, the attempt to add it to the `resources` collection must throw an exception. We'll handle this in a `Whole` helper method rather than in `DisposableStack` so that `DisposableStack` can stay a general purpose class. If all the parts are cloned and added to the `resources` collection, the part specific calls take place (`setParent`) and then the swaps are performed. The swaps must not throw an exception. After the swaps, the part clones (not `null`) are in `p1` and `p2`, and the old parts (also not `null`) are in `t1` and `t2`. The `resources` stack is cleared, and the old parts are pushed in ready to be disposed of in the `finally` block. The code tells us the critical exception guarantees of the methods:

- `DisposableStack.clear()` must never throw an exception. This is because the call to `clear()` happens after the pivot point.
- If a `DisposableStack` (eg `resources`) is initialized with a constructor argument `N`, then after a call to `resources.clear()` the next `N` calls to `resources.push(resource)` must not throw an exception. Again, this is because these calls happen after the pivot point.
- `DisposableStack.dispose()` should allow any exception that arises from it to escape. `DisposableStack` is a general purpose class and cannot judge the severity of a disposal through an interface.

An implementation: Stack of Disposable resources

How can we guarantee these constraints? You might think about using a collection class such as `ArrayList`. Can `ArrayList` methods throw exceptions? I'm not sure and I'm not going to bother looking because there is a better solution: use an array. This will not only give us complete control over exceptional behaviour it will probably also makes the solution a little faster. Notice that this version disposes of the resources in a last-in last-out fashion (which seems appropriate), and also avoids the need for any casting:

```
public final class DisposableStack
implements Disposable {
  public DisposableStack(
                int fixedCapacity) {
    resources =
          new Disposable[fixedCapacity];
    at = 0;
  }

  public void push(Disposable resource) {
    // PreCondition(resource != null)
    // PreCondition(at < resources.length)
    resources[at] = resource;
    ++at;
  }

  public void clear() {
    at = 0;
  }
```

```
    public void dispose() {
      while (—at > 0) {
        resources[at].dispose();
      }
    }
    private final Disposable[] resources;
    private int at;
  }
```

A refinement: Null Object pattern

One glitch in this implementation is that if you push `null`, bad things will happen. You could solve this by checking for `null` in a `push` precondition. Another way is to use the Null Object pattern, like this:

```
public final class NullDisposable
implements Disposable {
  public static final NullDisposable
instance = new NullDisposable();
  public void dispose() {
    // all done!
  }
}
public final class DisposableStack
implements Disposable {
  ...
  public void push(Disposable resource) {
    // PreCondition(at < resources.length)
    resources[at] =
        resource != null ? resource :
NullDisposable.instance;
    ++at;
  }
  ...
}
```

A refinement: swap method

Something else that bothers me about this solution is the lack of a swap abstraction. If you study Alan's original code you'll see that the swap cannot be factored out because it swaps local variables that are disposed in their following `finally` blocks. However, at the cost of introducing a temporary object, it is possible:

```
public class Whole {
  ...
  public void copy(Whole other) {
    DisposableStack resources =
                  new DisposableStack(2);
    try {
      Whole copy = new Whole();
      copy.p1 = (Part)other.p1.clone()
      push(resources, copy.p1);
      copy.p2 = (Part)other.p2.clone();
      push(resources, copy.p2);
      copy.p1.setParent(this);
      copy.p2.setParent(this);
      copy.swap(this);
      resources.clear();
      resources.push(copy.p1);
      resources.push(copy.p2);
    }
    finally {
      resources.dispose();
    }
  }

  public void swap(Whole other) {
    Part swap1 = p1;
    p1 = other.p1;
    other.p1 = swap1;
    Part swap2 = p2;
    p2 = other.p2;
    other.p2 = swap2;
  }
  ...
  private Part p1, p2; // never null
}
```

Note that this "solution" requires a default `Whole` constructor that creates an empty `Whole` object, that is, one that contains no resources. The idea is that you create an empty whole object, and then gradually fill it. If you've already implemented the default constructor and it doesn't do this you can simply create a private constructor with a dummy argument.

```
public class Whole {
  ...
  public Whole copy(Whole other) {
    DisposableStack resources = new
DisposableStack(2);
    try {
      Whole empty = new
Whole(Empty.instance);
      // as before
    }
    ...
  }

  private static final class Empty {
    public static final Empty instance =
        new Empty();
  }

  private Whole(Empty unused) {
    // all done
  }
  ...
}
```

A bug: shallow swap

This would be a reasonable solution if it worked but unfortunately it doesn't. Before the swap, both objects are well formed. The target of the copy (this) has its parts and these parts know this is their parent. Similarly, the copied object has its parts (clones of this's parts) and these parts know copied is their parent. All well and good! The problem is that the swap only swaps the references; it's a shallow swap and it needs to be a deep swap. After the swap, the parts will be referring to the wrong parents. To swap a whole you have to be able to swap its parts. It's not enough to just swap the references.

```
public class Part implements Cloneable,
Disposable {
  ...
  public void swap(Part other) {
    // swap all fields, must not throw
    ...
  }
}

public class Whole {
  ...
  public void swap(Whole other) {
    // shallow
    Part swap1 = p1;
    p1 = other.p1;
    other.p1 = swap1;
    Part swap2 = p2;
    p2 = other.p2;
    other.p2 = swap2;
    // deep
    p1.swap(other.p1);
    p2.swap(other.p2);
  }
  ...
}
```

A refinement: copy constructor

Let's stop for a moment to think about what we've just done. We've created an empty object so that we can gradually fill it with cloned parts. And remember that we're doing this so we can implement the copy method. With a little reflection the idea of using a copy constructor suggests itself. That way, we won't have to worry about the initial state of the object we're creating. (If you've already implemented the copy constructor and it doesn't do this you can simply create a private constructor with a dummy second argument as before). Should the copy constructor be public? Well, copy method is public so it seems reasonable.

```
public class Whole {
  ...
  public Whole(Whole other) {
    DisposableStack resources =
                  new DisposableStack(2);
    try {
      push(resources,
           p1 = (Part)other.p1.clone());
      push(resources,
           p2 = (Part)other.p2.clone());
      p1.setParent(this);
      p2.setParent(this);
      resources.clear();
    }
    finally {
      resources.dispose();
    }
  }
  ...

  private Part p1, p2; // never null
}
```

A final piece: copy method

With this copy constructor in place, the obvious (but wrong) way to write the `copy` method is as follows:

```
public class Whole {
  ...
  public void copy(Whole other) {
    Whole copy = new Whole(other);
    copy.swap(this);
  }
}
```

The problem is that `copy` does not dispose of the old parts after the swap. This is not C++ remember! Luckily we have exactly the right tool to fix this problem: `DisposableStack`.

```
public class Whole {
  ...
  public void copy(Whole other) {
    DisposableStack resources = new
DisposableStack(2);
    try {
      Whole copy = new Whole(other);
      copy.swap(this);
      push(resources, copy.p1);
      push(resources, copy.p2);
    }
    finally {
      resources.dispose();
    }
  }
  ...
}
```

A refactoring: resources as a field

This is now a reasonable solution. However, there is one last refactoring we can perform before returning to the assumption that `Part` implements `Disposable`. It revolves around the observation that there is a lot of pushing going on! A possibly more natural approach would be for the pushes to happen only at construction and for the clear to happen only at "disposal". The way to achieve this is to make the DisposableStack a field instead of a local variable. There are a number of points to note in this solution:

- If there is no exception the copy constructor *retains* the references to the parts. The assumption is that all the constructors will do this. To do this you need a Boolean variable because there is no way to query whether an exception is currently pending. (Another noticeable omission from the Java SDK; interestingly, I've been told that it is possible to determine whether an exception is pending if you drop down to the JVM level.)
- The `copy` method makes a copy using the copy constructor. If this succeeds it swaps itself with the copy. It then disposes of the parts of its *old self* (which of course are now held in the copy because the swap method swaps the resources field too). An issue you might like to consider is the possibility of an exception arising from the call to `copy.resources.dispose`. Should this exception be caught and suppressed? How valuable is the simple model of

use this would afford? Is it a deep philosophical truth (and holds in all applicable programming languages) that exceptions should never arise from destruction/finalization? Or is disposal different because the object being disposed is still in scope?

- It's noticeable how the `ResourceStack` is playing the role of a stack-based local variable in languages like C++ that supports scope based resources.

```
public class Whole {
  ...
  public Whole(Whole other) {
    boolean exception = true;
    try {
      push(p1 = (Part)other.p1.clone());
      push(p2 = (Part)other.p2.clone());
      p1.setParent(this);
      p2.setParent(this);
      exception = false;
    }
    finally {
      if (exception) {
        resources.dispose();
      }
    }
  }

  public void copy(Whole other) {
    Whole copy = new Whole(other);
    copy.swap(this);
    copy.resources.dispose();
  }

  public void swap(Whole other) {
    // shallow
    DisposableStack swap0 = resources;
    resources = other.resources;
    other.resources = swap0;
    Part swap1 = p1;
    p1 = other.p1;
    other.p1 = swap1;
    Part swap2 = p2;
    p2 = other.p2;
    other.p2 = swap2;
    // deep
    p1.swap(other.p1);
    p2.swap(other.p2);
  }

  private void push(Part resource)
    if (resource == null)
      throw ...
    }
    resources.push(resource);
  }

  private DisposableStack resources =
                  new DisposableStack(2);
  private Part p1, p2; // never null
}
```

An assumption revisited: Part Disposer

Now let's return to the original assumption. What if `Part` does not implement the `Disposable` interface? Simple, just use an `Adapter`.

```
public final class PartDisposer
                    implements Disposable
  public PartDisposer(Part adapted) {
    // PreCondition(adapted != null)
    adaptee = adapted;
  }
  public void dispose() {
    adaptee.dispose();
  }
  private final Part adaptee;
}
```

Fine, but how do you use the adapter? There is a final trap we must take care not to fall into. The obvious (but flawed) way to use this adapter is as follows:

```
public class Whole {
  ...
  private void push(Part resource) {
    if (resource == null) {
      throw ...
    }
    resources.push(new
PartDisposer(resource));
  }
  ...
}
```

The problem is that if the creation of a new `PartDisposer` throws an exception the copied `Part` will not be pushed onto the `resources` stack. There are a number of ways to fix this. One is via a careful use of a `try/finally` block in the `Whole.push` helper method:

```
public class Whole {
  ...
  private void push(Part resource) {
    if (resource == null) {
      throw ...
    }
    try {
      resources.push(
             new PartDisposer(resource));
      resource = null;
    }
    finally {
      if (resource != null) {
        resource.dispose();
      }
    }
  }
  ...
}
```

[continued at foot of next page]

Template Metaprogramming: Shifting Down a Gear

by Andrew Cheshire

Template metaprogramming (MP) in C++ is a powerful technique but the syntax used can be obscure and difficult to understand. Here I propose an alternative approach in which a subset of the standard C++ language is used to write template metaprograms in a natural and familiar style.

This article assumes either an understanding of template metaprogramming or a pretty good ability to absorb new ideas. If you want to read up on the topic before reading this article: see [1] online for Todd Veldhuizen's historical paper and some useful links; and [2] for a recent *Overload* article on the subject.

There are a number of forms of MP but in this document we use only the general-purpose one presented in Andrei Alexandrescu's *Modern C++ Design* [9]; I will refer to this form of MP as *AMP*.

Template Metaprogramming

C++ template-metaprogramming uses standard features of the language to achieve computation in the type-domain, at compilation time.

This means that computation is done on *types* rather than on *values*. This may sound bizarre but in practice it can aid both in design abstraction and in time/space efficiency. Applications of MP include high-performance numerical computing [3], matrix computation [4], reflection [5], dimensional analysis [6] and static configuration [7].

However, despite the power of the technique it isn't really used in the mainstream, and it's been 8 years since Erwin Unruh wrote the first MP program [8].

This may be due in part to a lack of suitable C++ compilers in the past but another reason must be that MP is not a *designed* part of the language: it's really an accident resulting from the interaction of several language features. And – as so often happens when something is used for other than its intended purpose – MP code can be obscure and difficult to understand.

factorial Example

Let's start by implementing the factorial function as a template metaprogram.

Here is a standard implementation of the factorial function:

```
int factorial(int n) {
  if (n==1) {
    return 1;
  }
  else {
    return n*factorial(n-1);
  }
}
...
// example call
int x = factorial(5);
```

The two branches of the conditional statement return the two possible outcomes:

- when `n==1` the function simply returns 1
- when `n!=1` the function returns the result of calling itself recursively with an argument of `n-1`

Here is an MP implementation of the factorial function[1]:

```
// definition
template<int n>
struct factorial {
  enum {RET = n*factorial<n-1>::RET};
};
// partial specialization
template<>
struct factorial<1> {
  enum {RET = 1};
};
...
// example call
enum {x = factorial<5>::RET};
```

The definition and partial specialization of the factorial class template here give the two possible outcomes:

- when `n!=1` the result is given by the result of instantiating itself recursively with a template argument of `n-1`
- when `n==1` the result is simply 1

1 The historical use of `enum` in this context was originally a workaround for compiler limitations. On a compiler that supports the latest version of the standard "`const int RET = 1;`" is not only perfectly legal, but perhaps a more idiomatic usage

Conclusion

Has it been worth it? Is this version useful? As always there are opposing forces. This version is something of a sledgehammer cracking a nut when the number of parts is small. But as the number of parts increases, so does the depth of the try/finally nesting, and so the more attractive this version becomes. However, it does so at the cost of creating extra objects. It's also noticeable that this version involves a lot less work if the `Part` classes implement the `Disposable` interface, thus avoiding the need for `Adapter`s.

It's also important to consider that Java programs normally express copying through the creation of new objects rather than emulating deep assignment. Perhaps it's best to think of this solution to Tom Cargill's whole-part copy problem as showing just *one* way to use the `DisposableStack` class. The important thing, as Alan says, is that the exception safety guarantees make sense in Java and should be applied when writing or reviewing code. The problem is that the Java language offers almost no in-built support for this activity. `DisposableStack` is a class that can help.

In design, as in life, you learn more from making the journey than you do from reaching the destination. I look forward to more, as yet, unvisited trails.

Jon Jagger
jon@jaggersoft.com

Compare the possible outcomes of the function with those of the class. Although the factorial class looks very different from the factorial function they have the same logical structure :-

- return 1 if the parameter is 1
- return `n` times the factorial of `n-1` otherwise

The big difference between the two implementations is that the template computation happens at compile-time instead of when the program is run. Integer results of template computations are available as compile time constants, for example, in the expression `factorial<5>::RET`. This means that, for example, you could declare an array like this:

```
int buffer[factorial<5>::RET];
```

Types can be manipulated at compile time too, using `typedef` to name intermediate and final results in the same way that `enum` (or `const int`) is used to name integer values. There's an example which uses types later in the article.

The MP Execution Model

If you want to know what happens – in general – when you run a program in a given language you need to know its *execution model*: a specification of what happens when a program written in the language is run on a conforming implementation.

My first acquaintance with the idea of an execution model for MP was a talk by Gabriel Dos Reis at ACCU 2001 [10] in which he showed how C++ template metaprograms could be modelled in the Scheme language. Scheme is a good model for MP because both languages' execution models are essentially those of *functional* programming languages.

Dos Reis talked about *M-values* (*M* stands for *meta*) being the MP equivalent to *values* in most programming languages. An *M-value* is a type or anything else that can be manipulated at compile time.

AMP *M-values*:

1. template instantiation plays the role of a function call
2. template partial specialization provides conditional branching
3. `enum`s set local aliases for complex expressions and return integer results
4. `typedef`s set local aliases for complex types and return type results

This information is summarized in Table 1.

If you look back to the MP implementation of the factorial function you will see that it uses features 1, 2 and 3 with these pieces of code:

1. `factorial<n-1>::RET`
2. `template<> struct factorial<1>`
3. `enum {RET = 1}`

Using these language features sophisticated programs can be written (even a Lisp interpreter [11]) but it's not easy: the syntax is unhelpful and the programs can't be effectively debugged (you can't single-step through a compilation ...).

Modelling AMP in Another Language

The C++ MP code in the factorial example and in Listings 1 and 2 can be difficult to follow but the abstract execution model is very simple. It has *single assignment* (variables are initialised on declaration and cannot be modified thereafter), *conditional selection* (choice, as in `switch` or `if`), but no *iteration* (no `for`, `while` or `do` loops – recursion is used instead).

Given the simplicity of the execution model we can consider writing programs in a *source language* with a more suitable syntax than C++ MP code. Programs written in this source language could be automatically translated to the correct C++ MP code.

But which language? A functional language such as Scheme or Haskell would have the right sort of execution model, but would not be taken up by many C++ programmers.

Instead I propose that we actually use a subset of C++ itself as the source language, which I've called **typeshift**. This will by definition be familiar to C++ programmers and there are other advantages, as we shall see.

Here's the factorial example in **typeshift** :-

```
int factorial(int n) {
  switch(n) {
    case 1:
      return 1;
    default:
      return n*factorial(n-1);
  }
}
```

Feature	MP Implementation	Example Code

conditional branching	template partial specialization	`template<typename T>`
		`struct Setup { /* code for general case */ };`
		`template<>`
		`struct Setup<Null> { /* code for T==Null */ };`
		`// OR`
		`template<int N>`
		`struct Factorial{ /* code for general case */ };`
		`template<>`
		`struct Factorial<0> { /* code for N==0 */ };`

integer expressions	enum	`enum {N=Length<T::Tail>+1};`
set integer alias	enum	
return integer results	enum	`enum {value=N};`
call MP "functions"	template instantiation	`typedef Next<T>::value NextType;`
set type alias	typedef	
return type result	typedef	`typedef Next<T>::value value;`

Table 1: AMP Execution Model

```
// typelists – standard approach. This
// particular approach even works on MSVC 6.0.

namespace typelists {
  // a list of types, each element has a head
  // and a tail - the head is one of the types
  // in the list, the tail is either another
  // list of types or Null
  template<typename HeadT, typename TailT>
  struct List {
    typedef HeadT Head;
    typedef TailT Tail;
  };

  // terminates type lists
  struct Null {};

  // - find the length of a type list -
  // ... the general case – the length is 1
  // more than the length of the tail
  template<typename T>
  struct Length {
    enum{RET=Length<typename T::Tail>::RET+1};
  };
  // ... the case of Null – the length is zero
  template<>
  struct Length<Null> {
    enum {RET = 0};
  };
};
```

Listing 1: tm1.h

It is, of course, the same code as the standard factorial function we gave earlier (which should be no surprise).

This code is similar but not identical to the factorial function we gave earlier. It uses `switch` instead of `if` because **typeshift** will not initially support `if`.

The point of **typeshift** is this: you write a program in the **typeshift** language and then use a translator to convert it to C++ MP code. The translator would convert the above factorial program to this MP program (again, this code is identical to that of the earlier example):

```
// definition
template<int n>
struct factorial {
  enum {RET = n*factorial<n-1>::RET};
};
// partial specialization
template<>
struct factorial<1> {
  enum {RET = 1};
};
```

So now you can write a program in something resembling everyday C++ and have it converted to a template metaprogram.

Now this is just a tutorial example: a MP factorial program isn't very practical because it has to be recompiled every time you want to compute a different factorial. An MP program is only useful if you can make use of the results of the compile-time computation when the program is run. Later on we'll look at an admittedly abstract but genuinely useful example of template metaprogramming.

```
// try out typelists – standard approach
#include <iostream>
#include "tm1.h"
using namespace typelists;

int main() {
  // declare a type list
  typedef List<int, List<double,
              List<char, Null> > > basictypes;
  // compute the length of the type list. the
  // whole right-hand side below is evaluated
  // at compile time (the result is 3)
  int n = Length<basictypes>::RET;
  std::cout << "n = " << n << std::endl;
  return 0;
}
```

Listing 2: tm1.cpp

typeshift

typeshift is a *small* subset of C++. We take only those features which are required to support the AMP execution model:

- classes/structs with simple data and function members
- variable initialisation
- switch statements
- return statements

We do, of course, need to be able to represent *types* so that we can support template type-parameters. You might think that this is where it gets complicated, but in fact it doesn't – in "real" C++ types are very different from values but in **typeshift** they are quite similar: everything is just an *M-value.*

typeshift uses distinguished identifiers like type, fixed_type and template_type to declare variables (and subclasses) which to represent types and such variables behave as they do in MP.

This execution model of AMP as supported by **typeshift** is shown in Table 2 [next page] which you will want to compare with Table 1. Remember that this is only the first version of **typeshift**: over time its syntax and semantics will be extended to make it even easier to write template metaprograms.

I have not yet looked into mapping other forms of MP into **typeshift** but I hope that we will only need to add a few more features of C++ to the language for it to be able to model any current use of MP.

Typelist Example

Here's an example of MP which uses types. Listings 1 and 2 are an implementation of the *typelist* data-structure from Alexandrescu [9] (not actually his implementation). They demonstrate how AMP can be used to implement a simple *type-data-structure* (a linked list of types) and a *type-function* which finds the length of such a list.

Listings 3 and 4 [next page] implement the same program, but in **typeshift**.

The .h file in Listing 3 is very different from the .h file in Listing 1 but if you read them while referring to Tables 1 and 2 you should be able to follow how the two sets of code correspond.

Feature	typeshift Implementation	Example Code

conditional branching (on types)	switch	`switch(T) {`
		`  case Null: // code for T==Null`
		`  default:  // code for general case`
		`};`
		`// OR`
		`switch(N) {`
		`  case 0:  // code for N==0`
		`  default: // code for general case`
		`}`

integer expressions	expression	`int N=Length(T.Tail)+1;`
set integer alias	definition	
return integer results	return	`return N;`
call MP "functions"	function call syntax	`type NextType=Next(T);`
set type alias	type definition	
return type result	return	`return NextType;`

Table 2: typeshift Execution Model

```
// typelists – typeshift approach
#include <ts_runtime.h>

namespace typeshift {
  namespace meta {
    namespace _typelist {

// a list of types, each element has a head
// and a tail - the head is one of the types in
// the list, the tail is either another list of
// types or fixed_type::null
struct List {
  // constructor
  List(const type& HeadT, const type& TailT)
    : Head(HeadT), Tail(TailT) {}
  // members
  const type& Head;
  const type& Tail;
};

// - find the length of a type list -
int Length(const type& T) {
  switch (T) {
  // the case of fixed_type::null – the length
  // is zero
  case fixed_type::null:
    return 0;
  // the general case – the length is 1 more
  // than the length of the tail
  default:
    return Length(dynamic_cast<const
                          List&>(T).Tail)+1;
  }
}
}}};
```

Listing 3: tm2.h

One important difference between the two sets of code is the namespace: it was `typelist` in the original C++ MP but is `metatypeshift::meta::_typelist` in the **typeshift** code. This is because we propose to signal the presence of **typeshift** code by enclosing it in a distinguished namespace `whose name begins with typeshift::meta`, or in a namespace derived from this. An enhanced C++ compiler or external tool can use this to pick out the **typeshift** code from the "normal" C++ code.

```
// typelists – try out typeshift approach

#include <iostream>
#include "tm2.h"
using namespace typeshift::meta_::typelist;

int main() {
  typedef List<int, List<double,
           List<char, fixed_type::null> > >
           basictypes;
  int n = Length<basictypes>::RET;
        // the generated code still uses RET
  std::cout << "n = " << n << std::endl;
  return 0;
}
```

Listing 4: tm2.cpp

The .cpp file in Listing 4 is practically identical to the .cpp file in Listing 2 – only the namespace identifier and the name for the null list-terminator are different. This is because I don't propose changing the syntax of *references* to the names in the generated C++ MP code (at least, not yet) because that is going to be rather more difficult to handle than just transforming the *definition*.

Coins Example

In [10] Gabriel Dos Reis gives an example from Abelson & Sussman [12] of a coin-counting program, first of all in Scheme and then in C++ MP code. Given an amount of money (in pennies) the program returns the number of ways in which change can be given using British coins.

Dos Reis's C++ MP code along with a test rig is given in Listings 5 and 6.

I translated it into **typeshift** and this (again, with a test rig) is given in Listings 7 and 8.

```
// coins – C++ MP approach
// This code is reprinted by permission from
// Gabriel Dos Reis' ACCU 2001 talk [10]

template<int coin_kind>
struct coin_value { };
template<>
struct coin_value<1> { enum {value = 1}; };
template<>
struct coin_value<2> { enum {value = 5}; };
template<>
struct coin_value<3> { enum {value = 10}; };
template<>
struct coin_value<4> { enum {value = 25}; };
template<>
struct coin_value<5> { enum {value = 50}; };

template<int amount, int coin_kinds, bool stop>
struct count_change_helper {
  enum {remaining_coins = coin_kinds - 1};
  enum {remaining_amount = amount
        - coin_value<coin_kinds>::value};
  enum {value =
         (count_change_helper<
            remaining_amount, coin_kinds,
            (remaining_amount <= 0 ||
            coin_kinds == 0)>::value
          +
          count_change_helper<
            amount, coin_kinds - 1,
            (amount <= 0 ||
            remaining_coins == 0)>::value)
  };
};

template<int amount, int coin_kind>
struct count_change_helper<amount,
                           coin_kind, true> {
  enum {value = (amount == 0) ? 1 : 0};
};

template<int amount>
struct count_change {
  enum {value = count_change_helper<amount, 5,
        (amount <= 0 || 5 == 0)>::value };
};
```

Listing 5: coinct.h

```
// coins – try C++ MP approach
#include "coinsct.h"
#include <iostream>

int main() {
  const int amount = 23;
  std::cout << "ways for " << amount << ": "
   << count_change<amount>::value << std::endl;
  return 0;
};
```

Listing 6: coinct.cpp

```
// coins – typeshift approach
#include <ts_runtime.h>

namespace typeshift {
  namespace meta {
    namespace coins {

int coin_value(int coin_index) {
  switch (coin_index) {
    case 1: return 1;
    case 2: return 5;
    case 3: return 10;
    case 4: return 25;
    case 5: return 50;
  }
}

int count_change_helper(int amount,
                        int coin_kinds, bool stop) {
  switch (stop) {
    case true:
      return (amount==0)?1:0;
    case false: {
      int remaining_coins = coin_kinds - 1;
      int remaining_amount =
                amount - coin_value(coin_kinds);
      int value = count_change_helper(
                  remaining_amount, coin_kinds,
                  remaining_amount <= 0 ||
                  coin_kinds == 0)
                  +
                  count_change_helper(amount,
                  coin_kinds - 1, amount <= 0 ||
                  remaining_coins == 0);
      return value;
    }
  }
}
int count_change(int amount) {
  int value = count_change_helper(amount, 5,
                      amount <= 0 || 5 == 0);
  return value;
}
}}}
```

Listing 7: coinrt.h

```
// coins – try typeshift approach
#include "coinsrt.h"
#include <iostream>
#include <sstream>

int main() {
  const int amount = 23;
  std::cout << "ways for " << amount << ": "
    << typeshift::meta::coins::count_change(amount)
    << std::endl;
  return 0;
}
```

Listing 8: coinrt.cpp

If you compare the two sets of listings I think you will agree that the **typeshift** version is clearer. But there's more – **typeshift** is a subset of C++ so we can actually compile and run a **typeshift** program *without* translating it to C++ MP code.

Both programs can be built[2] with gcc 2.91.66. When compiling the C++ MP code pass `-ftemplate-depth-99` to g++ to maximise the size of the problem that can be handled.

When run with the same input the programs give identical results so they are in some sense operationally equivalent. However, the C++ MP code computes the answer at *compile-time* but the **typeshift** code computes the answer at *run-time*.

This operational equivalence means that metaprograms can be written in **typeshift** and tested and debugged in the value-domain. Only then need the program be transformed to C++ MP code for execution at compilation time.

Even **typeshift** programs that use *types* can be compiled, run and debugged in this way using the **typeshift** '`type`' class library.

Implementation

There are two ways of implementing **typeshift**:

- by extending a C++ compiler
- by writing an external tool

A C++ compiler essentially already has the mechanism to compile **typeshift** because syntactically and semantically it is a true subset of C++. Two changes would be needed:

- Only a subset of C++ features are allowed in **typeshift** so the standard parser would need to be adapted to handle this restricted "dialect".
- The **typeshift** code needs to be transformed into the C++ MP code before it is compiled in the normal way. This is a fairly straightforward transformation.

There should be no impact on compilation outside the `typeshift::meta_xxx` namespace because from the point of view of code outside the namespace the names defined inside the namespace are from the transformed C++ code. It is not possible for code outside the namespace to "have any knowledge of" the original **typeshift** code.

Implementation by an External Tool

A proof-of-concept **typeshift** pre-processor is currently under development, and should be available from `http://www.typeshift.org` when this article is published. It will be downloadable in the form of source code for a C++ program released under the GPL (GNU General Public License [13]).

2 MSVC6 won't compile the C++ MP program because it does not support partial template specialisation (I have not tried MSVC7)

Conclusion

Template metaprogramming has traditionally been viewed as an esoteric and obscure area of C++, but using **typeshift** metaprograms can now be written (and even debugged) in a familiar language. Hopefully this will lead to metaprogramming being used much more widely in C++.

Andrew Cheshire
andyc@intersystems.com

Acknowledgements

Many thanks to Gabriel Dos Reis for his advice and encouragement, and also for his permission to quote the example program from [10].

Thanks too to Mark Radford for his invaluable comments and suggestions, which have certainly made the article easier to understand.

References

[1] Veldhuizen, 1995: Template Metaprograms (`http://www.osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html`)

[2] Walker, 2001: "Template Metaprogramming: make your compiler work for you" Overload 46

[3] Blitz (`http://www.oonumerics.org/blitz/whatis.html`)

[4] Generative Matrix Computation Library (`http://www-ia.tu-ilmenau.de/~czarn/gmcl/`)

[5] Giuseppe Attardi, Antonio Cisternino: Reflection support by means of template metaprogramming (`http://citeseer.nj.nec.com/451721.html`)

[6] John J. Barton, Lee R. Nackman: "Scientific and Engineering C++: Dimensional Analysis" C++ Report, vol 7 p39, Jan. 1995

[7] Ulrich Breymann, Krzysztof Czarnecki, Ulrich Eisenecker: Generative Components: One Step Beyond Generic Programming (`http://home.t-online.de/home/Ulrich.Eisenecker/dag.htm`)

[8] Unruh, 1994: Prime number computation (ANSI X3J16-94-0075/ISO WG21-462)

[9] Alexandrescu, 2001: Modern C++ Design (Addison-Wesley, ISBN 0-201-70431-5)

[10] Dos Reis, 2001: Metaprogramming in C++ (`http://www.cmla.ens-cachan.fr/~dosreis/C++/`)

[11] Czarnecki, Eisenecker: metalisp.cpp (`http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm`)

[12] Abelson and Sussman, 1985: Structure and Interpretation of Computer Programs (`http://mitpress.mit.edu/sicp`)

[13] GNU General Public License (`http://www.gnu.org/licenses/gpl.html#SEC1`)

[14] Dos Reis, 2002: (personal communication)

Copyrights and Trade marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trademark and its owner.

By default the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission of the copyright holder.

The Philosophy of Extensible Software

By Allan Kelly

In Overload 49 I wrote about extensible software, it's a theme I'm going to continue with for a couple more articles. If I were to attempt to summarise my philosophy of software development in one sentence it would probably be: Software must stay soft, malleable. The discipline of extensibility is the tool which best helps us achieve this. So, although I'm declaring an intention to stick with software extensibility for a little while, I'm actually intending to look at how we can keep our software flexible and open to change.

How does software resist change?

In order to understand how we can keep our software malleable it is worth considering how software loses this quality. After all, when you start a new project, the world is your oyster, you can take the project in any direction.

Ripple effect

A change in one module is just that: a change in one module of the system. The principles of abstraction and data hiding tell us that we should be able to make changes to hidden code which have no side-effects elsewhere. Often though we find there is a *ripple effect*.

If we drop a small stone into a pond we see ripples spread out across the pond. The water surface is perfect for propagating the effect. Software is, if anything, better than water at propagating the effect. Changing the interface of a module means we must make corresponding changes to the use of the interface elsewhere.

Speaking of *the interface* means more than just the class definition in some header file. This is the most clearly stated part of the interface but it is like an iceberg, there is much we can't see. The compiler can check the function signatures but can it check the comments?

Consider some code:

```
// SerialPort.hpp
// Read up to bufferSize characters from
// serial port
// store in buffer and return ptr to
// buffer
char* ReadSerialPort(char *buffer,
                     int bufferSize);
...

// SerialPort.cpp
// Read at least bufferSize chars from
// serial port
// store in buffer and return ptr to end
// of buffer
char* ReadSerialPort(char *buffer,
                     int bufferSize) {
  assert(buffer != 0)
  assert(bufferSize > 16)
  ...
}
```

The interface the compiler can check only forms part of the interface. The comments form another vital part of the interface and in this case they differ. Which set is correct? The developer must break the abstraction and look *under the hood* to see what is happening.

This is typically how a ripple starts, we've found something which isn't quite right, not so wrong it breaks the program but not good either.

Both sets of comments fail to tell us that the buffer supplied must be pre-allocated and at least 16 characters long. Sure, it may seem logical to allocate the buffer before calling the function but the C function `time` never worried too much about this.

At one level this is implementation detail, at another level it is interface, we can fix the comments, we can even change the function signature to reduce the problems with this function:

```
// SerialPort.hpp
// Read up to bufferSize characters from
// the buffer
// store in buffer and return number of
// bytes read
// bufferSize must be at least 16 bytes
// return value < bufferSize

int ReadSerialPort(char *buffer,
                   int bufferSize);
```

Now we have created a ripple effect, not only this module but several others will need recompiling. Wherever this function is used we must now change the code, our change has slipped out of our chunk. While it would be a pretty relaxed compiler that still allowed you to write:

```
char* buf = ReadSerialPort(buffer, 32);
```

A developer in search of a quick fix may be tempted to write:

```
char* buf = (char*)
ReadSerialPort(buffer, 32);
```

While `static_cast` will refuse this `reinterpret_cast` has no such qualms, and as demonstrated the old-style casts are still in the language and available.

The general rule of ripple effect is that *he* who made the ripple has to stop it, you find yourself running around all over the code, fixing ripples where they appear. This is all but impossible if you don't have a reliable build process – if you can't integrate your code easily then you can't tackle these issues. A good source code control system is essential in case things go wrong, or time runs out, and you need to back out your changes.

Nor do we have perfect foresight, we may grep the code for every instance of `ReadSerialPort` before we make our change, but we can't expect to find every case. Just searching the code may be a bigger job than actually performing the change. A logical directory structure is important here, if our code is scattered over several dozen disparate directories on different hard discs then what chance have we got of finding it?

Suppose we now find:

```
// pointer to general read function
typedef char*
          (*ReadPortFunc)(char*, int);

// read function
char* ReadData(ReadPortFunc reader) {
  int bufferSize = 256;
  char* buffer = new char[bufferSize];
  memset(buffer, 0, bufferSize);
  return reader(buffer, bufferSize);
}
```

Instead of diminishing, our ripple has grown. We can fix this, but suppose we find:

```
char* ReadUsbPort(char *buffer,
                  int bufferSize);
char* ReadKeyboardPort(char *buffer,
                       int bufferSize);
```

Do we fix these functions too? Or fudge it? The ripple is not just a simple compile time fix now, it has uncovered a bigger problem, and while we may have the code in a compilable state, we (should) feel a certain moral commitment to fixing this problem. The ripple has grown.

Ripples like this, and fear of ripples, is one of the main ways code resists change.

Friction of change

The ripple effect demonstrates the friction that can occur when changes are made. If these changes are within the same module then the friction is less because the changes are not visible elsewhere. The bigger the change, the more modules involved, the greater the friction. When a system is changing rapidly, dividing it up can be counter productive because there is a constant friction as changes ripple out of one module and into others.

But friction between modules comes in other forms too. Where there are several developers on a project there is always the opportunity for conflicting changes to happen. While exclusive locking through source code control can help, it is not a complete answer. At best it forces one developer to wait while another completes a change, the second developer then has the task of integrating the change with their requirements. Non-exclusive locking systems can hide this problem until the second developer checks their code in but the same problem arises.

Either way, friction is generated because two developers must co-ordinate their actions. If the developers are located in different teams, or even different countries, the friction is much greater still.

When a change introduces a new dependency into a module, say a new file must be `#include`'d, the initial friction may be small, a slightly increased build time. But when this changes the overall dependencies of the module, and in particular if this introduces a circular dependency the potential friction is greater still.

Observant readers may have noted the potential contradiction is talking of "ripple effect" and "friction" – after all ripples occur in frictionless water. Ripples are waves, and waves can only occur when two modules share a common boundary. Such a boundary propagates the wave – think of the way an earthquake wave carries.

Sound waves cannot travel in a vacuum, likewise software ripples cannot pass from one module to another if they are well spaced. Since our modules don't exist in isolation we can't place them in a vacuum, what we can do is try to minimise the friction at the boundary and thus minimise wave propagation by allowing each module to change without creating a wave beyond its own boundaries.

Process roadblocks

Software takes on many of the attributes of the organisation and process that creates it.

This idea is summarised as *Conway's law* – although the exact wording of the law differs. Jim Coplien's process pattern of the same name has the solution "Make sure the organisation is compatible with the product architecture."

Where an organisation is conservative and resists change their software will too. This may manifest itself in many ways: a business which resists change may create code which resists change, or, it may mean managers refuse to allocate time for modifications.

This can be a frustrating position for a software developer, they may know of a bug, they may know how to fix it but they may be refused permission to deal with it. Or, to fix it may require raising a bug report, having the work prioritised, authorised, scheduled, changed, tested, signed-off and released. Sure we need a process, but we must not put the process before the product. In process-centric organisations we find managers who know the price of everything but the cost of nothing.

Some organisations refuse to recognise refactoring as an exercise. "If it processes data, it can't be broken, can it?" "Reworking something means you made a mistake, right?"

Developers have refactored code since the beginnings of time, but only with the publication of Martin Fowler's book (2000) has it been a respectable activity. Unfortunately, it is still not an acceptable activity in many organisations. Failure to refactor code makes it more rigid, as we put change upon change it becomes inflexible and set in its way. Unfortunately it still processes data.

This is like not servicing a car, it continues working, there is no apparent problem, but the further you get beyond an oil change the more damage is being done to the internals. Like a car, over time software changes and without active attempts to improve the quality it invariably deteriorates.

Development is a learning process

As we develop software we learn, we learn more about the problem domain that the software addresses and we learn more about our solution domain – the tools we have used to address it. Naturally, this leads to new insights into both.

We also have time to dwell on problems and issues. We may take a week to draw up an class hierarchy, but we have the rest of our lives to rethink it and consider how we could have done it better. This can make life hard for us if we come to believe we made a mistake, or no longer agree with our original designs – or just see a better way. Maybe what once seemed a brilliant design now seems top heavy, or inefficient, or simplistic. Don't

be too hard on yourself, admit you made mistakes if necessary. If we don't do this we will not move forward, it is now us who are resisting change.

How does extensibility work?

Extensibility works because it forces an approach to problems based on:

- **An up front design which allows for addition**
  This is not to make a case for big up front design – quite the opposite in fact. Big up front design assumes you can design the entire system up front. An extensible design accepts you can't design everything in advance, instead it provides a light framework which can allow for changes.
  In some ways this is similar to the STL separation of container and algorithm. The STL doesn't claim to know all the algorithms that may be used with a container, but instead provides a mechanism (iterators) which allow algorithms to be added later.
- **Additions to be made in small, incremental steps**
  It is possible to produce an extensible system where the increments are big, take our command pattern example. The commands could be small, "Put the kettle on", rather than big: "Take over the world." If we make our commands too big we lose the element of extensibility, the original problem is relocated inside a single command, which is effectively the entire system.
- **Work elements to be separated into comprehensible units**
  Computers may run programs and source code may be compiled by a tool, but it is humans who have to read and understand the system. There is a human factor to all of this, just because we can write an immensely complex piece of code doesn't mean we should. Anyone who has tried to maintain by hand code that was originally produced by a code generator will have seen this problem – indeed Perl scripts exhibit the same problem at times. So, keep each unit at a human level.

How does extensibility help?

To achieve these objectives we need to emphasis traditional software development issues: high cohesion, low coupling, interface-implementation separation, and we need to manage our dependencies, and develop build procedures to perform constant integration. This imposes a discipline on our development.

Extensible design fits well with the principles advocated by the Agile methodologies and iterative development. It allows functionality to be implemented in small steps as required, thus it dove-tails with the minimal implementation, iterative development and frequent re-prioritisation often advocated by Agile development.

In an extensible design we cannot afford for one chunk to be too closely coupled with other chunks. The very essence of the system is embracing change, it is accepted that additions will be continual, if one chunk of the system resists such change it will make the whole design unworkable. Thus, we have placed the friction of change centre stage. Normally we would rather not think about friction, it is a problem we want to go away. By elevating the issue we are directly addressing it, the whole system is designed around the idea of change through addition.

If you are the kind of person who likes new, green-field, system development this may sound pretty horrid. Basically, I'm suggesting you lay minimal foundations of specification, design and framework coding and make a quick dash for the maintenance phase where you actually fit the functionality.

True, I hold my hands up, I agree. However, in my defence, I claim I'm actually moving as much new development as possible into the maintenance phase of the project. Extensible software allows you to write new code well after your first release. Indeed, if you find a chunk of functionality is difficult to understand, buggy, or just not extensible throw it out and start again.

What is important is to get an up front design which can allow for continued development. This is like a shipyard building the hull and inner structure of a ship but leaving the fitting out and completion of the super-structure until after launching. Once the ship has enough structure to float it no longer need to monopolise a slipway – indeed it may even be fitted out by a different yard. Over the course of its life it will undergo continual maintenance even as it plies the high seas with the occasional refit, which may completely change its use.

Extensibility is not "reuse"

Extensibility is no magic bullet, it is just another technique in our toolbox for tackling software development. Nor is it a code word for "reuse". True, many of the properties emphasised by extensibility are the same ones preached for reusable code: low coupling, high cohesion, modularity, but these properties are advocated by most software engineering themes. Indeed, who would argue for tightly coupled systems?

It may be that, having an extensible system, with malleable code allows your technology to be transferred to another project – many of the properties required of an extensible system make transfer easier. One could easily imagine a word processor system which offered a standard system and a *beginner* version with fewer options, plus a *professional* version with more – the same way Volkswagen sell the Golf in tandem with the Skoda Fabia (low end), Audi A3 (high end) and specialisations like the Beetle and TT.

But, such platform transfer is deriving from the minimalist camp – "less is more" is the starting point. Extensible software development is no license to add bells and whistles to your code in the hope that someone may use them. Quite the opposite, extendable software should be free of bells and whistles, it should be minimal while allowing itself to be extended.

Striving for extensibility should imposes a discipline on development leading to fewer, cleaner, dependencies, well defined interfaces and abstractions with corresponding reduction in coupling and higher cohesion.

I've been here before....

I tried at the top of this essay to summarise my software development philosophy. Looking back at my contributions to Overload in the last few years I can see this as a common theme. To keep software malleable we must be aware of the dependency structure of the program, this I addressed in Overload 41 when I wrote about layering in software; dependencies start with include files (Overload 39 and 40).

I believe inline functions reduce abstraction, increase dependencies and generally complicate matters – hence my piece in Overload 42. (If anyone ever produces a subset of C++ I'd lobby

for inline functions to be first against the wall.). More recently my pieces have looked at how we view software as models (Overload 46) or abstractions (Overload 47).

Extensibility of software happens in all sorts of ways, at different levels within the system. It is important to have a view of your software as a living, growing, entity.

And finally

Extensibility is a technique for reasoning about our software. It is not new but it has been neglected as a technique in its own right. In part this is because it is often an attribute of other techniques – as noted in my previous essay it is implicit in many design patterns.

The properties that make up an extensible system are not confined to your source code – there are build systems, source code control, bug tracking, documentation, team mangement, and more. (I tend to call this the *logistics tail* and I'll expand on that idea next time.)

Extensible source code must be supported with extensible build systems, directory trees, database access mechanisms, and so on. These systems shine when we are able to embed within a process that is aligned to the design, source code, management and developers to form a logistic process which becomes a reinforcing strategy.

Allan Kelly
Allan.Kelly@bigfoot.com

Bibliography

Conway's Law comes in several different forms, Ward Cunningham's Wiki page gives several different forms at http://c2.com/cgi/wiki?ConwaysLaw

Jim Coplien's version of Conway's law as a process pattern is at http://www.bell-labs.com/user/cope/Patterns/Process/section15.html

Fowler, M., 2000; *Refactoring – Improving the design of existing code*, Addison Wesley, 2000