contents

credits & contacts

Editor:

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

Advertising:

Pete Goodliffe, Chris Lowe
ads@accu.org

Membership:

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

Readers:

Ian Bruntlett
IanBruntlett@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 - Software Quality

I'm dreadful at testing code, as I rarely bother to do it. Yeah, yeah, I know that I should. It's just a mental blind spot that I have.

Have you ever experienced the phenomenon of reading what you intended to write instead of what you have actually written? It's the same situation when you have a piece of broken code that you stare and stare at but can't see any errors. You then show it to a peer, claiming that the compiler is busted, the microprocessor is messing up, and the laws of physics no longer hold true. They look at it for ten seconds, go 'ugh', and point at 'if(x=y)...'. I've extrapolated this mental assumption upwards from the level of syntactic correctness to the level of semantic correctness. I assume that because I intended the code to work, it must actually work. Thankfully, due to some introspection, and a few people politely, and not so politely, pointing out that it would be nice if I tested my code properly, I am aware of my short-comings in this area.

But, I'm not the only person who is bad at testing; the majority of development organizations are bad at it too. I have never discovered a development organization that conducts excellent testing. Most organizations are content to test to the 'good enough' level. I don't believe the individuals involved intend to build a shoddy product, there's just something wrong with the way most people conduct testing.

Testing has always been a hard problem that few software engineering organizations put much intellectual effort into solving. In evidence, we have the fact that most test groups spend much of their time executing test cases by hand. I've experienced developer groups where the prevailing attitude was that all responsibility for testing lay with the Quality Assurance group. The developers throw the release candidate over a wall; the testers kick it about for a while, and then throw it back if they find something wrong with it. The cycle then continues, seemingly endlessly, until the QA manager and the engineering manager resolve their differences over a drunken fistfight in the carpark. [1]

A common solution to poor product quality is to throw more testers at the problem. Microsoft demonstrates that this doesn't work. They have a very high ratio of testers to developers, yet still produce 'good enough' quality products. Are they unwilling to produce quality products? I don't think so.

Another problem is that the role of test engineer is not well regarded by the software engineering profession at large. Most testers I've worked with don't want to be test engineers. They want to be developers. For them the test group is a stepping-stone into the sustaining engineering, or development engineering groups. I'm in favour of career progression, but when everybody wants out and nobody wants in there's a problem.

Is the software quality problem due to testers just not being very good at testing? I don't think so. I've encountered some very smart and highly productive, dedicated and motivated testers.

I've talked to colleagues with an Electrical Engineering background about their experience of testing in hardware development organizations. They have told me that test engineering is a well-respected role that is staffed by engineers who are just as qualified as engineers in the development organization, and that testers are involved in all phases of the development process to ensure that the product can be efficiently tested for correctness.

The point to learn here is that testability has to be designed into the system, and the engineering group bears the responsibility for doing that. Typically the development engineering team doesn't think much further than unit testing. More upfront thought needs to be put into integration and system testing at the specification and design stages of development.

I've tried to reflect this lesson into my current project. I've thought about testing right from the beginning of the development process. There's a line item on the product requirements that states that the product must be easy to test. The design and implementation follow through against this by exposing interfaces specifically for testing. In this case our user interface is a C++ API, so the test interface we chose was a scripting language. The scripting API includes methods that expose some of the internals as text strings, so that the test cases can make assertions about the internal state of the system.

I don't know the solution to the software quality problem, and I'm not sure if anyone really does, but there are a few small things that we could be doing to improve matters. In summary, designing testability into the system allows us to fully leverage the QA group and therefore improve software quality.

Apologies

In Overload 50 I mistakenly published a draft version of Pete Goodliffe's STL-style Circular Buffers article. There were a few minor errors in the text, which have been corrected in the online version, which can be found on Pete's website. [2]

John Merrells
merrells@acm.org

[1] That's a slight exaggeration.

[2] http://cthree.org/pete/circularbuffer.1.pdf

STL-style Circular Buffers By Example, Part Two

by Pete Goodliffe

In the first part of this series [1] we started to look at how to implement a template container class in the style of the C++ standard template library (STL). The example container we're working with is a circular buffer. We started off by writing a minimal circular buffer class, and through a number of exercises refined this basic class until we had a mostly STL-like container class.

We're still not quite there yet. By the end of this article we'll have covered all the necessary ground, and our container class can truly be used like any other STL container.

As in the previous article, the approach adopted here is to present a number of exercises for you to perform. This is more than a clumsy article-writing device. Experience shows that you only really learn something when you try to actually do it. You can read around a subject as much as you like; it's only when you pick up tools and start applying what you've read, when the metaphorical rubber hits the hypothetical road, that you really solidify your knowledge. These exercises are then a good opportunity to learn in a practical way. Put as much effort into them as you want to get out of the article.

So where are we?

So far we have a template container class with forward and reverse iterators. Unlike traditional circular buffer implementations we've provided `operator[]` for random access; this is largely to provide the iterator's implementation.

You can download an example copy of the code this far from `http://cthree.org/pete/cbuf.html`.

More constructors

Amongst the functions that we still need to implement for our `circular_buffer` class are a number of constructors. As an analogue, think about `vector`. It has a several useful constructors. So off we go…

Exercise #1
What operations does a C++ compiler automatically generate for a class? How do you prevent this from happening?

The compiler will automatically generate

- a default constructor, which just calls the default constructors for the class' members and bases,
- a default copy constructor, which performs a member-wise copy of all data members, and
- a default comparison operator, which performs a bitwise comparison of two objects.

Sometimes these defaults are fine and do just what we want. Often when creating more adventurous classes (with dynamically created data members) they're not at all suitable and we have to provide our own, or just prevent the compiler from emitting its defaults.

If we provide our own versions, the defaults are suppressed. This is of particular note for constructors – *any* constructor you provide, no matter what signature it has[1], will suppress the parameterless default constructor. If you want to have a parameterless constructor you will then have to provide it, even if it does exactly what the compiler generated version would.

If you don't want to supply the operations but the compiler generated versions are wrong, you can suppress them by *declaring* the functions, but not *defining* them. For this to make any sort of sense, you should declare the functions as private members of the class.

Constructor 1: Copy constructor

A copy constructor allows us to 'clone' objects. It allows us to write:

```
// for some existent circular_buffer<int>
// called cbuf;
circular_buffer<int> copy_of_cbuf1(cbuf);
circular_buffer<int> copy_of_cbuf2 = buf;
```

Both of these copies are created using the copy constructor.

Exercise #2
What signature does the copy constructor have? Will the default version suffice for `circular_buffer`? If not, write an appropriate copy constructor.

[1] Actually, modulo any tempate constructors.

Copy Deadline

All articles intended for publication in *Overload 52* should be submitted to the editor by November 1st, and for *Overload 53* by January 1st 2003.

We're storing data in a dynamically created array, so the default copy constructor is not appropriate. Consider what would happen if we wrote:

```
{
  circular_buffer<int> cbuf1(100);
  circular_buffer<int> cbuf2(cbuf1);
}
```

The default copy constructor will copy the array pointer, `array_`. Now both circular buffer objects use the same array, which is plainly nonsensical. Things get worse, though. At the end of the block scope `cbuf2` will first be destroyed. The destructor calls `delete [] array_`. It is now an ex-array. Next the `cbuf1` destructor is called. It will attempt to delete the same array which is now pushing up the daisies, a plainly wrong operation; and one that is potentially disastrous.

Our replacement copy constructor can look something like the following:

```
circular_buffer(
            const circular_buffer &other)
  : array_(
      new value_type[other.array_size_]),
    array_size_(other.array_size_),
    head_(other.head_),
    tail_(other.tail_),
    contents_size_(other.contents_size_)
{
  std::copy(other.begin(),
            other.end(), array_);
}
```

Did you hand-roll your own loop for the copy operation? It pays to use standard library facilities where you can. This implementation works most of the time, but there are some lurking problems that we'll come back to later.

Constructor 2: Iterator-based template constructor

The `vector` class has a nice constructor that takes two forward iterators defining a range, as the initial contents of the container. Although not strictly required, we can have some of that goodness, too.

Exercise #3
Implement a constructor with the following signature:

```
template <class InputIterator>
circular_buffer(InputIterator from,
                InputIterator to);
```

Here we'll have to consider what initial size our buffer should take. We could make this value an extra parameter, but that makes the interface less flexible and more lumpy. We could require that the supplied iterators are *random access iterators* and make some decision based on the value of `to-from`. Again, this requirement makes the code less flexible. Or we can try some other scheme.

It is possible to implement this constructor if we provide another function, `reserve` that ensures we have a specified capacity. The `vector` class has this capability. Whilst it makes a bit more sense as a member function of `vector` (since by definition `vectors` can grow), we can make good use of it here.

```
template <class InputIterator>
circular_buffer(InputIterator from,
                InputIterator to)
    : array_(new value_type[100]),
      array_size_(100),
      head_(0),
      tail_(0),
      contents_size_(0) {
  while (from != to) {
    if (contents_size_ == array_size_) {
      reserve(static_cast<size_type>(
              array_size_ * 1.5)); // (*)
    }
    push_back(*from);
    ++from;
  }
}
```

The initial capacity value has been picked arbitrarily. I'll leave the implementation of `reserve` up to you. The choice of multiplication factor at the point marked `(*)` is interesting – there has been plenty of research into this. For most situations the value of 1.5 is practical, balancing the number of reallocations required with the amount of 'wasted space' in the buffer.

Assignment operator

This differs from the copy constructor considerably, although you might think it is doing the same job. That's because the constructor has to correctly create and initialise an entire object, whilst the assignment operator has to deal with an already created object; in our case memory has already been allocated.

Exercise #4
What signature does an assignment operator have? Write the assignment operator for `circular_buffer`.

How did you do it? Be careful about the capacity of the buffer parameter passed and the capacity of the buffer you're assigning to. There are two ways to implement this function. You'll probably have written something like the following:

```
circular_buffer &operator=(const
circular_buffer &other) {
  clear();
  // Ensure we have enough space
  if (other.contents_size_
                      > array_size_) {
    reserve(other.array_size_ );
  }
  // Now copy the contents across
  std::copy(other.begin(),
            other.end(), array_);
  head_          = 0;
  tail_          = other.size();
  contents_size_ = other.size();
}
```

We'll see the 'other way' later. This little function has the same lurking problem as the copy constructor. Can you see what it is?

Use an allocator

The STL containers all take an additional template parameter, the *allocator*. This class type abstracts the real concerns of allocating, using, and releasing memory. The default library `std::allocator` is implemented in terms of `new` and `delete`, however other platforms may provide different views of memory, be it pooled memory, garbage collected memory, or whatever. By using an abstract allocator interface we can avoid worrying about this sort of implementation detail and also gain the benefits of these kinds of facilities for free. Even if it weren't required, it would be worth it.

So how do we need to modify `circular_buffer`? First we add another template parameter with a default value. It goes at the end of the template parameter list:

```
typename Alloc = std::allocator<T>
```

This requires us to include the `<memory>` header file, which defines this default `std::allocator` template class. Now we modify our list of container class `typedef`s, thus:

```
typedef Alloc allocator_type;
typedef typename Alloc::value_type
              value_type;
typedef typename Alloc::pointer
              pointer;
typedef typename Alloc::const_pointer
              const_pointer;
typedef typename Alloc::reference
              reference;
typedef typename Alloc::const_reference
              const_reference;
typedef typename Alloc::size_type
              size_type;
typedef typename Alloc::difference_type
              difference_type;
```

Note we've added the `allocator_type` and we also modified these other definitions, since the allocator now provides us with all the correct definitions. Next, to work like any other STL container we add a data member `alloc_` of the allocator type and, for consistency, we also add a new member function, thus:

```
allocator_type get_allocator() const {
  return alloc_;
}
```

That's the basic stuff out of the way. Next we need to make the constructors and destructor call the allocator functions for their memory management, rather than `new` and `delete`. Given that the allocator class has the following functions, try exercise #5.

```
template <class T>
class std::allocator {
  public:
    typedef T      *pointer;
    typedef size_t  size_type;
    pointer allocate(size_type n,
                     /*hint parameter*/);
    void    deallocate(pointer p,
                       size_type n)
    /* ... */
};
```

Note that the 'hint' parameter can be ignored for our purposes, it defaults to zero and is usually unused. `allocate` returns space for `n` objects of type `T`. It doesn't initialise them. `deallocate` frees the memory resource, but doesn't destroy the objects.

Exercise #5
Alter the constructor, destructor, and `reserve` to use the `alloc_` member.

That's not too onerous really. However, we're beginning to see a hint of the problem I alluded to in exercise #1 of the first article, and this is the solution. The allocator does not *initialise* any objects in the memory pool in the way that an array would. This is crucial. Why? Well, it doesn't matter too much for a container of `int`s. But think about a complicated class with a slow constructor. Creating an array of these objects is therefore a *very* slow operation; an array initialisation will call the default constructor to create an object at each array position. It's especially wasteful when you consider that you'll ignore the default construction, and use assignment to write the first value you care about to each array element anyway.

That's an awful lot of wasted array effort. It also introduces an unnecessary dependency on the `value_type`: it requires that there must be a default constructor in order to be able to allocate the array. Now using this 'allocator' approach we are no longer bound by this constraint, on top of not wasting effort constructing useless objects. Bonus!

However, we will need to construct and destruct each object by hand instead. This is the price we pay. Given that there are the `std::allocator` member functions below, answer exercise #6:

```
void construct(pointer p, const T &val)
   { new (p) T(val); }
void destroy(pointer p)
   { p->~T(); }
```

Note the use of *placement new* by the `construct` function.

Exercise #6
What further `circular_buffer` member functions need to be changed to use `construct` and `destroy`?

`push_back` is the only function that needs to create new data elements[2]. Rather than using assignment in this function, we call `construct`. However, we don't need to do this when the buffer is full and we're throwing away the data at the end of the buffer by reusing the last buffer element. In this case we only have to assign to the element as before. `pop_front` removes elements from the container, as do `clear` and the destructor (are there any others in your implementation?). These therefore need to use `destroy`. Note that it will no longer be valid to use the `std::copy` algorithm as we did before – it won't take care of the object construction for us.

As a final allocator modification, we must now change `max_size`. The allocator class provides this function itself. This is logical enough since it knows how many objects could potentially be allocated using its memory model. We therefore make our `max_size` call the allocator's.

2 That is, if you implement the new constructors and assignment operator in terms of push_back, which is a very reasonable design.

Whistle-stop Tour of Exception Safety

This is a subtle subject that has to be approached very carefully. We don't have the time or space to do so in this article. So here's the nuts and bolts.

To be maximally useful our container classes need to be "exception safe." No one's going to argue there. But what does *exception safe* actually mean? This has been a hot topic in the C++ community for some time, and only really recently has it been understood to a reasonable depth.

Exception safe code works correctly no matter what exceptions come its way, for some definition of 'correctly' (we'll define this below). There is an additional constraint to consider when writing code like ours: **exception neutral** code propagates *all* exceptions up to its caller; it doesn't assume the meaning of any thrown exception and won't consume it. This is important for our 'generic' container – the contained objects may generate all sorts of exceptions that we don't understand. The container user will (well, should) understand them and it's therefore up to *them* to deal with these errors, not us. That's what exceptions are good for, after all.

Now there are several different useful levels of exception 'safety'. They are described in terms of guarantees to the calling code. These guarantees are:

Basic guarantee: If exceptions do occur in a function (resulting from an operation we perform, or a call on one of our contained objects) we will not leak resources. The container state will be *consistent* (i.e. it can still be used correctly) but we'll not necessarily leave in a known state. For example, say you have a container member function that adds 10 items to a container, and an exception propagates through the function. We will guarantee the container is still usable, but maybe no objects were inserted, maybe all ten were, or perhaps every other object was added. Container iterators may have been invalidated.

Strong guarantee: This is far more strict than the basic guarantee. Here we ensure that if an exception propagates through our member functions the program state will remain completely unchanged. The object hasn't been altered, no global variables changed, nothing. All container iterators will therefore still be valid. In our example above, we can assert that no objects will have been inserted into the container at all.

Nothrow guarantee: The final guarantee is the most restrictive. We guarantee that an operation can *never* ever throw an exception. If we are 'exception neutral' then this implies that the function cannot call any function that itself might throw.

Which of the guarantees you provide in your code is entirely up to you. The more restrictive the guarantee, the more widely (re)useable the code is. It turns out that in order to implement the strong guarantee in your member functions you will generally require a number of functions that provide the nothrow guarantee (for example, see the use of **`swap`** in this article). Most notably every destructor you write should *always* honour the nothrow guarantee. Always. Otherwise all exception handling bets are off.

Exception safety

If our class is not exception safe, it's of no real value in the Real World. See the sidebar for a brief description of what's involved in writing exception safe code. We don't have the space here to really describe the subject, but we'll take a look at what we need to do to make our code bullet proof. As it stands it is really not at all exception safe. In fact it's downright exception dangerous.

In truth, it's a bit late in the development to be considering exception safety. However, we really needed to have got this far to have illustrations of the issues involved.

We'd like to provide at least the basic exception safety guarantee in our member functions, and ideally we want to aim to provide the strong guarantee in every method. Let's quickly look at some of the reasons why our existing container is dangerous. We'll start at the beginning: some constructors, for example, are a cause of resource leaks.

Look at the copy constructor. The first thing it does is allocate memory in the initialiser list. If this fails then a `std::bad_alloc` exception is thrown and nothing leaks. This is fine behaviour, and a good reason to put dynamically allocated memory near the head of your list of variables (remember that data members are initialised in the order of the class definition, not the order you list them in the constructor's initialiser list). None of the other assignments might throw, so we're safe in the rest of this initialiser list.

However, on entering the constructor body, we loop around copying elements. Any of the `value_type` object constructors might throw an exception. Say we get half way through the copy loop and a constructor fails. The failure exception propagates through our constructor (so at least we're exception neutral here), and we leak the allocated memory. We've also created a number of contained objects that we didn't destroy – heaven help us if there are dangerous side effects from this behaviour.

Tightening up cases like this is not actually going to be impossible to do, since I've introduced facilities in a way that allows exception safety (for example, separating `front` and `pop_front`). We'll just have to think carefully about each member function. One of the golden rules when writing exception safe code is that each function should have at most one side effect, otherwise an exception safe implementation is complicated. You can see then that writing exception safe code *does* have an impact on your public API design, you have to design it in from the start.

When people think about exception safety they imagine code strewn with `try`/`catch` blocks. In fact this is far from the truth. Our exception safe code will be practically free of these. The basic technique we follow is to perform all potentially dangerous activities 'off to one side' in a way that means they'll get tidied up neatly if anything fails. Once these dangerous activities are complete we can *then* make the changes live using operations that are known not to throw.

In order to do this we are going to need some operations that are known not to throw. We asserted in the sidebar that our destructor can't throw. Now enter `swap`.

Exercise #7
Implement a member function `swap(const circular_buffer &)` that is guaranteed not to throw.

It's not too nasty. Note that we don't bother to swap the allocator objects. They should be identical if the `circular_buffer` types are the same.

```
void swap(circular_buffer &other)
                          /* nothrow */
{
  std::swap(array_,         other.array_);
  std::swap(array_size_,
                        other.array_size_);
  std::swap(head_,           other.head_);
  std::swap(tail_,           other.tail_);
  std::swap(contents_size_,
                    other.contents_size_);
}
```

Although we're providing the nothrow guarantee note that we don't put an empty exception specification at the end of the function signature. We should try to avoid writing these when we can, they can add unnecessary overhead to the running code; they are more likely to make the compiler generate much worse code than anything better. We know the function won't throw; we don't need the compiler to check this for us too.

Exercise #8
Using this *perform work off to the side then make it live* idiom, go over each member function and modify it to become exception safe (as strictly as possible). Do you need any `try/catch` blocks at all? Look first at the simplest constructor, then the `push_back` and `pop_front` methods.

OK, that's a big task. Let's start with the basic constructor. It's actually already strongly exception safe. The only operation that might throw is the memory allocation. This is fine, the `std::bad_alloc` exception will propagate up and we won't leak at all. How must you alter the other constructors? I won't show you here, but note that you *will* actually need `try/catch` blocks in this case. Next, `push_back`:

```
void push_back(const value_type &item) {
  size_type next = next_tail();
                   // no state change yet
  if (contents_size_ == array_size) {
    array_[next] = item; // (*)
    increment_head();
  }
  else {
    alloc_.construct(array_ + next,
                     item); // (*)
  }
  increment_tail();
}

private:
  size_type next_tail() {
    return (tail_+1 == array_size_)
                          ? 0 : tail_+1;
  }
```

How is this different? Our interest is at the points marked `(*)`. We move the constructions/assignments to positions above any other state manipulation. If an exception is thrown we haven't modified any other state. This is not quite a strong exception safety guarantee, though: our container is now as exception safe as the `value_type`'s constructor/assignment operators. This is best we can do.

As a final example, we'll consider the assignment operator. We use a neat idiom here to ensure that we are strongly exception safe:

```
circular_buffer &operator=(const
self_type &other) {
  circular_buffer tmp(other);
  swap(tmp);
  return *this;
}
```

Can you see how neat this is? We do all the work 'off to one side' by creating a temporary object that looks like `other`. If that fails (based on the fact the copy constructor is exception safe and won't leak) we will propagate the exception, but not have altered our own state and not leaked, so we are strongly exception safe. If it succeeds we 'make permanent' the change using two operations that are known not to fail: `swap`, and the destructor. We swap our current state with that of `tmp` so we now look as if we've been 'assigned to'. When `tmp` goes out of scope it is destroyed, taking with it our old state.

A lot of the other functions are adjusted in similar manners. We don't have space to describe them all here. See my reference code (in the **Getting the code** section below) for further examples.

The long and short of exception safety is: you can't reasonably bolt it on after writing the code. You have to factor it into the class' design from the start. Well written exception safe code shouldn't need tonnes of `try/catch` blocks. If it's not simple and clear to read then something's wrong. Generally you'll find that good 'exception safe' code is not just designed to be safe in the presence of potential exceptions at the expense of clarity and program efficiency – it will also be genuinely well designed and thought out code.

Miscellany

There are still a few loose ends we need to tie up, and then we're done.

Insertion behaviour policy

Currently `push_back` will always accept new data, even if the circular buffer is full. It does this by throwing away the oldest data members. Perhaps our users don't want this behaviour.

Exercise #9
Make this policy decision a template parameter. How much of the class needs to change?

It turns out that this is a simple change, and costs very little. Just add a new boolean template parameter (I called it the long winded `always_accept_data_when_full` and gave it a default value of `true`). You want to put this parameter before the allocator definition to follow convention.

The only other change needed is minimal: You need to make the 'throw data away' operation in `push_back` conditional on the value of `always_accept_data_when_full`. If it's `false` we do nothing. Since this value is known at compile time, the `if` statement will be optimised away.

Perhaps you'd like the function to throw an exception if the buffer is full. That's another policy decision, and I'll leave it up to you to work out how to do it.

This should lead us to consider how you would use a circular buffer in the Real World. You will know the rate at which a producer adds data to the buffer, and the rate and frequency at which the consumer will work. Given this information you should be able to calculate a reasonable size for the circular buffer. Obviously throwing away data is really a last-ditch operation, and your use of the buffer should prevent it if at all possible, so use the buffer carefully!

A random access iterator

We wrote a bidirectional iterator in the previous article. It's not much work at all to make it a random access iterator, we just need to add the following operations: `+ - += -=`, plus the following comparisons: `< > >= <=`. Some of these we'd already started on.

Exercise #10
Convert the `circular_buffer_iterator` into a random access iterator.

Don't forget to change the `iterator_tag`.

Comparison

It would be useful to be able to compare two `circular_buffer` classes.

Exercise #11
Implement `operator==` and `operator!=`. They only need to compare two `circular_buffer`s of the same type. Should they be member functions or free functions?

There is no requirement for these functions to be members of the container class, so by conventional C++ wisdom they shouldn't be. If your implementation required access to the private members of the class then you introduced unnecessary coupling.

There are two ways to implement the functions: the easy way and the hard way. If you hand coded a comparison loop then you did it the hard way. You can avoid a lot of the work by using `std::equal`.

```
template <typename A,
          bool B,
          typename C>
bool operator==(
     const circular_buffer<A, B, C> &a,
     const circular_buffer<A, B, C> &b) {
  return a.size() == b.size()
         && std::equal(a.begin(),
                       a.end(),
                       b.begin());
}
```

You can figure out `operator!=` from that. There's one more operator that the `vector` provides, which we can also provide:

Exercise #12
Implement `operator<` for the `circular_buffer` class.

Again, there's an easy way and a hard way. This time the easy way uses `std::lexicographical_compare`.

```
template <typename A,
          bool B,
          typename C>
bool operator<(
     const circular_buffer<A, B, C> &a,
     const circular_buffer<A, B, C> &b) {
  return std::lexicographical_compare(
                        a.begin(),
                        a.end(),
                        b.begin(),
                        b.end());
}
```

Getting the code

Whilst I know you've been slavishly following the exercises I know that you'll want to see my reference implementation to see how close you came to it. Or perhaps you just want an STL style circular buffer class and can't be bothered to write your own.

My **circular_buffer** class library is available from `http://cthree.org/pete/cbuf.html`.

Conclusion

We've now created an entire template container class. It follows the STL style carefully. Not only does this mean that we now know how to write STL-like containers, it also means that anyone using this `circular_buffer` class can pick it up with a minimum of hassle. It behaves in a known way, can be accessed as most other STL containers, and it is immediately compatible with existing STL algorithms. Perhaps you've also gained a respect for the work that has been put into the standard C++ libraries.

Hopefully you've enjoyed these articles, and by stepping through the exercises you have now picked up some useful techniques that can be applied to other code you write. Let me know if it's been useful.

Exercise #13
Take a well deserved break.

Pete Goodliffe
pete@cthree.org

References

[1] Pete Goodliffe, "STL-style circular buffers by example," *Overload 50*, August 2002, ISSN: 1354-3172.

Applied Reading – Taming Shared Memory

by Josh Walker

I consider myself something of a connoisseur of technical books. I pride myself on my small but expanding collection. Of course, an important part of maintaining a useful book collection is knowing what information your books contain, so you can find it when you need it. Though I often forget the details of a solution I read about, I remember where I saw it, and what the original problem was. My purpose in writing this piece is two-fold: to show you how I solved a particular problem, and to point you at the books I used along the way.

When I was given this task, some of the solution was already defined. There was an existing architecture with a client and server sending messages over TCP/IP. A requirement had arisen for an additional client interface for CGI processes. Since each CGI process is short-lived, creating a new socket connection in each process would be too expensive. So the idea was to route messages through a daemon that maintains a persistent connection to the server. Because the daemon can be located on the same Unix box as the CGI processes, the two can use an Interprocess Communication (IPC) mechanism with less overhead than sockets. This is where I started working from.

The first decision was to choose the exact IPC mechanism to use. Unix has several, including pipes, fifos, sockets, message queues, and shared memory. I ruled out pipes because they require a parent-child relationship between processes, which I did not have. Fifos, like pipes, are best suited for one-to-one communication, whereas I needed many-to-one. I had heard that message queues were unnecessarily complex and grandiose, so I was hesitant to use them. Shared memory, on the other hand, is basically a free-form medium, and I was pretty confident that it would be fast enough, so I started the implementation based on shared memory.[1] I'll walk you through the basics of my solution below.

Shared memory is what the name implies; it is a portion of address space that is shared between processes. Normally each process has its own address space, which is entirely separate from those of other processes. The operating system uses hardware to map a process's virtual address space onto physical memory. This ensures that no process can access memory in another process's address space; so no process can disturb the integrity of another. Shared memory is an exception to this system. When two processes attach the same shared memory segment, they are telling the operating system to relax the normal restrictions by mapping a portion of their address space onto the same memory.

The definitive book on Unix IPC is "UNIX Network Programming" by the late Richard Stevens [1]. Anyone doing heavy Unix should be familiar with it. After consulting this authority, we see that getting access to shared memory is a two-step process. First, a shared memory identifier is obtained, using the `shmget()` system call. The segment is identified by an integer key, and has a set of permissions controlling which processes may read, write, or execute it, much like a Unix file. The `IPC_CREAT` and `IPC_EXCL` flags control creation of the segment if it doesn't exist. Once a segment is created, it is available for use by any process with appropriate permissions. After obtaining a segment identifier, a process must attach the segment into its address space using the `shmat()` call. The return from `shmat()` is the base address where the segment was attached, i.e., the address of the first byte located in the segment. The memory is now available for use just like any other memory in the process's address space. Here's an example:

```
#include <sys/shm.h>
int main() {
  int id = shmget(0x1000 /*key*/,
         sizeof(int) /*size*/,
         0600 | IPC_CREAT /*creation flags*/);
  void* base = shmat(id /*identifier*/,
         0 /*base address*/, 0 /*flags*/);
  *reinterpret_cast<int*>(base) = 42;
}
```

After the call to `shmat()` above, the process's address space might look as shown in Figure 1 below.

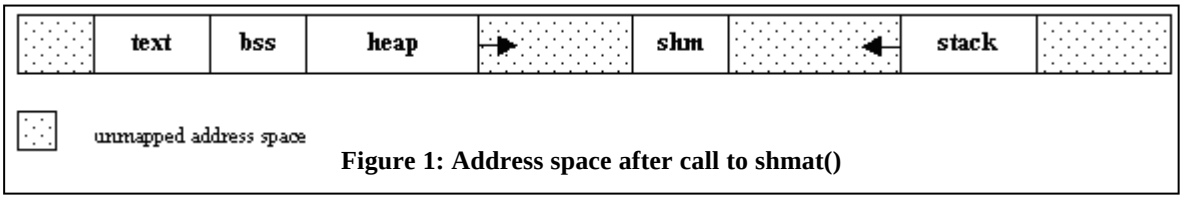


Figure 1: Address space after call to shmat()

The low-level C interface of `shmget()`[2] and `shmat()`, with their untyped integer handles and bit flags are just what you get with Unix system calls. Now I have great respect for Unix and C, but sometimes I want an interface with a little more structure...err...I mean class. So we'll wrap up some of the error-prone details in a C++ interface. Our first step in building a set of C++ classes to handle shared memory will be to encapsulate a shared memory segment:

```
class shm_segment {
  public:
    enum CreationMode { NoCreate, CreateOk,
                 MustCreate, CreateOwned };
    shm_segment(int key, CreationMode mode,
                int size=0, int perm=0600);
    ~shm_segment();
    void* get_base() const;
    unsigned long get_size() const;
    void destroy();
  private:
    int key_;
    int id_;
    void* base_;
    unsigned long size_;
    bool owner_;
};
class shm_create_exception
                  : public std::exception {
  public:
    const char* what() const throw() {
      return "error creating
                     shared memory segment";
    }
};
```

[1] Actually, I also had some prior experience using shared memory, so this may just be a case of everything looking like a nail when all you have is a hammer.

[2] Some people insist that all Unix system calls can be pronounced as they are written, no matter how vowel-deficient they may be.

This class simply wraps creation and destruction of a shared memory segment. The `CreationMode` argument determines whether the segment will be created if it doesn't exist. This corresponds to the meaningful combinations of the `IPC_CREAT` and `IPC_EXCL` flags. The `CreateOwned` option sets our `owner_` flag, which we'll use to determine if the shared memory segment should be removed when the `shm_segment` object is destroyed. It might be possible to store a reference count and remove the segment only after all processes are finished with it, but it's probably best to have a clear ownership policy instead. The implementation of `shm_segment` is straight forward.

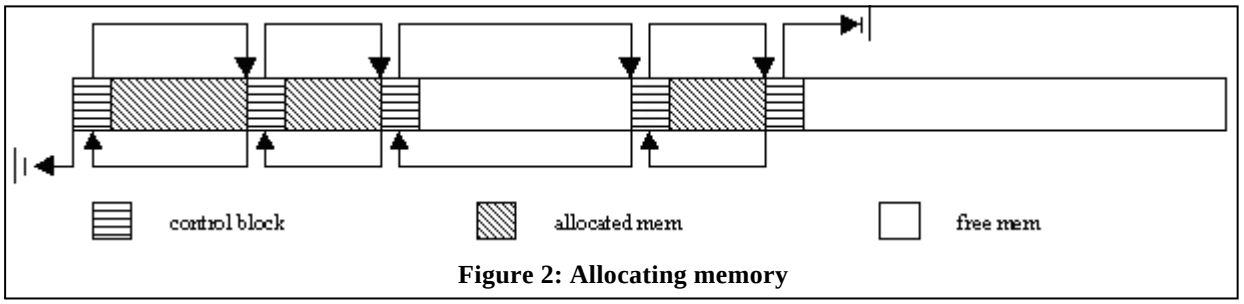


Figure 2: Allocating memory

The `shm_segment` class makes it a little easier to get a segment, but once it is created, there is still no internal structure to the segment; it's just a raw chunk of memory. In order to add more structure, we will need a way to store pointers inside the segment. But this poses a problem. Since each process may attach the segment at a different base address, a pointer into the segment created by one process may not be valid in another process; it may point somewhere else entirely. To solve this problem, we will store offsets from the base instead of raw pointers. We'll create a simple pointer class for this purpose:

```
template<typename T>
class shm_ptr {
  public:
    shm_ptr();
    shm_ptr(T* rawptr, const shm_segment& seg);
    // compiler generated copy constructor and
    // assignment are ok
    T* get(const shm_segment& seg) const;
    void reset(T* rawptr,
               const shm_segment& seg);
  private:
    long offset_;
};

class shm_ptr_out_of_bounds
                : public std::exception {
  public:
    const char* what() const throw() {
      return "shm_ptr cannot be created
             with address outside of segment";
    }
};
```

Notice that the only data member is an integer representing the offset in bytes from the base address. This offset will have the same meaning for all the processes using our shared memory segment, so it can safely be stored in shared memory. Notice also that our interface requires the user to pass the `shm_segment` when accessing the pointer. This is somewhat cumbersome, but it yields a general design where each process may attach multiple shared memory segments. Alternatively, `shm_segment` could be a singleton, which would make it easier for `shm_ptr` to provide the usual indirection operators, but I found this unnecessary.

Our `shm_ptr` gives us a safe way to store a pointer in shared memory, but we don't yet have a way to create that pointer. It would be possible to choose arbitrary locations for data and manually place it there with some pointer arithmetic, e.g.,

```
shm_ptr<char> buffer(reinterpret_cast<char*>(
        mysegment.getBase()), mysegment);
shm_ptr<char> buffer2(reinterpret_cast<char*>(
        mysegment.getBase())+1000, mysegment);
```

but I think we can all agree that this is too ugly and inflexible. What we really need is a way to allocate objects in shared memory with the ease of ordinary `operators new` and `delete`. To accomplish this, we'll write a shared memory allocator class:

```
struct shm_allocator_header;

class shm_allocator {
  public:
    explicit shm_allocator(const
                           shm_segment& seg);
    void* alloc(size_t nbytes);
    void free(void* addr);
    void* get_root_object() const;
    void set_root_object(void* obj);
    const shm_segment& get_segment() const;
  private:
    const shm_segment& seg_;
    shm_allocator_header* header_;
};
```

The interface provides `alloc()` and `free()` methods, as well as methods to get and set a "root object." The root object gives us a fixed access point for processes to share objects. There is also a `get_segment()` method to find out what shared memory segment an allocator is using.

When I was writing my first implementation, and I knew I needed an allocator, I remembered seeing one somewhere, and after flipping through a few pages, I found Chapter 4 in "Modern C++ Design," which describes small object allocations [2].[3] An example there gave me enough information to sketch out my initial implementation. The basic idea of an allocator is to store a control block in the memory preceding the address returned to the user. This example used:

```
struct MemControlBlock {
  bool available_;
  MemControlBlock* prev;
  MemControlBlock* next;
}
```

Starting from a root block, you just walk down the list of blocks looking for one that's free and big enough for the requested allocation. The layout of memory looks as shown in Figure 2 above.

3 Unfortunately, the small object allocator in Loki (the C++ library developed in [2]) appears to have a few unresolved implementation issues. If you choose to use it, you may learn more than you ever wanted about allocators. However, this does not detract from the value of "Modern C++ Design" itself.

Incidentally, Alexandrescu mentions the many tradeoffs associated with memory allocators, and points us to Knuth's masterpiece for more details [3]. For some variety, I decided to base our implementation on a section in "The C Programming Language" on implementing `malloc()` and `free()`; it is a little simpler to explain, and it gives me an opportunity to recognize a classic (and still relevant) book [4]. The idea for the allocator is the same, but the details have changed. Each block now stores its size and a pointer to the next block.

```
struct shm_allocator_block {
  shm_ptr<shm_allocator_block> next;
  size_t size;
};
```

A header block stores a pointer to the beginning of a linked list of free blocks:

```
struct shm_allocator_header {
  shm_ptr<void> rootobj;
  pthread_mutex_t lock;
  shm_ptr<shm_allocator_block> freelist;
};
```

The free list is a singly-linked circular list of blocks not currently in use. The list is linked in order of ascending addresses to make it easy to combine adjacent free blocks. Here is the implementation of `alloc()`.

```
void* shm_allocator::alloc(size_t nbytes) {
  scoped_lock guard(&header_->lock);
  size_t nunits = (nbytes +
        sizeof(shm_allocator_block) - 1)
        / sizeof(shm_allocator_block) + 1;
  shm_allocator_block* prev =
                header_->freelist.get(seg_);
  shm_allocator_block* block =
                prev->next.get(seg_);
  do {
    if (block->size >= nunits) {
      if (block->size == nunits)
        prev->next = block->next;
      else {
        block->size -= nunits;
        block += block->size;
        block->size = nunits;
      }
      header_->freelist.reset(prev, seg_);
      return (void*)(block+1);
    }
    prev = block;
    block = block->next.get(seg_);
  } while(block !=
            header_->freelist.get(seg_));

  return NULL;
}
```

The code above walks the free list looking for a block big enough to hold the requested number of bytes. If the block it finds is larger than needed, it splits it in two. Otherwise it returns it as is. If no block is found, it returns null. Size calculations are made in units equal to the size of one `shm_allocator_block`. This makes the pointer arithmetic a little simpler, and ensures that all memory allocated will have the same alignment as `shm_allocator_block`.

The implementation of `free()` is shown below:

```
void shm_allocator::free(void* addr) {
  scoped_lock guard(&header_->lock);
  shm_allocator_block* block =
     static_cast<shm_allocator_block*>
                                  (addr) - 1;
  shm_allocator_block* pos =
                header_->freelist.get(seg_);
  while (block > pos
         && block < pos->next.get(seg_)) {
    if (pos >= pos->next.get(seg_)
        && (block > pos
            || block < pos->next.get(seg_)))
      break;
    pos = pos->next.get(seg_);
  }
  //try to combine with upper block
  if (block + block->size ==
                    pos->next.get(seg_)) {
    block->size += pos->next.get(seg_)->size;
    block->next = pos->next.get(seg_)->next;
  }
  else
    block->next = pos->next;
  //try to combine with lower block
  if (pos + pos->size == block) {
    pos->size += block->size;
    pos->next = block->next;
  }
  else
    pos->next.reset(block, seg_);

  header_->freelist.reset(pos, seg_);
}
```

The block is inserted into its correct spot in the free list (remember the list is sorted by address). Then we check for adjacent free blocks. If any are found, we combine them with the current block by just forgetting that the upper block exists and increasing the size of the lower block.

So, after a few allocations and deallocations, memory might look as shown in Figure 3 below.

We'll also overload `operators new` and `delete` to work with `shm_allocator`. The basic details for doing this, and some pitfalls can be found in Item 36 of "Exceptional C++" [5].

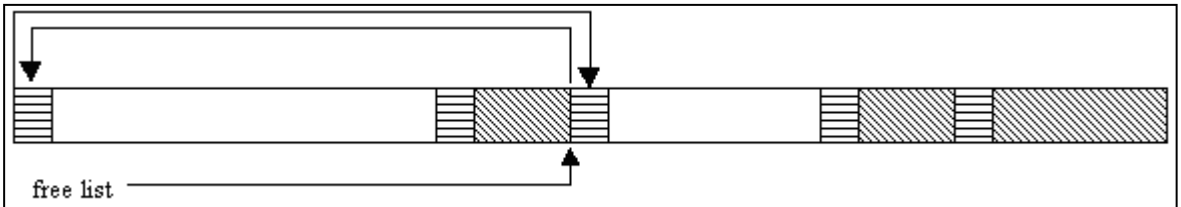


Figure 3: Memory after allocations and deallocations

```
void* operator new(size_t s,
                   shm_allocator& a) {
  return a.alloc(s);
}

void operator delete(void* p,
                     shm_allocator& a) {
  a.free(p);
}
```

We can overload `new[]` and `delete[]` similarly. The proper syntax for calling overloaded `operator new` or `delete` may be mysterious to you. Assuming we have already defined a class `Message`, we can create a new `Message` object in shared memory using:

```
Message* m = new (a) Message;
   // a is a shm_allocator
```

The above is a new expression which calls our overloaded `operator new`, followed by the `Message` constructor. Destroying the `Message` object is not quite as simple:

```
m->~Message();
operator delete(m, a);
```

We must first make an explicit call to the destructor and then deallocate the memory. Two steps are required because there is no way to form a `delete` expression which calls an overloaded `operator delete`. You'll notice that the lines above are equivalent to these:

```
m->~Message();
a.free(m);
```

So, one might be tempted to overload only `operator new`; however, it is very important that we overload `operator delete` as well, because if the `Message` constructor throws an exception our overloaded `operator delete` will be automatically called. If there was no `operator delete` to match the `operator new` used to allocate the memory, bad things would happen.

We now have a general purpose allocator that we can use to help us build arbitrary structures in shared memory. To demonstrate its use, we'll write our final class, a producer-consumer queue in shared memory:

```
template<typename T> struct shm_queue_header;

template<typename T>
class shm_queue {
  public:
    shm_queue(shm_allocator& a,
              size_t maxsize);
    ~shm_queue();
    void push(shm_ptr<T> obj);
    shm_ptr<T> pop();
    size_t size() const;
  private:
    const size_t MaxQueueSize;
    shm_queue_header<T>* header_;
    shm_allocator& a_;
    const shm_segment& seg_;
    // disallowed (not implemented)
    shm_queue(const shm_queue& copy);
    shm_queue& operator=(const
                         shm_queue& rhs);
};
```

This is a basic queue, with simple `push()` and `pop()` methods.[4] The queue is created with a `shm_allocator`, which it then uses to allocate nodes. A maximum size is also specified to ensure that the producer(s) don't get too far ahead of the consumer(s). The implementation is a singly-linked list.

```
template<typename T>
struct shm_queue_node {
  shm_ptr<T> data;
  shm_ptr<shm_queue_node> next;
};

template<typename T>
struct shm_queue_header {
  size_t size;
  shm_ptr<shm_queue_node<T> > head;
  shm_ptr<shm_queue_node<T> > tail;
  pthread_mutex_t lock;
  pthread_cond_t ready_for_push;
  pthread_cond_t ready_for_pop;
};
```

Now, those `pthread_xxx` members give me a chance to make my final recommendation: "Programming with POSIX Threads" by David Butenhof [6]. This is an excellent introduction to threaded programming, and specifically, Pthreads. "But wait a minute!" you say, "When did we introduce threads into the picture?" Well, the primitives needed to synchronize two threads (which inherently share memory) are very similar to those needed to synchronize processes using Unix shared memory. They are so similar, in fact, that Pthreads provides a way to use mutexes and condition variables in this scenario. If your system defines the `_POSIX_THREAD_PROCESS_SHARED` macro, you can place Pthread objects in shared memory if you set the `pthread_process_shared` attribute. This works on the Solaris system I tested on, but is not supported on my Linux system.[5]

For a little more demonstration of Pthreads, here's the implementation of our `scoped_lock` class:

```
#include <pthread.h>
class scoped_lock {
  public:
    scoped_lock(pthread_mutex_t* mutex) {
      mutex_ = mutex;
      pthread_mutex_lock(mutex_);
    }
    ~scoped_lock() {
      pthread_mutex_unlock(mutex_);
    }
    void wait(pthread_cond_t* cond) {
      pthread_cond_wait(cond, mutex_);
    }
```

4 Item 10 in [5] warns against returning objects in pop() methods. However, in this case, it is not necessary to provide separate top() and pop() methods; since we only store pointers the pop() operation cannot throw an exception while copying the return value.

5 An alternative synchronization method is to use Unix semaphores instead of mutexes and condition variables (semaphores can simulate either). I presented Pthreads here because it is cleaner both conceptually and for implementation.

```
  private:
    pthread_mutex_t* mutex_;
    scoped_lock(const scoped_lock& copy);
    scoped_lock& operator=(const
                           scoped_lock& rhs);
};
```

And, using `scoped_lock`, the implementation of `shm_queue::push()`:

```
template<typename T>
void shm_queue<T>::push(shm_ptr<T> obj) {
  scoped_lock guard(&header_->lock);
  while (header_->size
           > MaxQueueSize) {
    guard.wait(&header_->ready_for_push);
  }

  shm_queue_node<T>* node =
                  new (a_) shm_queue_node<T>;
  node->data = obj;
  node->next.reset(0, seg_);
  if (header_->head.get(seg_) != 0) {
    header_->head.get(seg_)->next.reset(
                                  node, seg_);
  }
  else {
    header_->tail.reset(node, seg_);
  }
  header_->head.reset(node, seg_);
  ++header_->size;
  pthread_cond_signal(
                     &header_->ready_for_pop);
}
```

Finally, we'll write a simple producer and consumer which use `shm_queue`:

```
struct Message {
  int sequence_num;
  shm_ptr<char> message;
};

// main routine for producer
int main(int argc, char*argv[]) {
  shm_segment seg(30000,
                  shm_segment::CreateOwned,
                  1000000);
  shm_allocator a(seg);
  shm_queue<Message> q(a);
  int message_count = 0;
  while (true) {
    std::cout << "enter a message: "
              << std::flush;
    std::string line;
    std::getline(std::cin, line);
    Message* m = new (a) Message;
    m->sequence_num = message_count++;
    m->message.reset(static_cast<char*>(
            a.alloc(line.size()+1)), seg);
    strcpy(m->message.get(seg),
           line.c_str());
    q.push(shm_ptr<Message>(m, seg));
  }
}

// main routine for consumer
int main(int argc, char*argv[]) {
  shm_segment seg(30000,
                  shm_segment::NoCreate);
  shm_allocator a(seg);
  shm_queue<Message> q(a);
  while (true) {
    shm_ptr<Message> p = q.pop();
    Message* m = p.get(seg);
    std::cout << "Message "
              << m->sequence_num
              << " "
              << m->message.get(seg)
              << "\n";
    a.free(m->message.get(seg));
    m->~Message();
    operator delete(m, a);
  }
}
```

There you have it: a way to access shared memory, allocate objects in it, and pass them on a queue. So keep up with your reading, and pay attention to the ACCU book reviews [7]!

Josh Walker
josh.walker@chutneytech.com

Acknowledgements

Special thanks to Satish Kalipatnapu and Thad Frogley for the valuable comments they provided on drafts of this article. Any errors that remain are mine alone.

References

[1] W. Richard Stevens: *UNIX Network Programming, Volume 2: Interprocess Communications*, Prentice Hall, 1998, ISBN 0-130-81081-9.

[2] Andrei Alexandrescu: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2000, ISBN 0-201-70431-5.

[3] Donald E. Knuth: *The Art of Computer Programming*, Addison-Wesley, 1998, ISBN 0-201-48541-9

[4] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*, Prentice Hall, 1988, ISBN 0-131-10362-8.

[5] Herb Sutter: *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, 2000, ISBN 0-201-61562-2.

[6] David R. Butenhof: *Programming with POSIX Threads*, Addison-Wesley, 1997, ISBN 0-201-63392-2.

[7] http://www.accu.org/bookreviews/public/index.htm

Pattern Collaborations: Observer and Composite

by Mark Radford

Patterns are a powerful tool in the software development toolbox, because they provide documentation not only of solutions to problems, but solutions that already have successful track records. Therefore there are at least two concrete ways in which they can help us to be more effective in designing software:

First, we can draw on patterns as a source of documented *experience* when doing our design. Second, seeing known patterns occurring in software once we *have* designed it, leaves us with a good feeling about what we have produced; that is, we can have increased confidence that our design is sensible, because we know we have used approaches that have already been proven to work well!

I'm sure it will not be controversial to say that the best known book about patterns within the development community in general is the "GoF book" [Gamma+1995], a work presenting a catalogue of twenty-three object oriented design patterns. Patterns are at their most effective when working together in collaboration, and while there is much to recommend this book, it has the drawback of failing to point out many effective collaborations.

There are a number of ways for patterns to collaborate. For example, in their original (Alexandrian) setting each pattern was an element of a *pattern language* [Alexander+1977]. Another way involves patterns working together in *teams*, as Ralph Johnson describes in his article "How Patterns Work in Teams", an article that can be found in [Rising1998].

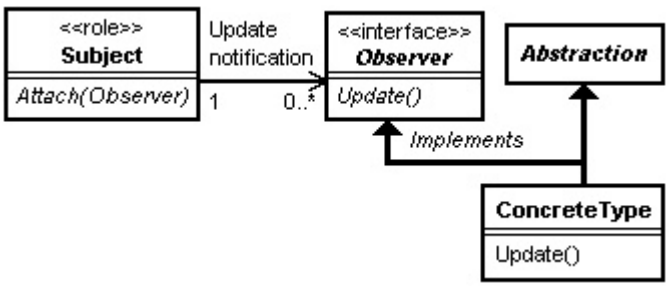
Now, in object oriented software design, two (of the many) problems that arise often are:

- How can an object notify others of changes in its state?
- How, at run time, a client can treat a group of objects uniformly – that is how can the group be made to appear as one object from the client's perspective?

OBSERVER Pattern addresses the first of these, and COMPOSITE Pattern the second – both being documented (and described more fully) in the GoF book. This article is about these two patterns. First, it will recap on the two patterns themselves by presenting a brief summary of each. Second, it will describe how design can potentially benefit from their collaboration.

OBSERVER Pattern

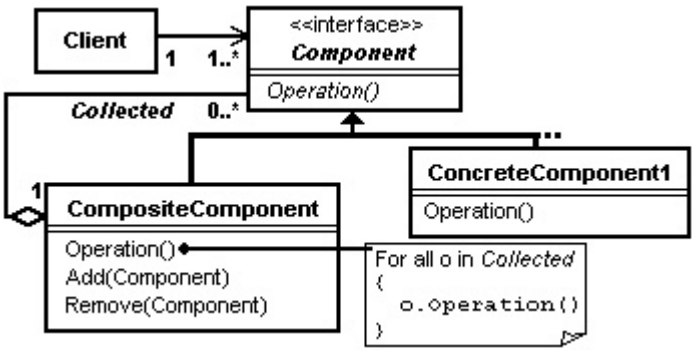
The purpose of the OBSERVER Pattern is to allow updates to the state of an object to be notified to others automatically. The following diagram shows the basic configuration.



SUBJECT is the role name for an object that is to notify others of updates to their state, *OBSERVER* is the role name for objects that are notified of state updates in their *SUBJECT*, and `Update` is the method that serves as notification handler. In order to keep coupling between *SUBJECT*s and *OBSERVER*s to a minimum, the role of *OBSERVER* is crystallised as an interface.

COMPOSITE Pattern

The purpose of the COMPOSITE Pattern is to allow clients to treat objects and compositions of objects uniformly. The following diagram shows the basic configuration.



All objects that are to be treated uniformly support the same design type interface *Component*. An object that holds a collection of *COMPONENT*s – *COMPOSITECOMPONENT* – also supports the *COMPONENT* interface. The implementation of each method in *COMPOSITECOMPONENT* simply forwards method invocations to each object in the collection. Therefore, the client actually deals with a collection of objects, but from the client's perspective it deals with one.

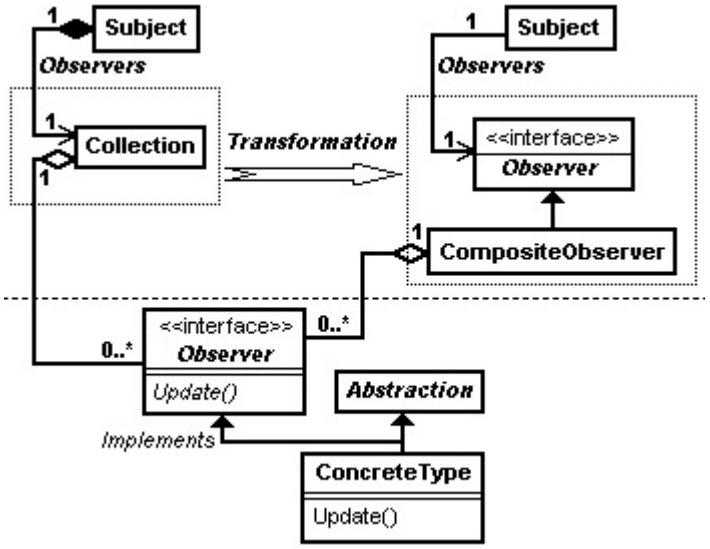
Collaboration

When it comes to implementing the OBSERVER Pattern, several issues arise, but there is one in particular that is of interest here: the *SUBJECT* must bear the weight of the "machinery" needed to notify between zero and many *OBSERVER*s. Actually, it is fair to put it more strongly: in order to allow itself to be observed, the *SUBJECT* must commit the design sin of assuming a dual role, i.e. its design role plus that of notifying *OBSERVER*s of state updates. This issue can therefore be elevated to problem status, and stated as follows: how can the *SUBJECT* be relieved of its extra responsibility and consequent machinery?

Obviously the *SUBJECT* can not be relieved of all such machinery, but the machinery can be simplified: instead of the *SUBJECT* talking to between zero and many *OBSERVER*s, let it talk to only one – a *COMPOSITEOBSERVER*! The diagram on the following page shows the transformation in the configuration.

The *SUBJECT*'s collection of *OBSERVER*s is replaced by a one to one relationship with a single *COMPOSITEOBSERVER*, used via an association with the *OBSERVER* interface. This has indeed afforded the benefit of the *SUBJECT* having a one to one association with a single *OBSERVER*. Therefore, a *COMPOSITEOBSERVER* can now be substituted transparently. However, this transformation has consequences both in favour of it and against it. Consequences in its favour are:

- The *SUBJECT* is simplified by the separation of concerns – putting it another way, although the (unavoidable) notification machinery is still in the system, the configuration has been transformed such that the *SUBJECT* is no longer burdened by it.



- The notification mechanism can be tested independently.
- The *SUBJECT* can send update notifications without worrying about the possibility of there being zero *OBSERVER*s: it's up to the *COMPOSITEOBSERVER* to handle it.

So far so good, but you don't get anything for nothing. The consequences against are:

- The boundary of encapsulation has moved: the mechanism for attaching and detaching *OBSERVER*s will *leak* into the client code.

Finally

The OBSERVER/COMPOSITE collaboration is just one of many examples of collaborations between two or more (object oriented design) patterns. Further, it is just one example of how the OBSERVER Pattern can collaborate with others.

The collaboration solves one problem: that of the *SUBJECT* being burdened by the *OBSERVER* notification mechanism. However it brings with it another problem – that of mechanism leakage. Perhaps collaboration with another design pattern such as FAÇADE [Gamma+1995] could be used to address this resulting problem...(?)

Mark Radford
mark@twonine.co.uk

References

[Alexander+1977] Christopher Alexander, Sara Ishikawa and Murray Silverstein with Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

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

[Rising1998] Editor Linda Rising, *The Patterns Handbook*, Cambridge University Press.

Book Review

Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley, 1994, ISBN 0201633612

reviewed by Ian Bruntlett

This book (aka Gang of Four / GOF) is widely accepted as an Object Oriented Design (OOD) classic. It comes in two parts. Part One introduces patterns and presents case studies (WYSIWYG document editor, maze game). Part Two is a catalog of design patterns split into three purposes - creational, structural, behavioral.

At times it says a lot of common-sense things ("One thing expert designers know not to do is to solve every problem from first principle") but it is overshadowed by flowery language - I can understand it these days but previously it was gobbledygook to me. I think its talk of sending requests to an object when it refers to invoking an object's member function is part of the text's Smalltalk bias. It also refers to "abstract operations" when its talking about virtual functions. On the other hand, I have been told that "The Design Patterns Smalltalk Companion" was written for people frustrated with this book's C++ bias.

So far whenever Design Patterns have been used, I've always seen a particularly impenetrable use of objects and inheritance - it is small comfort but their second principle is "Favor object composition over class inheritance". To me C++ programmers have to wear different hats, one being "syntax geek", another being "design geek". If you don't often indulge in being a design geek then this book will often seem impenetrable and you should ask questions on accu-general for guidance.

Creational patterns are dealt with first in the catalog. Some of the patterns seem deeply rooted in the world of framework development, something that the average developer is unlikely to need. Some of the ideas are intriguing and I would appreciate more examples of the creational patterns being used.

Structural patterns are dealt with next - they deal with how classes & objects are composed to form larger structures. They seem quite fun - Adapter (page 139) and Proxy (page 207) seem quite familiar already.

The final part, behavioral patterns, deals with the abstracting of algorithms and the assignment of responsibility. When I think of algorithms I tend to think of things like binary searches or quicksorts - this chapter is too flowery and abstract for me.

The conclusion states that this book is not the final word in patterns. Personally, I believe that Pattern Oriented Software Architecture is the book to read first, followed by Design Patterns, then Pattern Hatching then the Patterns of Programming Languages books. And, perhaps, one day Kevlin Henney will finish writing his patterns book.

This is an important, demanding book for anyone doing Object Oriented Design. It will take careful study and application before its true value is felt.

Ian Bruntlett
ianbruntlett@hotmail.com

Verdict : Highly recommended.

Extendable Software and the Bigger Picture

by Allan Kelly

My last two Overload contributions have described my philosophy of extendable software, why it is important, and how we can implement it in the code we write. But that isn't the end of the story, in fact that is just the start.

To create extendable systems you need space – space to write the extensions in, space to practice your art, space to think. If you're cramped into a small directory tree on your disc, or squeezed by working procedures, or forever fixing the build, you simply don't have the space you need. To this end extendable software is as much about process as it is about code, and this is where things get really interesting.

Extendable software naturally fits into an Agile development methodology. I've started from the code level and I'm working up, saying, "What do I need to do to support this style of code with my processes?" The Agile methodologists have started from process and are working towards code.

Software has a logistics tail (see side box). We ignore this tail at our peril, if we are to keep advancing our development we must pay attention to the build system, source code control, etc., etc.. If not, this tail will eventually wag the dog – you'll be so busy getting things to build you won't have time to fix the faults.

This article and the ones that follow are concerned with aspects of that tail which I think are neglected: directory structures, source code control, build systems and how these build into a process.

Looking to the bigger picture

Embracing extensibility in your system architecture is only the start of the story. It is worthless if you do not provide the support services needed. So, a development process that embraces extensibility will enhance the value of the source code that represents the architecture. Both the logical and the physical architecture must be aligned to this. Aligning the physical architecture means a coherent directory structure, which will enhance the value of source code control, which in turn enhances the build system.

This will remove uncertainty, thus improving repeatability and contributing to a successful product. A successful product will validate your process and architecture; we have a positive feed-back loop (see diagram on next page).

Our process influences the logical design of a system (Conway's law), which obviously affects the physical design and coding, this in turn affects the directory structure we use, which itself influences our source code control, and finally we get to build our product.

And none of this happens in isolation, actions feed backwards too, if you are hobbled by an ineffective source code control system you will find your design warps. All of these items interlock.

For example, one of the tenets of Extreme Programming (XP) is *Continuous Integration*, how can you hope to continuously integrate if you don't have a well-defined build process?

Too often we have been sold magic bullets, a development process that doesn't define an integration approach, or a source code control system that doesn't fit our process.

Enough!

In the name of extendable software we have to look at the bigger picture.

The Software Logistics Tail

Have you noticed that when you start coding you can move fast, laying down lots of code in a short time, but the more code you lay down the slower your rate of progress becomes? Likewise, when we think of a new feature we sometimes get the code written quite fast, but other times a relatively small feature takes an inordinate amount of time. Sometimes we get bogged down with the practicalities of life, checking in, checking out, building, fixing link problems, repairing our machine, dealing with ripple effects, and so on.

Software development has a logistics tail, just like an army advancing we need support, to move forward requires logistics support. When an army advances the assault troops at the front of the column need to be constantly supplied with fresh ammunition, food, medical facilities and countless other items. History provides many examples of armies that have advanced beyond their logistics support - or failed to push home their advantage for fear of running ahead of their logistics tail.

Paratroopers can capture objectives but they can't hold them alone. Assault troops may appear lightweight, fast and agile but they won't win the battle alone, they need heavy support and logistics.

In software development the logistics tail contains source code control systems, build systems, testers, system administration, database administration, release mechanisms, customer support, fault tracking systems and so on. Nor is it confined to activities, the logistics tail exists inside our source code too.

We have to consolidate our gains if we are not to lose them. A new feature may be demonstrated quite quickly, but we must ensure it is placed within a well-defined program structure, we need to give it a place, safely embedded in a library or DLL, not hacked onto some oddball Windows message. It is not enough to capture a new feature, we must secure it before we can move onto the next objective.

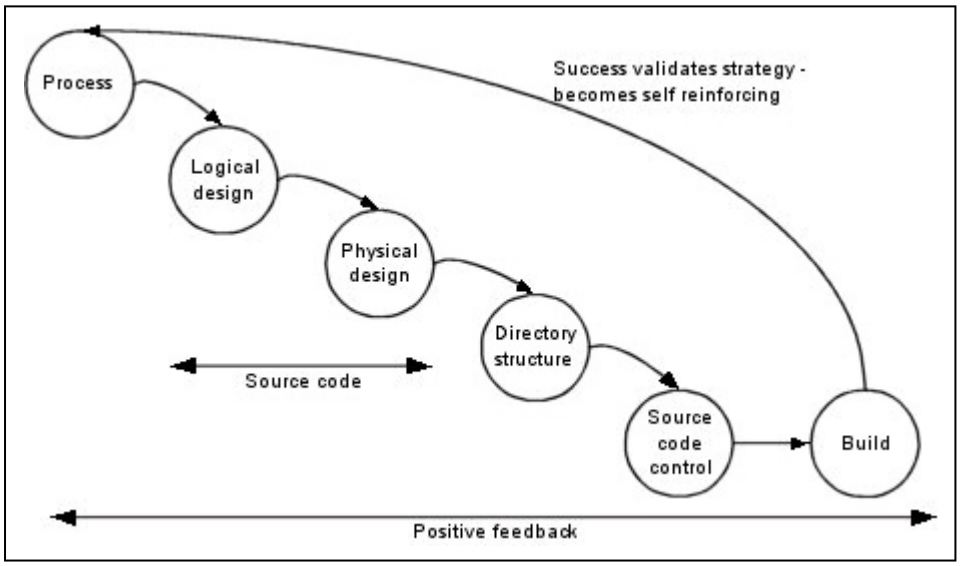
If we have some tricky code we treat it like a minefield, we don't hide this irritating detail in the hope that nobody will encounter it. Instead we **WRITE IT BIG**, with a great big sign saying 'Danger!' – not just a comment, but an assert, or better, a static assert, or maybe we ensure code can only be called in the safe way.

Having a well-defined and reliable logistics tail means we can move forward with confidence. Imagine working without a source code control system, imagine working 'manually' – flipping the 'read only bit', slowly making changes to different files, manually finding out who has the latest copy of any particular piece of code, distributing our results – the problem is exponential in size. So while setting up a CVS server may seem like a distraction from the objective it is an essential part of securing the territory.

Of course this military analogy may seem somewhat dramatic, nobody ever shot a software developer for failure to deliver on time, but dramatic ideas allow us to visualise, and help produce a clear, communicable idea.

Maybe a few of you reading this, will be thinking: 'I want to be the SAS of software development, get in, do the job and get out fast' - nice idea, and perhaps something to aim for. But don't forget that meticulous planning that goes into such operations, and even the SAS couldn't operate without a logistics tail of support staff, helicopter pilots, ground crew, training and special equipment. Perhaps ironically, the logistics tail becomes more important if you want to be truly responsive and agile.

(Of course it helps if you know how to implement polymorphism in 15 different ways with your bare hands.)



Strategic fit

What we are actually talking about here is not just a process – itself a much overloaded word in software – but a strategy which will fit the various pieces of development tightly together and interlock them, and lock software development into the company. It may come as a surprise to some software developers but we are not alone here, discussing business strategy Michael E. Porter of Harvard Business School writes:

> *"What is Southwest's [Airlines] core competence? Its key success factors? The correct answer is that everything matters. Southwest's strategy involves a whole system of activities, not a collection of parts. Its competitive advantage comes from the way its activities fit and reinforce one another.*
>
> *Fit locks out imitators by creating a chain that is as strong as its* strongest *link." Porter, 1996*

Of course Southwest does have imitators, just look at Ryan Air and EasyJet in Europe, but in its home market Southwest is *the* success story of the American airline industry since deregulation.

To isolate one element of the development process, say writing a specification, and divorce it from the process as a whole is wrong. Each activity needs to be consistent with the other activities, Porter calls this "first order fit", he goes on to define second order fit as "activities that are reinforcing" and finally, third order fit as "optimisation of effort."

We can't expect to write extendable software in isolation, in fact, we can't expect to write any software in isolation. Nor can we expect to look at software development isolated from the rest of the organisation.

Process must accept addition

Our aim in an extendable system is to allow new features and fixes to be implemented in new code. New code may contain its own mistakes but the chances of introducing a fault into existing code are reduced substantially. Let's dwell on that for a moment: how many occasions have you fixed some code only to find a new problem has been introduced in the same code? Would that fault have been introduced if the change had been implemented in fresh code? Yes, the fresh code may have its own faults but we expect new code to have faults and hence test for it.

So, if we are to enhance our system through addition, our development process must accept addition too.

There are two places addition may take place: in existing files or in new files. The former is often necessary, say, to add a new method for an existing class, the later usually implies we are adding new classes, we are working with our architecture.

However, our development process and environment can make these additions difficult. Simplistic management using lines-of-code as a metric can discourage developers from making additions. This is short sighted management, code maintained under such a regime sprouts control flags as functions are coerced to do double duty, what should be two functions, or even two classes, gets implemented as a single function:

```
void CalculateInterest(int accountNumber,
bool isDeposit) {
  if (isDeposit) {
    // .... do deposit interest rate
    // calculation
  }
  else {
    // .... do current account
    // calculation
  }
}
```

It doesn't take much foresight to see what happens when we get a third type of account. Almost as obvious is the question: should this be a class hierarchy?

The problem here is short sighted management who have either failed to realise the relationship between process and code, or, managers who are actively trying to architect the system themselves. True, adding the control flag was a cheap way of getting the functionality quickly – and hence improving profits – but this has introduced future cost to the system. Unfortunately, anyone who regards lines-of-code as a good software metric probably isn't going to be persuaded by such arguments.

New features, New files

The creation of new files should be a natural part of the software process but it doesn't always feel this way. One well-known software house I know is proud of its ISO-9001 procedures that include a paper audit trail for a system. Adding new files to the system required a form to be filled in and signed off. This was a sure fire way to ensure that existing files grow unnaturally large – even if there where only a dozen or so files.

Similar things happen if we deny developers access to source code control or don't automate the build process. Buying each developer a license for a top quality source code control system may seem a little pointless, after all they only use it for a few minutes each day, why not buy one license? Maybe, to keep our paper trail accurate Fred can run the source code control, and we can request files by e-mail, or signed form, and he can get them out and e-mail them back? Or maybe just place them on a shared drive?

This may all sound like a sick joke but I've seen it done.

It is human nature to do the easy things and avoid the hard things. If we want our system to grow in an orderly manner we can't put obstacles in the way of what we want to happen. Architecture designed for extendability will fail if we don't align our process. It must be easy to integrate new code.

Paper trails and proxy results

One of the keys to alignment is to automate as much as possible, thus making it easy to do the right things. If you really must have a paper trail for all file changes then configure your source code control system to produce the necessary documents. If you must have sign-off before code enters the system, then use a promotion model for your files. If your ISO-9001 procedures are getting in the way then change them.

In the worst case the process documentation says one thing and developers do it another way then fake the paper trail, sometimes with management connivance and sometimes without their knowledge. Once you've crossed this Rubicon code quality and communication will rapidly deteriorate.

At the end of the day we want to deliver software, this means writing code. It does not mean writing documents or conforming to lines-of-code estimates. A process that emphasises such criteria is targeting a proxy result rather than the end product. Not only is the proxy the wrong measurement but it distracts us from our main goal and is subject to manipulation.

Monitoring a proxy variable can have a place, take file version numbers for instance. If our average file has undergone 10 revisions but one particular file has undergone 30 then it is worth investigating why. Revision number may be a proxy variable but they are by-products of our normal working practice, we don't expend any extra effort to produce the revision number, thus, is less susceptible to manipulation.

Overnight builds *aka* the batch build

One of the corner stones of any development should be an automated batch build process. Having said that all development processes need to be customised I'm saying categorically: create an automated batch build and run it every night. I'm not alone in this.

> *"In our global survey we found that 94% of successful companies completed daily or at least weekly builds, whereas the majority of less successful companies did them monthly or less often." Hoch 2000.*
>
> *"If you build it, it will ship. If you don't, it won't. ... I don't mean 'Build it once and ship it.' I mean 'Build it often and regularly.' You must get that product visible. Public." McCarthy 1995.*

And after all, what is XP's continuous integration but a batch build?

A repeatable build shows you can build your product. But there is more to it than that. It shows that for all the hard work and money thrown at the problem there is some kind of solution, not just strange files on developers PCs. It even shows that there is something that might ship sometime soon.

A regular build also acts as a restraint on developers. You won't check in some half-baked code that will break the build. And if you do it is easy to see who did it. We don't want to get into blame culture here, we just want to encourage everyone to be responsible.

And when something does break the build there is a clear audit trail to find out why. There is usually a good reason and not always because someone messed up. Why wait until you've finished coding to see if all the bits fit together?

The build is also a ritual of software development. Human civilisation is built on rituals, marriage, births, cards for this, cards for that, and so on. Rituals give us anchor points in a changing, uncertain world. A ritual build serves the same use.

The build provides a heartbeat to a project: you come to work, you write code, you check it in, it builds, you start over again. When the project won't build there is usually something wrong.

As McCarthy says, make it public. Let everyone see it works, e-mail the build log to the developers, display the latest build number on a flashing sign in the lobby. Show you are active. It easy for organisations to lose sight of the work of developers, especially if you're just a bunch of geeks sitting in Dilbert cubes.

But the build has to be automated. It has to be carried out by machine – free of human interference. Human intervention introduces a random factor. Automating the process proves it is understood, proves it is repeatable. If you can't automate it and repeat it what does that say about the state of your project? Doing a build is repetitive, boring, easily forgotten, time consuming. All the things that humans don't like and machines are good at.

So, how do we get there?

We have the pieces of a solution to hand but you are on your own in putting them together. Yes, we have models we can follow, but there are no guarantees that they will work for us, in our environment. Quite the opposite in fact. Alistair Cockburn (2002) says something similar "The level 3 listener [an experienced practitioner] knows that all the published software development techniques are personal and somewhat arbitrary."

In fact, Porter might argue that we must develop our own methodologies! Implementing a given methodology may well improve our operational effectiveness, which in turn improves our ability to delivery value to our organisations...

> *"However, it [operational efficiency] is not usually sufficient. Few companies have competed successfully on the basis of operational efficiency over an extended period, and staying ahead of rivals gets harder every day. The most obvious reason for that is the rapid diffusion of best practice. Competitors can quickly imitate management techniques, new technologies, input improvements, and superior ways of meeting customer needs.*
>
> *The second reason that improved operational effectiveness is insufficient – competitive convergence – is more subtle and insidious. The more benchmarking companies do, the more they look alike." Porter, 1996*

So, even if we could successfully implement, to the letter, SSADM, RUP, Yourdon, XP, or any other methodology it would do us no good. Competitors could just copy us. What we need is some unique strategy, methodology, which we have tailored to our business needs.

Contrast Porter's words with those of Goldman cited by Cockburn:

> *"Agility is dynamic, context-specific, aggressively change embracing, and growth-oriented. It is not about improving efficiency, cutting costs, or battering down the business hatches to ride out fearsome 'storms.' It is about succeeding and about winning: about succeeding in emerging competitive arenas, and about winning profits, market share, and customers in the very centre of the competitive storms many companies now fear." Goldman, 1997.*

[concluded at foot of next page]

Mutate? or Create?

by Alan Griffiths

"Who is the audience?" – is a key question to consider for any piece of writing and one that programmers often forget to address. Just because the language is C++, Java or Cobol doesn't relieve the author of the responsibility to consider the audience. The code that I write to explore an idea for myself will look very different to the code I write to illustrate the idea in an article and the code that uses the idea in a production environment will look different to both of these.

It is interesting to hear some of the answers programmers give when asked to describe the audience for whom they are writing. The most common answer is to deny that the question is meaningful. Other answers include "there isn't one" and "the compiler". And, to judge from most of the code I see, this is the belief in even more cases than the frequency of these answers suggests.

When I write production code the audience splits into two categories: The main audience is those that will use it from their code but there is also a smaller group who will maintain it. (And the audience for this article also writes code for these audiences.)

What follows describes the meeting between a typical developer and an audience that will be making use of it. The context of this meeting is a review of the public interfaces to the classes in a package prior to implementing them. Present at the review are George (the author), Alice (the system designer) and Harris (another programmer on the project). While this is a true account I have coloured the events and changed the names.

What do we mean by "add"?

George, Alice and Harris filed nervously into the room. This was the first review of "class designs" and none of them knew what to expect. George had followed my instructions and supplied Javadoc documentation of the package containing the classes being reviewed. Everyone had a printed copy.

I recited the prayer customary on these occasions: "we are here to help George improve his design, to identify issues he needs to consider more fully; we are not here to score points or to do the design ourselves."

The classes to be reviewed related to the manipulation of measures that have values, units and dimensions (e.g. [100, metres, length] or [5, kilos, weight]). Much can be (and has been) said about combining disparate units and commensurate dimensions. It has been said many times, by many people and in many places. I'm not going to repeat it here! Instead, I want to focus on the discussion of these two methods in the `Value` class:

```
class Value implements Cloneable {
  public static Value add(Value a, Value b)
  public Value add(Value v)
  ...
}
```

I have deliberately given the methods in this form - and not the Javadocs that were reviewed - because I want you, the reader, to think about the semantics you would expect from these methods before reading on.

This link between the name and the semantics is where considering the audience becomes important. A compiler is not critical of the choice of names – it doesn't care if a method called '`add`' is used to print a sequence of Goldbach prime pairs. A programmer, however, might express surprise - even if there was documentation to that effect.

Alice had her opinion of what the `add` methods should do and, when we got to the description of one of these methods, asserted

[continued on next page]

So, if you're sitting back and saying "I have enough trouble getting software written, never mind worrying about the business" you may want to consider what role your software plays in keeping your company competitive.

Software development is about business. It is about selling products – directly or indirectly – so all the concerns of the business are the concerns of the developers. As more and more business depends on software development for their very existence (selling it or using it) the very process of software development becomes a key business function, and a key business discriminator.

It is easy to see how these kinds of arguments relate to shrink-wrapped product software but this accounts for only a small percentage of all software written. Increasingly companies are their software. Many of today's business processes would be impossible without software, the process gives the business its competitive edge, and without software to enable to the process there would be nothing.

Where do I go from here?

You probably have most of the pieces you need from experience. My last two articles have set out an approach, an attitude, toward software that, hopefully, adds some more pieces to the jigsaw. There is no shortage of literature on process but I hope you found something new here.

What I think is missing, and where I want to turn my attention next is the nuts-and-bolts of how we organise our source code (directory trees and source code control) and how we get continuous integration (the batch build) to work.

Actually, writing about build systems is something I've wanted to do for a while and is surprisingly difficult. Part of the reason is that it is incredibly difficult to write about in isolation from directory structures and source control. However, I am conscious that I promised ACCU-General a while ago that I'd write something on build systems, hopefully, by the time you read this I'll have late drafts of the next two pieces available at `http://www.allankelly.net/writing`.

Allan Kelly
`Allan.Kelly@bigfoot.com`

References

Beck, Kent, 2000, *Extreme Programming Explained*, Addison-Wesley.

Cockburn, Alistair, 2002, *Agile Software Development*, Addison-Wesley.

Goldman, S., Nagle, R., Preiss, K., 1995, *Agile Competitors and Virtual Organizations*, John Wiley & Sons.

Hoch D.J., Roeding, C.R., Purkert, G., Linder, S.K., 2000, *Secrets of Software Success*, Harvard Business School Press.

Mazzucato, Marinana, 2002, *Strategy for Business*, Sage Publication / Open University.

McCarthy, Jim, 1995, *Dynamics of Software Development*, Microsoft Press.

Porter, Michael E., 1996, "What is Strategy," *Harvard Business Review*, November/December 1996, reprinted in Mazzucato 2002.

that it "didn't do what one would expect". I hope you have formed your own expectation of what these methods do. How does it correspond to the following (the documented) behaviour?

```
/**
* returns a new Value containing a
* quantity that is the sum of the
* quantities held in 'a' and 'b'
*/
public static Value add(Value a, Value b)
/**
* returns a new Value containing a
* quantity that is the sum of those held
* in the current object and that supplied
*/
public Value add(Value v)
```

Alice was not in disagreement about the first of these but the second led to a lot of discussion. (In the end the description, the signature and the name of the method all came in for criticism – but we are getting ahead of the story.) To illustrate her point Alice took possession of the whiteboard and produced the following code fragment:

```
Value accumulate(Iterator i) {
  final Value total = new Value();
  while (i.hasNext()) {
    total.add((Value)i.next());
  }
  return total;
}
```

(Actually, Alice's code used shorter identifiers "V" for "Value", "t" for "total", but she and I are writing in a different context and code differently as a consequence.)

She claimed that this code has surprising behaviour. George responded by asserting that "one shouldn't use a method without referring to the documentation". Harris was in full agreement "I don't care what its called I can see what it does".

George took over the whiteboard and suggested the 'correct' code:

```
Value accumulate(Iterator i) {
  Value total = new Value();
  while (i.hasNext()) {
    total = total.add((Value)i.next());
  }
  return total;
}
```

I think that it is significant that no one queried what Alice thought the original code should achieve. In claiming that it was wrong George and Harris are missing the point - the documentation of a method should support the name given, not invalidate it. Alice suggested that the description should read "adds the supplied value to the current value" or "adds the supplied value to the current value and returns this".

George's correction leads to code that has the intended effect but Harris observed that it creates numerous temporary `Values`. This leads to the title question:

Should `add` mutate a `Value` instance? Or create a new`Value`? With Alice's change the user has the option of both behaviours. Mutate:

```
total.add((Value)i.next());
```

or create:

```
total = Value.add(total,
(Value)i.next());
```

George accepted that the suggested behaviour would be useful, but didn't think that the mutating behaviour was reflected by the name `add` – if you calculate "1 add 2" then neither of the numbers changes to "3".

Harris then suggested that the return type of this `add` method should be `void`. His suggestion that the return type should be `void` came from the desire to invalidate `total = total.add((Value)i.next());`. This seemed reasonable (but we'll revisit this after hearing what George had been thinking about name of the method).

At this point the meeting moved on to other matters. George took the issues raised away to consider and later circulated a proposal that resolved them.

After drawing an analogy with the + and += operators George suggested that the correct name would be `plusEquals`. Accepting this argument implies that the original static method should also be renamed `plus`. In the light of this Harris is concerned about the analogue of "`i = i += 1;`". This is legal (but clearly daft) for the primitive types and so, by analogy, "`total = total.plusEquals((Value)i.next());`" should also be legal.

Consequently, George proposed (and ultimately implemented) the following:

```
/**
* returns a new Value containing a
* quantity that is the sum of the
* quantities held in 'a' and 'b'
*/
public static Value plus(Value a, Value b)
/**
* Adds the supplied value to that
* contained in the current object.
* @return this
*/
public Value plusEquals(Value v)
```

Afterword

The discussion above took about 15 minutes and involved four developers. It avoided any developer that used `Value` writing code like Alice's example and subsequently spending an indeterminate amount of time looking for the problem.

The fact that the review forced documentation to be written for the `add` method also prevented someone writing code like Alice's finding the problem and then "correcting" the `add` method to the detriment of any code that used it as intended.

Are these changes important? Considered from the compiler's point of view it makes little difference. From the available functionality it makes little difference. From the point of view of providing an unsurprising interface that is easy to use effectively it does make a difference.

Alan Griffiths
alan.griffiths@microlise.com

Exception Handling in C#

by Jon Jagger

Painful Procedural Error Handling

In the absence of exceptions the classic way to handle errors is to intertwine your statements with error checks. For example:

```
public sealed class Painful {
  ...
  private static char[] ReadSource(
                             string filename) {
    FileInfo file = new FileInfo(filename);
    if (errorCode == 2342) goto handler;
    int length = (int)file.Length;
    char[] source = new char[length];
    if (errorCode == -734) goto handler;
    TextReader reader = file.OpenText();
    if (errorCode == 2664) goto handler;
    reader.Read(source, 0, length);
    if (errorCode == -5227) goto handler;
    reader.Close();
    Process(filename, source);
    return source;
    handler:
    ...
  }
}
```

This style of programming is tedious, repetitive, awkward, complex, and obscures the essential functionality. And it's too easy to ignore errors (either deliberately or accidentally). There have to be better ways. And there are. But some are better than others.

Separation of Concerns

The fundamental thing that exceptions allow you to do is to *separate* the essential functionality from the error handling. In other words, we can rewrite the mess above like this:

```
...
public sealed class PainLess {
  public static int Main(string[] args) {
    try {
      string filename = args[0];
      char[] source = ReadSource(filename);
      Process(filename, source);
      return 0;
    }
    catch (SecurityException    caught) {...}
    catch (IOException          caught) {...}
    catch (OutOfMemoryException caught) {...}
    ...
  }
  private static char[] ReadSource(
                             string filename) {
    FileInfo file = new FileInfo(filename);
    int length = (int)file.Length;
    char[] source = new char[length];
    TextReader reader = file.OpenText();
    reader.Read(source, 0, length);
    reader.Close();
    return source;
  }
}
```

There are several things to notice about this transformation.

- The numeric integer error codes that utterly failed to describe the errors they represented (e.g. what does `2342` mean?) are now descriptive exception classes (e.g. `SecurityException`).
- The exception classes are not tightly coupled to each other. In contrast, each integer code must hold a unique value thus coupling all the error codes together.
- There is no `throw` specification on `ReadSource`. C# does not have `throw` specifications.

However, by far and away the most important thing is how clean, simple and easy to understand `ReadSource` is. It contains the statements required to implement its essential functionality and nothing else. There is no apparent concession to error handling. This is possible because if an exception occurs the call stack will unwind all by itself. This version of `ReadSource` is the "ideal" we are aiming it. It is as direct as we can make it.

Ironically, exceptions allow us to get close to this ideal version of `ReadSource` but at the same time prevent us from quite reaching it. The problem is that `ReadSource` is an example of code that acquires a resource (a `TextReader`), uses the resource (`Read`), and then releases the resource (`Close`). The problem is that if an exception occurs after acquiring the resource but before releasing it then the release will not take place. The solution has become part of the context. Nevertheless, this "ideal" version of `ReadSource` is useful; we can compare forthcoming versions of `ReadSource` to it as a crude estimate of their "idealness".

finally?

The solution to this lost release problem depends on the language you're using. In C++ you can release the resource in the destructor of an object held on the stack (the misnamed Resource Acquisition Is Initialization idiom). In Java you can use a finally block. C# allows you to create user-defined struct types that live on the stack but does not allow struct destructors. (This is because a C# destructor is really a `Finalize` method in disguise and `Finalize` is called by the garbage collector. Structs, being value types, are never subject to garbage collection.) Therefore, initially at least, C# must follow the Java route and use a finally block. A first cut implementation using a finally block might look like this:

```
private static char[] ReadSource(
                             string filename) {
  try {
    FileInfo file = new FileInfo(filename);
    int length = (int)file.Length;
    char[] source = new char[length];
    TextReader reader = file.OpenText();
    reader.Read(source, 0, length);
  }
  finally {
    reader.Close();
  }
  return source;
}
```

This version has had to introduce a try block (since a finally block must follow a try block) which isn't in the ideal solution but apart from that it's the same as the "ideal" version of `ReadSource`. It would be a reasonable solution if it worked. But it doesn't. The problem is that the try block forms a scope so `reader` is not in scope inside the finally block and `source` is not in scope at the return statement.

finally?

To solve this problem you have to move the declarations of `reader` and `source` outside the `try` block. A second attempt might be:

```
private static char[] ReadSource(
                            string filename) {
  TextReader reader;
  char[] source;
  try {
    FileInfo file = new FileInfo(filename);
    int length = (int)file.Length;
    source = new char[length];
    reader = file.OpenText();
    reader.Read(source, 0, length);
  }
  finally {
    reader.Close();
  }
  return source;
}
```

This version has moved the declaration of `reader` and `source` out of the try block and consequently, inside the try block, *assigns* to `reader` and `source` rather than *initializing* them. That's another difference (and two extra lines) from the "ideal" version of `ReadSource`. Nevertheless, you might consider it a reasonable solution if it worked. But it doesn't. The problem is that assignment is not the same as initialization and the compiler knows it. If an exception is thrown before reader is assigned then the call to `reader.Close()` in the finally block will be on `reader` which won't be assigned. C#, like Java, doesn't allow that.

finally?

Clearly you have to initialize `reader`. A third attempt therefore might be:

```
private static char[] ReadSource(
                            string filename) {
  TextReader reader = null;
  char[] source;
  try {
    FileInfo file = new FileInfo(filename);
    int length = (int)file.Length;
    source = new char[length];
    reader = file.OpenText();
    reader.Read(source, 0, length);
  }
  finally {
    reader.Close();
  }
  return source;
}
```

This version introduces `null` which isn't in the "ideal" version of `ReadSource`. Nevertheless, you might still consider it a reasonable solution if it worked. But it doesn't (although it does compile). The problem is the call to `reader.Close()` could easily throw a `NullReferenceException`.

finally?

One way to solve this problem is to guard the call to `reader.Close()`. A fourth attempt therefore might be:

```
private static char[] ReadSource(
                            string filename) {
  TextReader reader = null;
  char[] source;
  try {
    FileInfo file = new FileInfo(filename);
    int length = (int)file.Length;
    source = new char[length];
    reader = file.OpenText();
    reader.Read(source, 0, length);
  }
  finally {
    if (reader != null)
      reader.Close();
  }
  return source;
}
```

Of course, the guard on `reader.Close()` isn't in the "ideal" version of `ReadSource`. But this is a reasonable version if only because it does, finally, work. It's quite different from the "ideal" version but with a bit of effort you can refactor it to this:

```
private static char[] ReadSource(
                            string filename) {
  FileInfo file = new FileInfo(filename);
  int length = (int)file.Length;
  char[] source = new char[length];
  TextReader reader = file.OpenText();
  try {
    reader.Read(source, 0, length);
  }
  finally {
    if (reader != null)
      reader.Close();
  }
  return source;
}
```

In some cases you might be able to drop the `null` guard inside the finally block but in general this is the best you can do with a finally block solution. (Consider if `file.OpenText` returned `null`.) You have to add a try block, a finally block, and an if guard. And if you are using Java you have to do those three things *every* time. And therein is the biggest problem. If this solution was truly horrible and completely and utterly different to the ideal solution it wouldn't matter a jot if we could abstract it all away. But in Java you can't. The Java road stops here, but the C# road continues.

using Statements

In C#, the nearest you can get to the "ideal" version is this:

```
private static char[] ReadSource(
                            string filename) {
  FileInfo file = new FileInfo(filename);
  int length = (int)file.Length;
  char[] source = new char[length];
  using (TextReader reader = file.OpenText())
    { reader.Read(source, 0, length); }
  return source;
}
```

This is pretty close. And as I'll explain shortly it has a number of features that *improve* on the "ideal" version. But first let's look under the lid to see how it actually works.

using Statement Translation

The C# ECMA specification states that a using statement:

```
using (type variable = initialization)
  embeddedStatement
```

is equivalent to:

```
{
  type variable = initialization;
  try { embeddedStatement }
  finally {
    if (variable != null) {
      ((IDisposable)variable).Dispose();
    }
  }
}
```

This relies on the `IDisposable` interface from the `System` namespace:

```
namespace System {
  public interface IDisposable {
    void Dispose();
  }
}
```

Note that the cast inside the finally block implies that variable must be of a type that supports the `IDisposable` interface (either via inheritance or conversion operator). If it doesn't you'll get a compile time error.

using TextReader Translation

Not surprisingly, `TextReader` supports the `Disposable` interface and implements `Dispose` to call `Close`. This means that this:

```
using (TextReader reader = file.OpenText()) {
  reader.Read(source, 0, length);
}
```

is translated, under the hood, into this:

```
{ TextReader reader = file.OpenText();
  try { reader.Read(source, 0, length); }
  finally {
    if (reader != null) {
      ((IDisposable)reader).Dispose();
    }
  }
}
```

Apart from the cast to `IDisposable` this is identical to the best general Java solution. The cast is required because this is a general solution.

Do It Yourself?

It's instructive to consider what would happen if `TextReader` didn't implement the `Disposable` interface. The lessons from this will show us how to implement Disposability in our own classes. One solution is the Object Adapter pattern. For example:

```
public sealed class AutoTextReader
                                : Idisposable {
  public AutoTextReader(TextReader target) {
    // PreCondition(target != null);
    adaptee = target;
  }
  // readonly property
  public TextReader TextReader {
    get { return adaptee; }
  }
  public void Dispose() {
    adaptee.Close();
  }
  private readonly TextReader adaptee;
}
```

which you would use like this:

```
using (AutoTextReader scoped =
       new AutoTextReader(file.OpenText())) {
  scoped.TextReader.Read(source, 0, length);
}
```

To make things a little easier you can create an implicit conversion operator:

```
public sealed class AutoTextReader
                                : Idisposable {
  ...
  public static implicit operator
         AutoTextReader(TextReader target) {
    return new AutoTextReader(target);
  }
  ...
}
```

which would allow you to write this:

```
using (AutoTextReader scoped =
                     file.OpenText()) {
  scoped.TextReader.Read(source, 0, length);
}
```

struct Alternative

`AutoTextReader` is a sealed class intended, as its name suggests, to be used as a local variable. It makes sense to implement it as a struct instead of class:

```
public struct AutoTextReader : Idisposable {
  // exactly as before
}
```

Using a struct instead of a class also gives you a couple of free optimizations. Since a struct is a value type it can never be null. This means the compiler can omit the null guard from generated finally block. Also, since you cannot derive from a struct its runtime type is always the same as its compile time type. This means the compiler can also omit the cast to `IDisposable` from the generated finally block and thus avoid a boxing operation. In other words, when `AutoTextReader` is a struct, this:

```
using (AutoTextReader scoped =
                     file.OpenText()) {
  scoped.TextReader.Read(source, 0, length);
}
```

is translated into this:

```
{
  AutoTextReader scoped = new file.OpenText();
  try {
    scoped.TextReader.Read(source, 0, length);
  }
  finally {
    scoped.Dispose();
  }
}
```

It should come as no surprise that I prefer the using statement solution to the finally block solution. In fact, the using statement

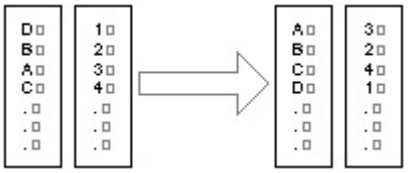
[concluded at foot of next page]

Pairing Off Iterators

by Anthony Williams

Introduction

Recently, a colleague approached me with an interesting problem; he had two containers with corresponding elements, so the *n*-th entry of container *A* was related to the *n*-th entry of container *B*, and he needed to sort these containers so the elements of *A* were "in order", without losing the correspondence property, as in the figure below:



There are several solutions to this, each of which has its merits and disadvantages, such as:

1 Create a third container holding either the numeric index, or some form of iterator or pointer into the containers. This container can then be sorted using a special comparison function that references the original containers. Code that processes the data can then either use the new container to index into the originals, or the original containers can be re-shuffled to match the order specified in the new container.

2 Copy the values into a container of pairs and sort that, possibly using the self-sorting property of the Standard Associative Containers, if appropriate

It was a third solution that really interested me – what my colleague conceptually had was a container of pairs of values, even if it was physically stored as a pair of containers of values; why then couldn't we treat the data *as* a container of pairs? This would allow sorting in place, and intuitive access to the data. The answer is: we can – just write an iterator adaptor that iterates through both containers simultaneously, and returns a pair when dereferenced; the pair of containers can thus be viewed as a sequence of pairs of values. The rest of this article covers the complexities hidden in that "just".

Iterator Categories

To cover the original problem (sorting), we need only worry about random access iterators, since `std::sort` requires random access. However, the problem had me hooked, and I wanted a general solution, with maximum flexibility, and all the complexities that involved.

For maximum flexibility, we want our adapted iterator to be as capable as possible, but no more – we cannot efficiently provide more facilities than the underlying iterators. Therefore, we must choose the most basic iterator category of the underlying iterators. Since the iterator category tags other than `output_iterator_tag` form an inheritance hierarchy, we can use the implicit conversions applied by the ternary conditional operator `?:` to determine the most basic category – the return type of a conditional expression where the two result expressions are pointers is a pointer to the common base class of the pointed-to classes, so the type of the expression

```
false ? (std::forward_iterator_tag*)0
      : (std::random_access_iterator_tag*)0
```

is `std::forward_iterator_tag*`, for example. We can then code to handle output iterators separately – if either of the iterators is an output iterator, the result is an output iterator, unless the other is an input iterator, in which case it is an error. This is all handled by the `CommonCategory` class template, shown in listing 1 – the `categoryCheck` functions and `CategoryMap` class templates are shown in listing 2.

Meeting the Requirements

Having chosen our iterator category, we need to implement the appropriate operations to fulfil the Standard Iterator Requirements from Section 24.1 of the C++ Standard [1]. Most operations can easily be implemented by forwarding to the corresponding operations on the underlying iterators – the pre-increment operator can be implemented by incrementing the underlying iterators, for example. It is the dereference operator (`operator*`) and the choice of `value_type` which is complicated, as it depends on the iterator category. Input Iterators *may* return by value, so if either of the underlying iterators is an Input Iterator, we need to copy the result of dereferencing the underlying iterators. On the other hand, the only operation permitted on the result of dereferencing an Output Iterator is to assign to it, so we cannot store a copy – the dereference operator must return something which, when assigned to, assigns to the result of dereferencing the underlying iterators. Finally, for all other iterators, we need to return a reference that can be used to access and update the elements of the underlying sequences.

The `ValueForCategory` class template assists us with our choice – the `PairIt` iterator template just delegates to `ValueForCategory`, once the appropriate `iterator_category` has been determined, and this is

---

[continued from page 25]

solution scores several extra points in comparison to the "ideal" solution. A using statement

- Works! It always releases the resource.
- Is an extensible mechanism. It allows you to create an abstraction of resource release. Creating your own resource releaser types such as `AutoTextReader` is easy.
- Allows you to pair up the resource acquisition with the resource release. The best moment to organize the resource release is the moment you acquire the resource. If you borrow a book from a library you're told when to return it as you borrow it.
- Creates a scope for the variable holding the resource. Look carefully at the compiler translation of a using statement and you'll see that it cleverly includes a pair of outer braces:

```
using (AutoTextReader scoped =
                              file.OpenText()) {
  scoped.TextReader.Read(source, 0, length);
}
scoped.TextReader.Close(); // scoped is not
                           // in scope here
```

This is reminiscent of C++ declarations in conditions. Both allow you to restrict the scope of a variable so it's only usable when in scope and is only in scope when usable. This is more than just a syntactic nicety since any attempt to use a released resource could well throw an exception.

Jon Jagger
jon@jaggersoft.com

```
template<typename Cat1,typename Cat2>
struct CommonCategory {
  private:
    enum {categorySize=sizeof(
            helper::categoryCheck(
            false?(Cat1*)0:(Cat2*)0))};
  public:
    typedef typename
         CategoryMap<categorySize>::Type Type;
};
// specializations
template<typename Cat>
struct
  CommonCategory<std::output_iterator_tag,Cat> {
  typedef std::output_iterator_tag Type;
};
template<typename Cat>
struct
  CommonCategory<Cat,std::output_iterator_tag> {
  typedef std::output_iterator_tag Type;
};
template<>
struct CommonCategory<std::output_iterator_tag,
                      std::output_iterator_tag> {
  typedef std::output_iterator_tag Type;
};
template<>
struct CommonCategory<std::input_iterator_tag,
                      std::output_iterator_tag> {
  // no Type, because error
};
template<>
struct CommonCategory<std::output_iterator_tag,
                      std::input_iterator_tag> {
  // no Type, because error
};
```

Listing 1: The `CommonCategory` class template

```
// Small, Medium, Large and Huge are types
// with distinct sizes
Small categoryCheck(std::input_iterator_tag*);
Medium
    categoryCheck(std::forward_iterator_tag*);
Large
  categoryCheck(std::bidirectional_iterator_tag*);
Huge
  categoryCheck(std::random_access_iterator_tag*);

template<>
struct CategoryMap<sizeof(Small)> {
  typedef std::input_iterator_tag Type;
};
template<>
struct CategoryMap<sizeof(Medium)> {
  typedef std::forward_iterator_tag Type;
};
// etc.
```

Listing 2: The `categoryCheck` overloaded functions and `CategoryMap` specializations

```
template<typename Iter1,typename Iter2>
struct OutputPair {
  private:
    Iter1& firstIter;
    Iter2& secondIter;
    OutputPair(const OutputPair&); // can't be
                                   // copied
  public:
    OutputPair(Iter1& firstIter_,
               Iter2& secondIter_)
      : firstIter(firstIter_),
                secondIter(secondIter_) {}

    template<typename SomePair>
    OutputPair& operator=(const
                          SomePair& other) {
      *firstIter=other.first;
      *secondIter=other.second;
    }
};
```

Listing 3: The `OutputPair` class template

specialized for Input Iterators and Output Iterators, leaving the primary template to handle the other cases.

Implementing the dereference operator for Input Iterators and Output Iterators is actually quite straightforward. For Input Iterators, the `value_type` can be a plain `pair` of values, the elements of which are the `value_types` of the underlying iterators, and the dereference operator can just copy the values from the underlying iterators into a pair held within the iterator, and return a reference to that pair[1]. For Output Iterators, the `value_type` is `void`, but the result of the dereference operator is something quite different – an `OutputPair` that contains references to the underlying iterators, and which dereferences and writes to the iterators when assigned to. `OutputPairs` cannot be copied, so our iterator's dereference operator should return a reference to an internal instance of `OutputPair`. The definition of `OutputPair` is shown in listing 3.

Supporting Forward Iterators, Bidirectional Iterators and Random Access Iterators is more complicated – the dereference operator must return a reference to the `value_type`, which must hold real references to the elements in the sequences covered by the original iterators. Just to add complexity, we really want the `value_type` to be *Copy-Constructible* and *Assignable*, and to copy the *values* of the elements, not the references, as users wouldn't expect modifying copies of the values to affect the originals; this implies that objects of the same class sometimes contain references to data held elsewhere, and sometimes hold the data directly. For this purpose, we define the `OwningRefPair` class, which has references for its public data members, and contains an internal buffer for the values – the references can either point to external data, in which case the buffer is empty, or they can point to the buffer, in which case the buffer contains instances of the appropriate objects. The objects are stored in an internal buffer, rather than on the heap, to avoid the overhead of dynamic memory allocation; however, this does require care to ensure that the objects are properly destructed, and to ensure that the buffer is correctly aligned.

1 We could return the pair by value, but for uniformity with the other types of iterators, it makes sense to return a reference to an internal object.

```
namespace utils {
  template<typename T>
  struct Struct {
    T t;
  };

  class Unknown;

  union align_t {
    bool b;
    char c;
    short s;
    int i;
    long l;
    wchar_t w;
    float f;
    double d;
    long double ld;
    void* vp;
    void (*fp)();
    void (Unknown::*mfp)();
    Unknown* (Unknown::*mdp);
    Struct<bool> sb;
    Struct<char> sc;
    Struct<short> ss;
    Struct<int> si;
    Struct<long> sl;
    Struct<wchar_t> sw;
    Struct<float> sf;
    Struct<double> sd;
    Struct<long double> sld;
    Struct<void*> svp;
    Struct<void (*)()> sfp;
    Struct<void (Unknown::*)()> smfp;
    Struct<Unknown* (Unknown::*)> smdp;
  };

  template<typename T>
  union RawMem {
    char data[sizeof(T)];
    align_t align;
  };
}
```

Listing 4: `rawmem.hh` (include guards omitted)

For alignment, we use a `union` of an appropriately-sized array of `char`, and an instance of `align_t`. `align_t` is itself a `union` of all the fundamental types, and `structs` containing them. On most platforms, this will have the most rigorous alignment of any type, so (on most platforms[2]) the `union` of `align_t` and the array of `char` is guaranteed to be correctly aligned for any type that has a `sizeof` less than or equal to the size of the array[3]. The `RawMem` template union uses the `sizeof` the template parameter as the size of the `char` array.

We can then cast the address of the `char` array in our `RawMem` union to a pointer of the required type, and use it with placement `new` to construct an instance of the specified type. At the appropriate point, we can also manually invoke the destructor to clean up the object – i.e. in the destructor of our object, we check to see if the buffer contains an object or not, and invoke the destructor if it does, since this is a fixed property of the `OwningRefPair` object – either it contains the referred-to objects in its buffer, or it doesn't, this property doesn't change during its lifetime. In this case, the owned object is an `OwnedPair`; the details are shown in listing 5. Since our `value_types` are distinct, and have different construction syntax, we delegate the actual task of construction and destruction to the `ValueForCategory` template, to give a uniform interface to `PairIt`.

```
#include <rawmem.hh>
template<typename T,typename U>
struct OwningRefPair {
  public:
    T& first;
    U& second;
    typedef T first_type;
    typedef U second_type;
  private:
    struct OwnedPair {
      T first;
      U second;
      template<typename Val1,typename Val2>
      OwnedPair(Val1& v1,Val2& v2)
        : first(v1),second(v2) {}
    };
    utils::RawMem<OwnedPair> pairBuf;
    const bool ownsFlag;
    OwnedPair* getPairPtr() {
      return reinterpret_cast<OwnedPair*>(
                                  pairBuf.data);
    }
    template<typename Val1,typename Val2>
    void createCopy(Val1& v1,Val2& v2) {
      new(getPairPtr()) OwnedPair(v1,v2);
    }
  public:
    OwningRefPair(T& first_, U& second_,
                  bool copy)
      : first(copy ? getPairPtr()->first
                   : first_),
        second(copy ? getPairPtr()->second
                    : second_),
        ownsFlag(copy) {
      if(ownsFlag) {
        createCopy(first_,second_);
      }
    }
    ~OwningRefPair() {
      if(ownsFlag) {
        getPairPtr()->~OwnedPair();
      }
    }
};
```

Listing 5: The `OwningRefPair` class template

2 Platforms *may* arbitrarily choose to make the alignment of one particular user-defined type distinct from that of any other types.

3 For an implementation that discards types bigger than the type we need the alignment of, to avoid wasting space, see [2]

Putting together the fundamentals

Having pinned down the `value_type` for our iterator, and what we get when we dereference it, we can put together a basic version of our `PairIt`, as in listing 6. This highlights a couple of issues. Firstly, we delegate all the type selection to the `PairItHelper` template, so we can inherit from an appropriate instance of `std::iterator<>` without having to specify all the types explicitly. Secondly, even though `std::iterator<>` defines all the required `typedef`s, we have to repeat them here, so we can use them within the class definition; the base class is a dependent name, so it isn't searched during resolution of unqualified names within the class. This begs the question of whether or not we need to inherit from `std::iterator<>` at all; some existing code expects all iterators that aren't raw pointers to inherit from `std::iterator<>`, and doing so causes no harm. We also can reuse the memory management from `OwningRefPair`, so we don't have to rely on any particular properties of the `value_type`s of the underlying iterators, except this time we delegate the construction and destruction to `PairItHelper` as well. Note also that all the members are `mutable` – this is because we don't want to pass on any requirements that the encapsulated iterators be non-`const` for specific operations, and the cache needs to be updated in response to dereferencing the iterator, which is a `const` operation.

Beyond the basics

Now that our iterator supports the basic operation of dereferencing, we need to cover the remaining iterator requirements from the C++ Standard. For Input Iterators, the relevant section is 24.1.1, and table 72. This requires that in addition to dereferencing, we also require:

- Copy-construction,
- Assignment,
- Equality and Inequality operators, and
- Pre- and post-increment operators.

These operations are also sufficient for Output Iterators, as they cover all the requirements from section 24.1.2 and table 73.

In all cases, we can just defer to the underlying iterators, and perform the operations on them. However, there is a consequence for exception safety – since we know nothing about the effects of the operations on the underlying iterators, including whether or not they through exceptions, and whether or not the iterator types support a non-throwing `swap` operation, we have to add a disclaimer to the usage of our iterator – if an operation on a `PairIt` throws an exception, then the iterator is to be considered to have become invalid. If we don't add this disclaimer, then it is possible that the state of the iterator may become confused, as (for example) one of the underlying iterators may have advanced, and the other one not.

Another point to make is that copy-construction and assignment should only copy the iterators, not the cache. This is to avoid unnecessary copying of the cached data, which would only provide an additional source of exceptions for no gain – the cache must be regenerated every time the iterators are dereferenced anyway to support Input Iterators that automatically advance when read (which may not actually be allowed anyway). When dealing with non-Input Iterators, the cached value is only a couple of references, so this should add little performance penalty. The alternative is to have every function that modifies the underlying iterators call `emptyCache()`, and only call `initCache()` if the cache is not initialised.

```
#include <rawmem.hh>
template<typename Iter1,typename Iter2>
class PairIt : public
          PairItHelper<Iter1,Iter2>::IteratorType {
private:
  typedef PairItHelper<Iter1,Iter2> PairDefs;
  typedef typename
              PairDefs::ValueTypeDef ValueTypeDef;
public:
  typedef typename
     PairDefs::iterator_category iterator_category;
  typedef typename PairDefs::value_type value_type;
  typedef typename
     PairDefs::difference_type difference_type;
  typedef typename PairDefs::reference reference;
  typedef typename PairDefs::pointer pointer;
private:
  pointer getValuePtr() const {
    return reinterpret_cast<pointer>(
                                     dataCache.data);
  }
  void emptyCache() const {
    if(cacheInitialized) {
      ValueTypeDef::destruct(getValuePtr());
      cacheInitialized=false;
    }
  }
  void initCache() const {
    emptyCache();
    ValueTypeDef::construct(getValuePtr(),it1,it2);
    cacheInitialized=true;
  }
public:
  PairIt(Iter1 it1_,Iter2 it2_)
    : it1(it1_),it2(it2_),cacheInitialized(false){}
  ~PairIt() { emptyCache(); }
  reference operator*() const {
    initCache();
    return *getValuePtr();
  }
  pointer operator->() const {
    initCache();
    return getValuePtr();
  }
private:
  mutable Iter1 it1;
  mutable Iter2 it2;
  mutable utils::RawMem<PairDefs::DeRefType>
                                          dataCache;
  mutable bool cacheInitialized;
};
```

Listing 6: A basic implementation of `PairIt`

Moving Forward

For the cases where both the underlying iterators are at least Forward Iterators, we need to meet additional requirements, for `PairIt` to also work as a Forward Iterator. These are given by section 24.1.3 and table 74 of the Standard, and are actually mostly semantic constraints, rather than operational constraints,

so these are implemented automatically if the underlying iterators are themselves Forward Iterators. The only additional operation required is that `PairIt` must be *Default-constructible*, which is trivially implemented.

If our underlying iterators are at least Bidirectional Iterators, we should also implement the pre- and post-decrement operators to maintain that level, as detailed in section 24.1.4 and table 75 of the Standard. As for pre- and post-increment, these can be implemented merely by forwarding to the underlying iterators:

```
PairIt& operator—() {
  —it1;
  —it2;
  return *this;
}
```

Dereferencing at Random

If our underlying iterators are both Random Access Iterators, then we have a whole swathe of additional requirements to support, as detailed in section 24.1.5 and table 76 of the Standard. These are:

- The arithmetic operators `+` and `–`,
- The arithmetic assignment operators `+=` and `-=`,
- The comparison operators `<`, `>`, `<=` and `>=`, and
- The subscripting operator `[]`.

The arithmetic operators are trivial – just forward to the underlying iterators. The comparison operators require a bit more thought – what do we do if the first iterator is less than its partner, but the second isn't? – but the problems can be defined out of existence; iterators can only be compared if they are in the same range. This implies that one is reachable from the other. If the first and second underlying iterators don't give the same results when compared against their partners in the `PairIt` we are comparing against, then this can't be the case, and the issue can be avoided – in fact, we could get away with just comparing the first iterator in each pair, so the comparison operators are also trivial. The subscripting operator is easy, too – `it[n]` is just syntactic shorthand for `*(it+n)`, so we can implement it that way.

However, a bit more thought reveals that doing things the "simple" way requires that all these operations are either member functions or `friend`s of `PairIt`, yet some could be implemented in terms of others. For example, the idiomatic way of implementing `+` is to use `+=` on a copy, as follows:

```
template<typename I1,typename I2>
PairIt<I1,I2> operator+(PairIt<I1,I2> temp,
                        std::ptrdiff_t n) {
  temp+=n;
  return temp;
}
```

The same applies to the comparison operators – all the others can be implemented in terms of `operator<`. In fact, `operator<` itself can then be implemented in terms of subtraction, since these are Random Access Iterators, so the list of `friend`s and member functions is now down to:

- `operator+=`,
- `operator-=`,
- `operator-` where both operands are iterators, and
- `operator[]`, which is required to be a member function.

Helper functions

Sometimes we don't want to have to specify the precise type of our iterators explicitly, because we're creating a temporary object to pass to an algorithm, and doing so requires excessive typing, if it is possible at all. For this reason, we also provide a helper template function `makePairIterator` that takes two iterators as parameters, and returns a `PairIt` containing them. This simple function and its use is shown in listing 7.

```
template<typename Iter1,typename Iter2>
PairIt<Iter1,Iter2> makePairIterator(
                        Iter1 it1,Iter2 it2) {
  return PairIt<Iter1,Iter2>(it1,it2);
}

// use of makePairIterator
std::vector<int> src1;
std::deque<double> src2;
bool myPairComparisonFunc(
                const std::pair<int,double>&,
                const std::pair<int,double>&);
std::sort(
 makePairIterator(src1.begin(),src2.begin()),
 makePairIterator(src1.end(),src2.end()),
   myPairComparisonFunc);
```

Listing 7: The `makePairIterator` helper function

Of course, as written, this will copy the elements of the containers to do the comparison, as the `value_type` of the pair iterators is a custom pair type, as described above, so a temporary `std::pair` has to be constructed. It is therefore more efficient to write functor class with a template function call operator:

```
struct MyPairComparisonFunctor {
  template<typename PairType>
  bool operator()(const PairType&,
                  const PairType&) const;
};
```

You could, of course, write a comparison function to take the precise custom pair type in question, but this varies depending on the iterators, so is not as straightforward as it may seem.

Conclusion

Implementing an iterator adapter to treat a pair of sequences as a sequence of pairs is not a trivial task, though some of the individual parts are; the biggest headache is deciding the `value_type` and the return type for the dereference operator.

However, it provides a genuinely useful service, and in combination with other iterator adapters and function objects, can be used to access data in intuitive ways, however it is stored.

Anthony Williams
anthony_w@onetel.net.uk

References

[1] ISO/IEC 14882, Programming Languages – C++. International Standard, September 1998.

[2] Andrei Alexandrescu. Generic<Programming>: Discriminated unions (II). *C/C++ User's Journal*, 20(6), June 2002. Available online at http://www.cuj.com/experts/2006/alexandr.htm.