overload|76

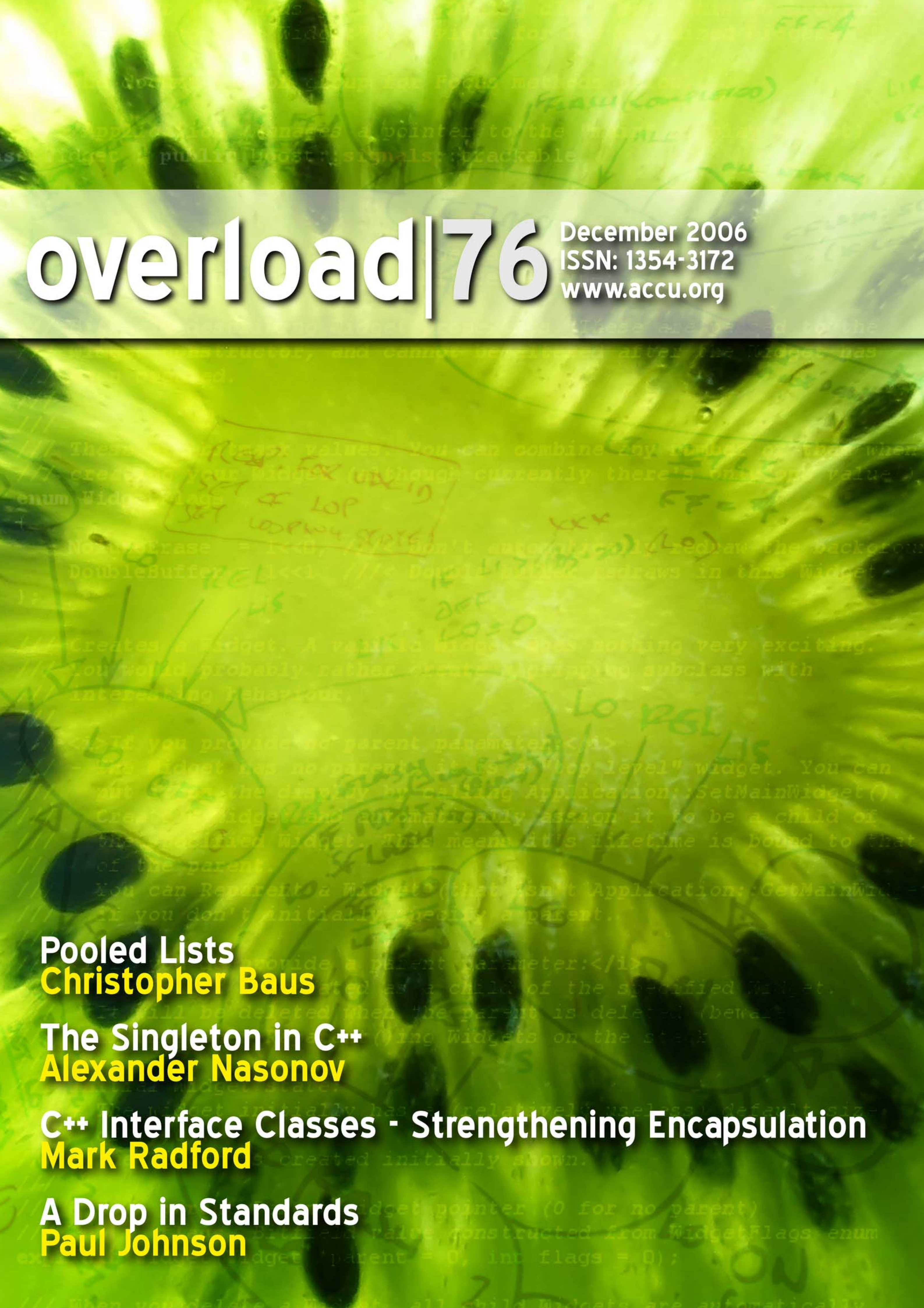
December 2006
ISSN: 1354-3172
www.accu.org

Pooled Lists
Christopher Baus

The Singleton in C++
Alexander Nasonov

C++ Interface Classes - Strengthening Encapsulation
Mark Radford

A Drop in Standards
Paul Johnson



OVERLOAD 76

December 2006

ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Contributing editors

Mark Radford
mark@twonline.co.uk

Paul Johnson
paul@all-the-johnsons.co.uk

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Alistair McDonald
alistair@inrevo.com

Anthony Williams
anthony.ajw@gmail.com

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Ric Parkin
ric.parkin@ntlworld.com

Roger Orr
rogero@howzatt.demon.co.uk

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 77 should be submitted to the editor by 1st January 2007 and for Overload 78 by 1st March 2007.

Overload is a publication of the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

4 Pooled Lists
Christopher Baus looks at pooled memory allocation.

8 The Singleton in C++ – A force for good?
Alexander Nasonov addresses some problems that arise when using Singleton in C++.

11 C++ Interface Classes: Strengthening Encapsulation
Mark Radford looks at how the separation of interface and implementation in C++ helps to strengthen encapsulation.

13 A Drop in Standards?
Paul Johnson asks why recent graduates seem less capable than their predecessors.

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

How Overload Comes to You

You look forward to Overload arriving in the mail every couple of months. This time it came very close to not arriving at all.

What we did well

At the moment it seems like it was a long time ago and in a galaxy far, far away that lots of material was submitted to Overload by enthusiastic authors, an enthusiastic team of volunteer advisors reviewed it all, the editor swiftly passed their comments back to the authors who produced a new, much improved draft of the article which then was seen to be good by all and put in the magazine for your reading pleasure.

OK, that was a “typical golden age” story – it never really went quite that smoothly, but it has often been pretty close. Articles sometimes had to go around the loop several times before being deemed good enough. Sometimes there has been a shortfall in the material submitted – but a couple of the advisors (Phil Bass and Mark Radford) were pretty good at producing material at the last minute. Sometimes all the advisors had other commitments and failed to review an article or two – but this was rare and the editor found time to cover the gaps. Sometimes the editor was busy, but then the contributing editor (or one of the advisors) found time to do a little extra (like passing review comments back to the author directly). And of course, there were occasional articles that required special handling (most often reworking the English of an author who was not writing in his or her native tongue) and someone was always there to do this.

What we could have done better

But time goes by, and the time people are able to give varies. (Have I mentioned we are all volunteers, working on Overload because we think it is worth doing?) Some of the advisors have taken up new responsibilities in their non-ACCU lives and cannot commit the time they used to, others have changed their focus to C Vu as a result of recent changes there. I first mentioned the need for you (anyone that agrees that Overload is worthwhile) to get involved a couple of editorials ago (Overload 74):

> A second reason is that recently several of the current advisors have taken on new commitments in their lives. (Some of these commitments have to do with ACCU, some are personal, but all have the effect that the editorial team has reduced capacity and would welcome some new blood.)

This hasn't, however, addressed the problem: the only volunteer to join the Overload team this year (Tim Penhey) has since moved on to edit C Vu. In the run up to the current issue I too have found that recent commitments have taken more of my time than I anticipated.

The result of these changes was that when articles were submitted and I circulated them to the advisors nothing happened. For a while I thought that I would soon have time to review them myself, but as time went on and this time continued to recede I eventually realised that this wasn't going to happen. What is more, with only a couple of weeks to go I also discovered that nowhere near enough material had been submitted.

At this point I called for volunteers to help on ACCU-general. There were a number of people who volunteered their time to review the two articles I had (thanks are due to Paul Thomas, Ric Parkin, Simon Sebright and Roger Orr). In addition there was a very welcome offer from Paul Johnson (who recently gave up the editorship of C Vu) to take over all the editing of the articles and also to write a couple of articles himself.

Help was also forthcoming from our reliable Contributing Editor – Mark Radford – who, in addition to any help he has provided Paul in handling the articles submitted, once again performed his magic to produce an article when the shortage became apparent.

That left me with the editorial to write – and has made this issue of Overload possible. It also means that for the first time in years I'll be able to join you in reading the articles in Overload for the first time when it arrives through the post.

Please note however, that while it is good to have material to fill the pages, writing articles isn’t the role of the editorial team. Earlier this year I considered that the biggest problem was the shortage of articles being submitted. In Overload 72 Editorial (April) I wrote:

> The team of advisors listed at the front of Overload are not here to write the magazine for you, they are here to help with the selection of the best stories to publish and to work with the authors to ensure that their stories are told well. And after the advisor's efforts to fill the pages of Overload this time I'm going to enforce a rest. *I will not be accepting any articles from the advisors for the next issue of Overload.* (If the next issue consists only of an editorial then you'll know why.)

After the efforts by Paul and Mark for this issue I will be giving serious consideration to reinstating this ban for the next issue.

What we’d like to try

I know that my situation for the next issue will not be much improved and that finding time will remain difficult. But messages of support from a number of you have addressed any thoughts that I had about it being time for me to step down. I’ve discussed this with the Publications Officer and I will remain with overall responsibility and ensure that the character of

Alan Griffiths is an independent software developer who has been using “Agile Methods” since before they were called “Agile”, has been using C++ since before there was a standard, has been using Java since before it went server-side and is still interested in learning new stuff. His homepage is http://www.octopull.demon.co.uk and he can be contacted at overload@accu.org

Overload is preserved. However, if Overload is to carry on, it cannot be dependent on vast amounts of my time.

The current team of volunteer advisors is in a similar position to the one that I find myself in (not being able to contribute the time we'd like), so while you may see that familiar names cease to appear I have hopes that some of you, perhaps the volunteers who have helped out by reviewing (and editing) articles for the last couple of issues, will volunteer again – but on a more permanent basis.

Thankfully, the first volunteer to was not hard to find: Paul Johnson is keen to do "whatever he can" – so beginning with this issue Overload now has two "Contributing Editors". No-one (not even they themselves) knows exactly what a "Contributing Editor" is supposed to do in the context of Overload, but we will sort that out amongst ourselves. Whatever we decide, it includes soliciting articles, parts of the editorial process, and sometimes contributing articles.

What you'd like to do

I know that you have heard this before, but it remains true. Overload is your magazine and it needs your support.

Also, one volunteer does not make a team. If you appreciate the writing in Overload you can help make it happen. You don't need to be an expert on the English language – much of the reviewing is "this point needs explaining because..." or "this explanation doesn't make sense because..." and, of course, deciding whether the article covers material that belongs in Overload. There is no limit to the number of helpers we can use – the more people that contribute to this work, the less each has to do.

Let me repeat the invitation from Overload 74:

> If you have a clear idea of what makes a good article about software development and would like to help authors polish their writing then now is the time to get involved – while the old hands are here to "show the ropes" (and before some of them find their new commitments push contributing to Overload every issue out of their lives). It isn't hard – and the authors do appreciate the work done reviewing their articles.

It is not just reviewing that is needed though. Last issue however, we lacked contributions and were below our minimum length for the first time since I took over as editor, and this issue again we have had problems. (I guess that some of this can be laid at my door since I've not been as active as usual in asking people to write up the things they have been telling me about.) This changes quickly: in Overload 74 things the supply of articles seemed to be good and I wrote:

> Once again I'm pleased with the response from authors – for the second issue running the advisors have been able to concentrate on providing assistance to the authors and to the editor and have not found it necessary to write articles to reach our minimum length. As you can see, we have comfortably achieved that again. Thanks to everyone that submitted an article! (I know some were redirected to C Vu, but the effort is appreciated.) I trust that those of you who have been thinking of writing will be inspired by this example and do so next time.

Everyone in ACCU has a unique perspective and experience and knows things that others do not. Writing about them not only helps those that lack your knowledge, but also helps you understand and value what you do know.

> It can be hard to identify something worth writing about – after all we all tend to think things we understand are obvious. But one doesn't have to interact with many people to discover that there are "obvious" things that other people don't understand. You don't want to keep explaining, so write it up and send it in – then, instead of going over the same old ground, you can say "here's a good article about it" (and wait for them to notice the author's name).

Overload on the web

Since Allan Kelly wrote about putting Overload onto the ACCU website a number of you have written to me with enthusiasm for the idea – a task that I've taken on in my capacity as Publicity Officer. As you may have guessed from what I've written about this issue of Overload, I've yet to make time to progress this (and, to be honest, some other things the Publicity Officer should be doing). I still think making Overload publicly available is a good idea (and there are no technical issues to overcome) but it has to take its place behind family and social commitments, work, and editing Overload. I'm sorry, but this isn't likely to happen before the conference – not much to show for a year in the post. Naturally, if someone would like to help out then I might be able to report progress at the AGM.

In Conclusion

I cannot claim any credit for this issue of Overload – it is all the work of others, principally Paul Johnson. But, while I cannot claim credit, I have to take the blame for anything that is wrong as I allowed a very difficult situation to develop and then left others to sort it out. I will be feeling rather nervous when I open this issue of Overload.

Pooled Lists

Christopher Baus explains the advantages of using a pooled memory allocation strategy for high performance applications.

Motivation

By default, the C++ `list` container allocates memory with the global operator `new` when elements are added to the list, and frees memory with the global operator `delete` when elements are removed. While this provides a convenient interface for most applications, high performance and long lived applications could benefit from a pooled memory allocation strategy, where all memory used by the list is preallocated at list creation time.

Most default implementations of the operators `new` and `delete` resolve to a direct call to the underlying system heap. This has the following drawbacks:

- List insertion operations may throw indeterminately.
- System heap operations create contention between otherwise unrelated threads.
- System heap operations require an often expensive system call.
- Frequent allocations and deallocations of small objects can result in fragmentation, and inefficient use of memory.

Kevlin Henney [Henney] says this regarding collection memory management:

> If you are serious about managing the allocation of a container, then get serious and manage it: Write your own container type.

The standard `list` allocator interface is the logical starting point for implementing a pooled list container, but as Pablo Halpern noted in his proposal to the C++ standards committee [Halpern, 2005], there some inconsistencies in the standard. The handling of allocators which compare unequal is currently under specified in C++, as is noted in issue 431 of the C++ Standard Library Active Issues List [C++ Active Issues]. While Howard E. Hinnant [Hinnant] provides guidance for a solution, it currently is not part of the standard library. Instances of pool allocators of the same type compare unequally when they allocate from different memory pools, hence it isn't possible to implement satisfactory pooled allocators given the current standard. This leaves Kevlin's suggestion to implement a custom container.

This article investigates implementing a pooled list container using the C++ standard `list` as the underlying data store. The objective is to provide pooling whilst delegating most functionality to the standard library.

The Interface

The `pooled_list` class provides a simple and familiar interface to C++ developers. With a few exceptions, `pooled_list`s behave similarly to standard `list`s. When using `pooled_list`, the user must first create a `pool` as follows:

```
size_t pool_size = 256;
pooled_list<int>::pool list_pool(pool_size);
```

Christopher Baus can be contacted at christopher@baus.net

The `pool` allocates memory for both the element data and the `list`'s internal structure. No additional allocations will be made after the `list_pool` has been constructed.

One or more `list`s are attached to the pool:

```
pooled_list<int> my_list(list_pool);
```

The user can then operate on the `list` just as a standard `list`. For instance:

```
my_list.push_back(5);
my_list.pop_front();
```

Exceptions and error handling

Using the standard `list` container `insert` operations can fail unpredictably. If memory is not available to complete the operation they will throw `std::bad_alloc`. In comparison, the `pooled_list` container provides deterministic failure. `Insert` operations only throw on a `pooled_list` with a depleted `pool`. Since the `pooled_list` takes a reference to a `pool`, pools must have a lifetime greater than all the lists from which they are created. If a `pool` is destroyed before the lists which are created from it, the behaviour is undefined.

Since memory is allocated from the C++ runtime heap to create `pool`s, `pool` construction may throw `std::bad_alloc`.

The semantics of operations which act upon multiple lists are affected by pooling. The operations including the overloading of `merge()` and `splice()` require that `pooled_list`s are allocated from the same pool. The condition is checked with an `assert` and violation results in undefined behaviour. These semantics are borrowed directly from Howard E. Hinnant's recommendation for swapping containers with unequal allocators [Hinnant].

To provide the strong exception guarantee, the assignment operator is implemented by creating a temporary copy of the right hand list, and then swapping the temporary with the left hand list. The pools of the left hand side and right hand side are also swapped (again as recommended by Hinnant). This requires that the right hand side list's pool contains at least as many free elements as are used by the right hand side list, even though fewer elements maybe required upon completion of the operation.

In explicit terms, the assignment operation involves three lists: the list being assigned from (`right`), the initial list being assigned to (`left`), and the final state of the list being assigned to (`left'`). Following the operation both `left'` and `right` will use the same `pool`, even if `left` has used a different `pool`. All elements will be allocated from the `pool` used by `right`. The assignment operator, again, to provide the strong exception guarantee, creates `left'` (as a copy of `right`) before `left` is destroyed. For the operation to succeed, the `pool` used by `right` must contain at least the number of elements in `right`. When `right` contains more elements than `left`, and uses the same `pool` as `left`, it is not sufficient for the `pool` used by `right` to contain `right` minus `left`

Heap access causes **contention** between **threads** which would otherwise be **unrelated**

number of elements, even though that number of elements will be used (in conjunction by **left'** and **right**) after the operation completes.

Concurrency

The global heap is a resource which is shared among all threads, and access to the heap is often serialized by the underlying operating system to prevent corruption. Microsoft [Microsoft] describes the problem and offers the following with respect to Windows:

> A single global lock protects the heap against multi threaded usage... This lock is essential to protecting the heap data structures from random access across multiple threads. This lock can have an adverse impact on performance when heap operations are too frequent.
>
> In all the server systems (such as IIS, MSProxy, DatabaseStacks, Network servers, Exchange, and others), the heap lock is a BIG bottleneck. The larger the number of processors, the worse the contention.

Heap access causes contention between threads which would otherwise be unrelated. The standard **lists insert()** and **erase()** operations require heap access, and hence can cause contention. Consider the following example: a program contains two threads in which each thread creates a **list** which is only accessed by that thread. While it is safe for either thread to **insert** or **erase** elements from its respective list, access to the heap is serialized by those operations. The result is reduced performance even though there is no data shared between the threads. A **pooled_list** does not access the heap directly after the pool is created, so there is no contention between lists.

Like the STL, the **pooled_list** class makes few guarantees about thread safety. There is no thread synchronization in the **pooled_list** class, hence multiple threads can not concurrently **insert** or **erase** elements in lists which share the same pool. This requires that users externally synchronize list operations when one or more lists which use the same pool are accessed from multiple threads. If **pooled_list**s do not share pools, there is no need to synchronize access to **pooled_list**s.

The implementation

Linked lists impose storage overhead beyond contiguous arrays due to their node structure. Typically a **list** node holds pointers to the next node in the **list** and the previous node in the **list**. A **list** node could be defined as shown in Listing 1.

List allocation requires not only memory for the element data, but the **list** node structure as well. This is why the specification requires that allocators provide a **rebind()** [Rebind] implementation, which allows them to allocate node types. Providing allocators that only return elements of type **T** is insufficient. In classical C linked list implementations, such as the one provided by the Linux kernel [LinkedList], the user provides user allocated nodes at insertion time. With the C++ standard **list**, nodes are abstracted from the user.

```
template<typename T>
struct node {
  node(T initial_value):element_(initial_value){}
  node* next_;
  node* previous_;
  T element_;
}
```

Listing 1

The goal of the **pooled_list** implementation was to preallocate not only element data, but the node data, and hence all the memory used by the **list**. This can be achieved by implementing the **list** from scratch – creating new node types and implementing all the **list** operations. For the sake of expediency and correctness, I chose a hybrid approach which uses the standard **list** itself as the underlying structure for the **pooled_list**.

The **pooled_list** implementation uses two standard lists: the free and active lists. When created, the free list contains *n* number of elements and the active list is empty. When elements are inserted, the required nodes are moved from the free list to the active list. When elements are erased, the released nodes are moved from the active list to the free list. This is accomplished with the standard list's **splice()** operation, which allows lists to be combined in constant time without memory allocation. While this is a rudimentary implementation, it does offer some challenges in correctly providing value semantics.

A naive implementation

The structure of my first attempted implementation was similar to Listing 2.

While it will work for built-in types and PODs [POD], it results in the constructors for objects being called when the free **list** is created, not when elements are inserted. Likewise, destructors for the elements are called when the free **list** is deleted and not when elements are erased. To solve this problem the underlying lists must contain unstructured data, which leads to the next attempt, Listing 3.

By using standard lists of arrays of **char** the construction/destruction of elements can be constrained to **insert** and **erase** operations, properly implementing value semantics. Unfortunately this leads to some subtle

```
template<typename T>
class pooled_list
{
  ...
private:
std::list<T> free_list_;
std::list<T> active_list_;
};
```

Listing 2

destructors for the elements are called when the free list is deleted and not when elements are erased

alignment problems. Internally to the standard **`list`**, the **`char array`** is a member of the node, as shown in the node code example in Listing 3, and C++ does not guarantee all types T will be properly aligned. Stephen Cleary [Cleary] provides further discussion of alignment in his documentation for the boost **`pool`** library.

Lists of type **`std::list<char*>`** are used by the final implementation which is based on the following from Cleary's discussion:

> Any block of memory allocated as an array of characters through operator **`new[]`** (hereafter referred to as the block) is properly aligned for any object of that size or smaller

For an example, see Listing 4.

The final implementation differs slightly in that the **`free_list_`** is moved to a separate pool class which allows it to be shared by multiple **`pooled_list`**s. The alignment workaround does impose one pointer's worth of space overhead per element for each node used in free and active lists. This could be avoided by developing a custom **`list`** rather than using the standard **`list`** as a base implementation.

List iterators

Since the underlying list is of type **`std::list<char*>`** and not **`std::list<T>`**, iterators to the active list can not be used directly. The data must be dereferenced and cast to the type **`T`**. The boost iterator library is employed to perform the operation. This greatly simplifies the implementation at the cost of a dependency on the boost iterator library (see Listing 5).

Potential enhancements

As noted in the threading section, multiple threads can not concurrently **`insert`** or **`erase`** elements in lists which share the same **`pool`**. I chose to not impose the overhead of thread synchronization by default. I do not recommend sharing **`pool`**s across threads, but this could be supported by adding a synchronization policy to the **`pool`** with no additional overhead for the default case.

Element data is allocated by the pool using **`new[]`**. This might not be sufficient for all use cases (for instance if the user wants to allocate elements from an arena or global **`pool`**). This could be also be addressed

```
template<typename T>
class pooled_list
{
...
      iterator insert(iterator pos, const T& value)
      {
         if(!free_list_->empty()){
           // use inplace new on the raw data
           // and splice from the free list to
           // the active list
           ...
        }
         else{
           throw std::bad_alloc();
         }
         return --pos;
       }
...
private:

std::list<char[sizeof(T)]> free_list_;
std::list<char[sizeof(T)]> active_list_;
};
```

Listing 3

```
template<typename T>
class pooled_list
{
...
    pooled_list():free_pool_(pool_size, 0)
    {
      std::list<char*>::iterator cur;
      std::list<char*>::iterator end
        = free_list_.end();

      for(cur = free_list_.begin(); cur != end;
         ++cur){
         *cur = new char[sizeof(T)];
      }
    }
...
     iterator insert(iterator pos, const T& value)
      {
         if(!free_list_->empty()){
           // use inplace new on the raw data,
           // and splice from the free list to the
           // active list
           ...
         }
         else{
           throw std::bad_alloc();
         }
         return --pos;
      }
...

private:

std::list<char*> free_list_;
std::list<char*> active_list_;
};
```

Listing 4

if you are serious about memory management, you must rely on custom containers

```
template<typename T>
class pooled_list
{
...

// Functor to convert iterator to underlying data type to type T.
class reinterpret_raw_data_ptr
{
public:
  typedef T& result_type;

  T& operator()(raw_data_type& raw_data) const
  {
    return *reinterpret_cast<T*>(raw_data);
  }
};

typedef char* raw_data_type;
typedef std::list<raw_data_type> raw_data_ptr_list;
typedef typename std::list<raw_data_type>::iterator raw_data_ptr_list_it;
typedef boost::transform_iterator<reinterpret_raw_data_ptr, typename raw_data_ptr_list::iterator,
   T&, T > iterator;
...
};
```

Listing 5

by adding an allocation strategy to the `pool`. It should be noted that because the standard `list` is used as the underlying data structure, it would be difficult to change the allocation strategy of the node structures. Providing an alternate strategy to allocate `list` nodes would require a reimplementation of the `list` structure.

Conclusion

The C++ library specification currently requires developing custom containers to implement allocators with shared state. While it might be possible to develop `pool` allocators which work with existing standard library implementations, it is not be possible to guarantee that the `pool` allocators would work correctly across library implementations. As Kevlin says, if you are serious about memory management, you must rely on custom containers. Pool allocation is a proven strategy for many long lived, concurrent applications with high reliability and performance requirements such as network servers. The provided implementation provides a simple solution which successfully leverages the standard library for most operations. The result is a pooled list container that is compatible with any standard compliant standard library implementation. ■

Acknowledgements

Thanks to C++ guru Thomas Becker for discussion on data alignment and help with `const_iterator` syntax.

References

[C++ Active Issues] Issue 431, C++ Standard Library Active Issues List : http://www.open-std.org/JTC1/SC22/WG21/docs/lwg-active.html#431

[Cleary] Stephen Cleary : http://www.boost.org/libs/pool/doc/implementation/alignment.html

[Halpern, 2005] Proposal to the C++ standards committee : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1850.pdf

[Henney] Dr Dobb's Journal : http://www.ddj.com/dept/cpp/184403779

[Hinnant] Hinnant, H.E. : http://www.open-std.org/jtc1/wg21/docs/papers/2004/n1599.html

[LinkedList] Classical C linked list : http://isis.poly.edu/kulesh/stuff/src/klist/

[Microsoft] Microsoft : http://msdn2.microsoft.com/en-us/library/ms810466.aspx

[POD] PODs : http://www.fnal.gov/docs/working-groups/fpcltf/Pkg/ISOcxx/doc/POD.html

[Rebind] Rebind documentation : http://msdn2.microsoft.com/en-us/library/5fk3e8ek.aspx

The Singleton in C++ – A force for good?

Alexander Nasonov addresses some problems that arise when using Singleton in C++.

According to [Alexandrescu], the Singleton pattern has a very simple description but can have complex implementations. Indeed, [GoF] states that this pattern limits an object to one instance in the entire application, then they give an example. It doesn't seem very complex but it isn't that simple (see Listing 1).

```
// Declaration
class Singleton {
public:
  static Singleton* Instance();
protected:
  Singleton();
private:
  static Singleton* _instance;
};

// Definition
Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance() {
  if(_instance == 0) {
      _instance = new Singleton();
  }
        return _instance;
    }
```

Listing 1

Actually, the _instance doesn't always point to one object. Before a first call to **Singleton::Instance**, there is no **Singleton** object. And this they call **Singleton**???

You may argue that this mechanism is completely transparent to a user but that's not quite correct.

Systems with tight time-constraints would suffer from the fact that calling this accessor function may take as long as the longest execution path in Singleton constructor. This time should be added to every function that calls **Singleton::Instance** even though this long path is executed only once[1].

There is another visible effect in multi-threaded programs caused by unprotected concurrent access to **Singleton::Instance**. Refer to [DCLocking] for more details.

What's more, the object is never deleted!

Why did the authors select this approach if they could just wrap a global variable to ensure one instance of a Singleton, as shown in Listing 2?

```
// Declaration
class Singleton : private boost::noncopyable {

public:
  static Singleton instance;

private:
  Singleton();
  ~Singleton();
};

// Definition
Singleton Singleton::instance;
```

Listing 2

The authors should have had a very strong reason to prefer the first solution. Unfortunately, the book doesn't explain it well.

Modern C++ Design has a special chapter about singletons. It answers what's wrong with the second code snippet. The order of initialisation of objects from different translation units is unspecified. It's possible to access an uninitialised object during the initialisation of another object as demonstrated by this code:

```
// Somewhere in other TU:
int undefinedBehavior =
Singleton::instance.getAccess();
```

The [GoF] version fixes this problem because the Singleton instance is automatically created when **Singleton::Instance** is called for the first time:

```
// Somewhere in other TU:
int goodBehavior
   = Singleton::Instance()->getAccess();
```

The question is, does this really make it better?

Alexander Nasonov has been programming in C++ for over 8 years. Recently his interests expanded to scripting languages, web technologies and project management. A few months ago he became an occasional blogger at http://nasonov.blogspot.com. Alexander can be contacted at alexander.nasonov@gmail.com.

First of all, it doesn't completely define an order of initialisation. Although dependencies at initialisation time are tracked correctly, they define only a partial ordering. It's worthwhile to note that these dependencies are built at run-time, that is, they may vary from run to run. Often these dependencies are not clear to developers (they tried to manage them automatically, after all!).

You may wonder why track the dependencies if all singletons are initialised correctly. Don't forget that the program should stop its execution correctly as well. Singletons not only depend on other singletons at initialisation time but also when they are destroyed.

One fundamental principle of C++ is "first initialised, last destroyed". It helps to maintain a list of objects that can be used in a destructor of some

1. It can be proved that some functions are never the first to call Singleton::Instance and discount them, but this distinction would complicate the matter even further.

Memory zero-initialised at the static initialisation phase may be accessed even though an object is not yet constructed

global object. That list contains only global objects constructed earlier than this object.

The [GoF] solution doesn't rely on this principle of C++ because objects are created on the heap. They don't explain how to destroy these objects, though. You can read *Modern C++ Design* for a good explanation of managing singleton lifetimes.

All these problems keep me away from using this code unless a framework as good as the boost libraries appears on my radar screen. Until then I stick to global variables with uniqueness guarantees (if affordable) and rely on compilers that allow me to specify an order for non-local objects initialisation.

I found the following principle very useful:

- Minimise global variables to a bare minimum
- Minimise dynamic initialisation of global variables

Static initialisation doesn't have a dependency problem because all objects are initialised with constant expressions. Compare:

```
pthread_mutex_t g_giant_mutex
   = PTHREAD_MUTEX_INITIALIZER;
```

with

```
boost::mutex g_giant_mutex;
// User-defined constructor
```

The former guarantees initialisation before dynamic initialisation takes place. The latter does not. Please note that private constructors required to ensure uniqueness enforce dynamic initialisation.

Protect yourself from access to uninitialised objects

Memory zero-initialised at the static initialisation phase may be accessed even though an object is not yet constructed. Sometimes it crashes the program but it can also yield surprising results.

For example:

```
// TU 1
money dollar(1.0, "USD");

// TU 2
double valueOfDollar = dollar.getValue();
```

If initialisation starts with `valueOfDollar`, `dollar.getValue()` may return `0.0` instead of `1.0`.

To protect yourself from this kind of access, you can use a boolean global variable constructed during the static initialisation phase. This variable is true only if an object it controls is alive. See Listing 3.

Control dependencies in constructors of global variables

My preferred method is not to use any global variables directly. It's best demonstrated by example:

```
// Declaration
class Singleton : private boost::noncopyable {
public:
  static Singleton* instance();
private:
  Singleton();
  ~Singleton();

  static bool g_initialised;
  static Singleton g_instance;
};

// Definition
bool Singleton::g_initialised;
  // static  initialisation
Singleton Singleton::g_instance;
  // dynamic initialisation
Singleton::Singleton()
{
  g_initialised = true;
}

Singleton::~Singleton()
{
  g_initialised = false;
}

Singleton* Singleton::instance()
{
  return g_initialised ? &g_instance : 0;
}
```

Listing 3

```
Singleton Singleton::g_instance(Logger::instance(),
                              Factory::instance());
```

All Singleton's dependencies are visible at the point of definition. They are easily accessible inside the Singleton constructor as actual arguments. You don't need to access them through the `instance` functions.

Don't introduce more dependencies in a destructor

In general, you should use only dependencies defined at construction time. This ensures that those objects are not destroyed (remember the first constructed, last destroyed rule).

However, there is one exception; some singletons such as `Logger` should be accessible from all other singletons. You can implement a technique similar to `iostream` objects initialisation or use non-virtual functions that don't access member variables if `Logger` is already destroyed[2] (see Listing 4).

2. Strictly speaking, calling any function of a dead object is bad. In practice, calling non-virtual function is safe if it doesn't access memory pointed to by this pointer.

```
void Logger::log(char const* message)
{
    if(!g_initialised)
        std::cerr << "[Logger is destroyed] "
           << message << '\n';
    else
        // Put your cool logger code here
}
```

Listing 4

Build your program in more then one way

Nowadays, people rarely link statically. However, many loaders manage dependencies among shared libraries automatically and hide potential problems. It’s harder to setup a static build but it’s usually worth it.

Conclusion

I recently came across an interesting blog entry of Mark Dominus. He says: Patterns are signs of weakness in programming languages. I don’t completely agree with this statement because even if you put all patterns in a language you’ll soon discover other patterns. But I agree with Mark that the Singleton pattern is a sign of weakness of the C++ language.

Whilst there are many ways to implement a singleton, the method provided in Listing 5 fulfils the criteria presented in this article. ■

References

[Alexandrescu] Andrei Alexandrescu. *Modern C++ Design*

[DCLocking] Scott Meyers and Andrei Alexandrescu. *C++ and the Perils of Double-Checked Locking*, Doctor Dobb’s Journal, 2004. http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

[Dominus] Mark Dominus. *Design patterns of 1972*. http://www.plover.com/blog/prog/design-patterns.html

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

```
// Singleton.hpp
#ifndef FILE_Singleton_hpp_INCLUDED
#define FILE_Singleton_hpp_INCLUDED
#include <boost/noncopyable.hpp>

class Singleton : private boost::noncopyable
{
  public:
    static Singleton* instance();
    void print(char const* str);
  private:
    Singleton();
    ~Singleton();
    static bool g_initialised;
    // static  initialisation
    static Singleton g_instance;
    // dynamic initialisation
    char m_prefix[32]; // to crash the program
};

#endif // FILE_Singleton_hpp_INCLUDED

// Singleton.cpp
#include "Singleton.hpp"
#include <ostream>
#include <iostream>
#include <cstring>

bool Singleton::g_initialised;
   // static  initialisation
Singleton Singleton::g_instance;
   // dynamic initialisation

Singleton::Singleton()
{
    g_initialised = true;
    std::strcpy(m_prefix, ">>> ");
}

Singleton::~Singleton()
{
    g_initialised = false;
}

Singleton* Singleton::instance()
{
    return g_initialised ? &g_instance : 0;
}

void Singleton::print(char const* str)
{
    std::cout << m_prefix << str;
}

// main.cpp
#include "Singleton.hpp"

struct X
{
    X() { Singleton::instance()->print("X\n"); }
   ~X() { Singleton::instance()->print("~X\n");
}
} x;

int main()
{
    Singleton* p = Singleton::instance();
    p->print("Hello, World!\n");
}
```

Listing 5

C++ Interface Classes – Strengthening Encapsulation

Mark looks at the separation of interface and implementation in C++, and how the separation helps to strengthen encapsulation.

Separating a class's interface from its implementation is fundamental to good quality object oriented software design/programming. However C++ (when compared to, say, Java) provides no indigenous mechanism for expressing such a separation. Therefore, a number of idioms supporting this separation have evolved in C++ practice, and was the subject of an article in Overload 66 I co-authored with Alan Griffiths [Radford-Griffiths]. The idioms covered in that article do not just cover object oriented programming, but other approaches (such as value based programming) as well.

For object oriented programming, the principle mechanism of separation is the Interface Class. An Interface Class contains only a virtual destructor and pure virtual functions, thus providing a construct analogous to the interface constructs of other languages (e.g. Java). I discussed Interface Classes in *C++ Interface Classes: An Introduction* [Radford:1] and explored an example of their application and usefulness in *C++ Interface Classes: Noise Reduction* [Radford:2] (published in Overloads 62 and 68 respectively).

In this article I would like to discuss the role played by Interface Classes in strengthening encapsulation. In doing so, I hope to extend the discussion to use unit testing as an example of how Interface Classes underpin encapsulation (while taking a swipe at the Singleton design pattern [Gamma et al, 2005] in the process).

A motivating example

Consider a GUI based drawing program, where the user manipulates shapes – such as lines and circular/elliptical arcs – within a window. It is an old chestnut that serves well as a motivating example.

First, we have a **class** hierarchy for the shapes. This will be headed up by an Interface Class called (guess what) **shape** – see Listing 1.

Second, the shapes are stored in a drawing, let's represent this programmatically with an Interface Class called **drawing** – see Listing 2.

Please take note of the **drawing::save(repository& r)** member function – specifically, its **repository&** parameter. This means we need a definition for repository, as shown in Listing 3.

```
class shape
{
public:
  virtual ~shape();

  virtual void move_x(distance x) = 0;
  virtual void move_y(distance y) = 0;
  virtual void rotate(angle rotation) = 0;

...
}
```

Listing 1

```
class drawing
{
public:
    virtual ~drawing();
    virtual void save(repository& r) const = 0;
...
};
```

Listing 2

```
class repository
{
public:
  virtual ~repository();
  virtual void save(const shape* s) = 0;
...
};
```

Listing 3

The **repository** class is a programmatic representation of the repository where drawings are kept when not in memory, i.e. the storage (e.g. a database).

Having introduced the participants in this example, it is time to move on. Before I do though, there are a couple of things I would like to point out:

1. I have introduced the participants only as Interface Classes, without any implementation classes. This example does not require all of them to have implementations shown. Therefore, implementations will be introduced as (and if) needed.
2. In reality, the **shape** class would need member functions for the extraction of its state; this is so its state can be stored in a database (or other storage mechanism used in the implementation of repository). However these are (once again) not needed for this example and are therefore omitted for brevity.

Repository as a singleton

Presumably there will only be one instance of repository needed by the drawing program, and in this example, I am assuming this is the case. Therefore, it seems reasonable (or does it? – but I'm coming to that) to apply the Singleton design pattern – i.e. to make it such that:

- There can be only one instance of repository in the program
- The one instance is globally accessible wherever it is needed

Mark Radford is a freelancer with nearly twenty years experience in software development. His current main interests are in design and programming in (among other languages) C++, C#, Python, Perl and PHP. He can be contacted at mark@twonine.co.uk

```
class repository
{
public:
  static repository& instance()
  {
      static repository inst;
      return inst;
  }
  void save(const shape* s);
...
};
```

Listing 4

```
class counting_repository : public repository
{
public:
  counting_repository() : count(0) {}
  virtual void save(const shape* s)
  { ++count; }
  unsigned int num_saved() const
  { return count; }
...
private:
  unsigned int count;
};
```

Listing 5

There are many ways to implement Singleton, but the one used in Listing 4 to implement repository is quite a common one.

Now for the part played in this article by unit testing – it is time to write a (single) unit test for the **drawing::save** member function. I'm going to assume we have a drawing instance that contains five shapes. The unit test I want to write saves a drawing object in the repository, and then verifies that the number of shapes actually saved is equal to five:

```
void unit_test(const drawing& d)
{
  d.save(repository::instance());
  ...
}
```

As you may have noticed, there is no such test in the above function. This is because it suddenly becomes apparent that some work needs to be done. For this test, what I need is a repository implementation that can count shapes.

With the Singleton approach to repository's implementation, there are (at least) the following associated issues:

- In order to use repository implementations specialised for unit tests, it is necessary to link in a specialised test version of repository.
- As **repository** is – and with this approach, must be – hard coded in the test by name, it is not possible to have more than one **repository** implementation to test different things. Therefore, one test version of **repository** must support *all* tests, and must be modified when a new test is added.
- As a result of one test repository implementation supporting all tests, it is more difficult to test specific pieces of code. That is, implementing *unit* tests becomes more difficult.

The above issues are a direct result of accessing the repository in a global context – that is, a consequence of bypassing unit_test's programmatic interface.

This approach can be made to work. However, at this point, I'm suggesting there is a simpler method.

Using mock object

Let's put repository back the way it was when first introduced:

```
class repository
{
public:
  virtual ~repository();
  virtual void save(const shape* s) = 0;
...
};
```

It is now, once again, an Interface Class, and this means different implementations are possible. The unit test under discussion requires a repository implementation that can count shapes stored in it – see Listing 5.

There you have it: **counting_repository::save** does not (in this particular test implementation) actually save anything, it just increments a counter. This approach is known as Mock Object (sometimes known as Mock Implementation). It's time to take stock of how the unit test now looks:

```
void unit_test(const drawing& d)
{
  counting_repository counter;
  d.save(counter);

  assert(counter.num_saved() == 5);
}
```

Note that because **counting_repository** is specific to this particular unit test, it can be defined within the unit test's code (e.g. within the same source file as the **unit_test** function). As a (pleasant) consequence, there is no need to link in any external code, and the unit test assumes full control over the test to be performed.

Finally

The approach of using Mock Object with unit_test is an example of a pattern known as Parametrise From Above. One perspective on Parametrise From Above is that it is the "alter-ego" of Singleton (and other approaches involving globally accessible objects). Singleton is a dysfunctional pattern – one that transforms the design context for the worse, rather than for the better. Parametrise From Above is a pattern that is "out there", but for which (to the best of my knowledge) there is (so far) no formal write up.

Encapsulation is fundamental to object oriented design – and Interface Class is an idiom that can underpin the strengthening of encapsulation. The Mock Implementation of repository is possible because repository is an Interface Class. Observe how **unit_test** can use different repository implementations, without **unit_test**'s implementation being affected.

References

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

[Radford/Griffiths] Mark Radford and Alan Griffiths, Separating Interface and Implementation in C++, Overload 66 (www.twonine.co.uk/articles/SeparatingInterfaceAndImplementation.pdf)

[Radford:1] Mark Radford, C++ Interface Classes – An Introduction, Overload 62 (www.twonine.co.uk/articles/CPPInterfaceClassesIntro.pdf)

[Radford:2] Mark Radford, C++ Interface Classes – Noise Reduction, Overload 68 (www.twonine.co.uk/articles/CPPInterfaceClasses-NoiseReduction.pdf)

A Drop in Standards?

Paul Johnson asks why, with the increase in numbers of those passing school-based examinations, today's graduates are less able than in previous times. What is going wrong?

In the UK, examinations are taken at 16 (GCSE) and 18 (A Level) before progressing onto a degree. In recent years, there has been an obsession with the annual increase in those passing exams at 16 and 18. According to the statistics, examination results from C to A* have been increasing year on year [BBC] and yet the quality of students going into (and more importantly, coming out of) both Further and Higher Education is on the decrease. If the official statistics are correct, why is there such a difference in the standards between now and 10 years back?

Or is it as cited by Baz Luhrmann in his song *Everybody's free (to use sunscreen)* [Luhrmann]:

> *Accept certain inalienable truths, prices will rise, politicians will philander, you too will get old, and when you do you'll fantasize that when you were young prices were reasonable, politicians were noble and children respected their elders.*

While this article is not so much about programming, it may cause some to sit up and question what is going on – especially to those who are professional mentors to newly qualified graduates in a company and wonder why their knowledge and key skills are not as high as expected.

The debate and ACCU

In 2004, I presented to the ACCU annual conference a talk on teaching platform independent code. While the talk was possibly not as good as it could have been (one of my failings is that I frequently completely re-write my presentations within a very short time-frame as they are possibly not quite what is expected by myself or the potential audience), the questions at the end did raise some interesting topics.

It had been noticed by recruitment agencies and employers present that while candidates came from Higher Education with the usual spread of 1st to 3rd class honours in Computer Sciences, they had difficulty with threading models, their knowledge of the STL varied from being able to use it at a basic to medium level (they could understand a template, but possibly could not construct one) to having been taught C++ using C-Front (so it was effectively a very old implementation) and key skills such as numeracy and literacy should a distinct lack of understanding or application.

There was a difference if the student came from Oxbridge, but it was not as much as would have traditionally been expected.

Some complained that those mentoring had now to put in more time in doing what the education system should have done than at any other time in their careers. To paraphrase one person in the discussion: There is something rotten in education and it is compounded the further up the scale they go.

Having worked in Further and Higher education since 1992, this revelation, while not completely unexpected, was interesting to say the least. It was further compounded by my teaching experience of 1st and 2nd year students on BSc programming courses.

Was the assertion that the compounding was happening further up or was it a case of an institution only being able to work with the materials provided?

Rationale for this article

Recently, I left my otherwise completely secure position at the University of Salford to commence teacher training at a local school. I already have my teaching qualifications for teaching in a college of F.E. and in universities and wanted to round it off with being able to teach 11 to 16 year old children.

I had been at Salford since 1995 and had witnessed a decrease in ability over my 11 years there. It was not only the actual knowledge side which was not as high, but also the attitude of the students. I will admit that I finished my first degree in 1998, but the prevailing attitude was that if I missed some work, it was not up to the lecturer to pass me the notes, but that I had to make the effort to obtain them from one of my fellow students. This especially applied when taking higher degrees.

During my final year of lecturing at Salford (2005), the attitude was no longer that they had to work, but I had to spoon feed them. The upshot was a degradation of the final worth of their degree.

My question though was with the constant claims of improvement at GCSE and A level, why were students coming to university sometimes unable to string a single cohesive argument together and why could they not understand the need for something as pointer checking when allocating memory?

Were was education failing?

Let's start off this trek by going backwards from degree to college level. For this, I have had the assistance of two lecturers at a local college. Due to the nature of their comments and the state of colleges within the UK currently, they will remain anonymous as will the names of their institutions.

Colleges of Further Education or Colleges of Getting Them In?

Colleges, in the UK, are run as businesses and have been since 1992 [FHEA 1992].

> The more bums there are on seats, the happier the bean counters are. Staffing levels aren't that important as they don't always seem to realise that without enough staff, money walks out of the door. They do realise though that we cost money.

While this is not always a bad thing, it does raise the question of the suitability of students coming in and going out. It is certainly the case that in the mid 1990s, colleges would take anybody in, irrespective of their ability and the way the funding councils worked was that the more that

Paul Johnson is a qualified lecturer (HE and FE) as well as a programmer. Until September 2006, he worked at the University of Salford. He is currently on an agency lecturing contract in a college of FE. He can be contacted at: paul@all-the-johnsons.co.uk

There is something **rotten** in education and it is **compounded** the **further up** you go

entered, the more money the college received. It was therefore in the interest of the colleges to spoon feed.

Happily, this has changed now – it is now roughly 10% on enrolment, the rest when they qualify. The benefit is that colleges now do not have to take everyone.

This sounds far more like it – they are working on true business lines of performance related earnings.

At this point, I will let my narrative move to the questions I asked of my colleagues.

- Is FE today geared more towards a product [ProdLearn] rather than a process model [ProcLearn]? In other words, the students are spoon fed and it's pretty near impossible for them to fail rather than when we did a course, we had to work to get the grade.

  The passing everyone thing happened because the funding model was shifted from we pay you in full for every one that enrols to you get a bit and the rest when they pass though a certain amount of it goes on in recent years there has been a move towards what has been termed recruitment with integrity) so we are not forced to pass everyone.

- Over the years that you've been in FE, would you say that the level of student coming in (academically) has dropped, stayed the same or improved. I'm gearing this towards the annual statistical figures the government release.

  Currently students seem to be coming in with weaker skills in the areas of writing, reading and maths which leads to much higher amounts of support being required; in some cases ridiculous levels of help are required just to achieve a passing grade.

  The abilities in maths and English do appear to be lower than in the past and in some noticeable cases, lower than expected over all academic ability.

  I would put this down to the schools obsession with targets. Let me illustrate: let's say the government target is 4 GCSE's at C or better. Some schools use a GNVQ intermediate which is equivalent to that to make up the numbers.

- Is the approach taken at FE failing the student in that because of the method of teaching taken at school and college level, their ability to think and work independently has been diminished to such an extent that they are not really that suitable for the outside world of work.

  I do feel that at our institution at least we try to promote their ability to think and work independently in there chosen academic field this is made difficult some times by the mentality they come out of school with (this depends on school).

- Is there an undue bias when students come in and leave to particular software and/or operating systems and hardware architectures and at the end of the day, is this really a bad thing? Remember though when answering this the major problems with any and all mono-cultures.

  It is true that nearly all computer systems in use in teaching in schools and colleges is the old Microsoft/lintel platform not really a bad thing in a way as that is the kind of equipment the students are likely to have at home, this can be bad if they become faced with an entirely different platform when they enter employment.

  In specialist subject areas however other platforms / architectures are in use more than Intel/Windows, for instance Art and media departments use more Macintosh computers as they are more favoured by the digital art and design community.

  In what I will call "technical computing" (How computers work, Writing programs, networking and file servers) Microsoft Intel does play a large part but other platforms such as Unix/Linux do feature prominently in teaching about the some subjects with in this area.

  In an area I will call "applied computers", not computing, computers. things like engineering, computer aided design and manufacture there is a tendency towards technology unique to a piece of equipment, some based on Intel/windows some based on computerised control units called Programmable logic controllers (PLC's).

  More generally in what I will call "IT" - using computers (word processing , spreadsheet use, data entry etc.) the Microsoft/ Intel is all pervasive in with this group i would also place all teaching areas out side of the specified areas above.

- Given and depending on the above, is it your professional opinion that education (or the method of education) is failing those it aims to help?

  I feel that the following things have a detrimental effect on the education system particularly the further education system that could be described as aspects leading to the system failing people who want an education.

  Governments unrealistic ideas e.g 50% of the population will go to university [DfES 2002].

  Funding geared to passing students this means there is heavy pressure to get students through this leads to some students receiving undue help to complete when they should be leaving with a fail grade. In the eyes of employers this leads to a devalued qualification and may lead to a college leaver being sacked for not being able to do what the qualification says they can.

  Schools disadvantaging the school-leavers with the range of easy subjects they are taught in preference to the more difficult subjects that can lead to better jobs. There was an article in the Times Ed. supplement [Tiimes] not too long back that course work is to be dropped from some GCSEs.

  The content of GCSE and A levels has been watered down over time. The marking of these is just as rigorous as it has

The more **bums** there are on **seats**, the **happier** the **bean counters** are

always been – the criteria has been changed and how grades are derived from marks out of 100.

There is now almost no funding for adult courses for people to reskill for a new career or help get them off Benefits and into work for good or into university *erm* what was that about 50% of the population again?

So, it's not all the colleges fault then?

Certainly food for though. A college is only able to deal with what comes through their doors and if a good chunk of that time is spent in bringing the student up to the level required for them to be able to understand the basics, then it is of little wonder that those moving up to Higher Education aren't able to cut it. The problem there though is that by the time students reach HE, educationally, they're probably only at what would be considered FE level 6 years ago.

This certainly adds strength to the argument that students coming through are not up to the same standards as they were in the mid to late 1990s.

Let's go back to school

Schools are obsessed – and not directly with the education of children. They want to be at the top of their respective league tables. It doesn't matter that the statistics don't tell the whole truth (some schools improve their results by not entering pupils who are likely to fail or not achieve a C, and rarely take into account local demographics such as unemployment and the effect[LitTrust] that has on learning). If they can be close to the top of the league, they are able to fill their classrooms far easier and attract lucrative grants.

To counter this, the government launched their "Every Child Matters" initiative [ECM]. This is a very well meaning and, if it works, an attempt to tailor learning to each child. There is only one problem – schools do not exercise this correctly, it is commonly referred to (by teaching staff) as "*Every Child Matters as long as they get a C or above*". The league table mentality still applies.

If we ignore this for the moment, can the problem really be placed as far down as secondary education? Are pupils coming out of schools really any worse than in the late 1980s academically? It is not a good idea to compare the examination results at either GCSE or A Level for this either. On a BBC Five Live interview (2004) at the time of the A level results coming out, when asked by Nicky Campbell, the education minister admitted on air that A levels had become easier!

Examining the exams

Exams are still marked as rigorously as they were 20 years ago. This has not changed. What has changed though is what is on the paper. Take the example of a question on a GCSE 2005 Maths [AQA] paper shown in the sidebar.

GCSE Maths Paper – 2005

The time taken, in minutes, by each of 15 pupils to travel to school, is shown in the ordered stem-and-leaf diagram.

Key 3 | 2 represents 32 minutes

0	5	5	8		
1	0	2	4	5	9
2	3	5	6	6	
3	2	4			
4	6				

a. One of these pupils is chosen at random. What is the probability that this pupil took less than 20 minutes to travel to school?
b. What was the median number of minutes taken to travel to school
c. Another people takes 37 minutes to school. Tick the correct box to show, if any, this has on
   - the median (decreases/same/increases)
   - the range (decreases/same/increases)

If we ignore the poor English used on the paper, this is one of the harder questions in the traditionally harder section B. In total, 50 minutes is allowed for the paper with a break in between each section.

I will assume for the time being that you've managed to stop laughing at this question. You'll need to have stopped otherwise you'll have a wet patch after this one – when A level is reached, a modular exam can be taken as many times as a student wants to (even if they have passed) and only the highest mark is the mark which is put forwards!

If gaining the mark means that the teacher spoon feeds or tailors the teaching to purely teach to the syllabus with very little time for much else, then that is what happens. This is probably going to be as close to the Utopian dream of factory style education. With all that has been said of Every Child Matters, it is impossible to reconcile the two.

This "pass whatever comes" attitude is costing us dearly. Not just educationally, but also economically as we produce graduates who are unable to compete with our European neighbours when it comes to what they come out of our education system with.

At this point, the words of Baz come to mind again. Do we not have this "it was easier in my day" point not apply here? It's a fair comment – most generations consider examinations harder in their day, however....

In 1994, I was part way through my HNC. I was teaching some bright 2nd year A level students Chemistry. They were full of themselves having just passed with flying colours another module test. I set them a challenge as it was close to Christmas. They were to sit the GCE paper I sat in 1987. Those who gained C or above, I'd buy drinks for on a night out – those who failed bought me the drinks.

Needless to say, I was very drunk that weekend!

Yes, this is anecdotal, but it did show something. These students, all of whom were heading for an A could not pass a GCE paper set 7 years earlier.

The culture of statistics and unachievable targets in schools seems to be the main driver of a lower set of standards

Back to University

We have gone full circle now with each step down the education path showing that the failing comes from the one preceding it and at the end, the culture of statistics and unachievable targets in schools seems to be the main driver of a lower set of standards.

What does this mean to universities?

Other than having to include remedial English and Maths to enable students to write their final dissertations, we have second year students on programming courses who have never come across `new`, `delete`, `calloc`, `malloc` or `free`, never check to see if a pointer has a value allocated to it and who have such a mediocre grasp of the STL as to make you cringe [C++]. The use of debuggers is almost unheard of in some establishments!

Is it little wonder that companies rarely trust graduates? Sure, there is a variance across the university sector as well as a variance across year groups, but the trend is still downward irrespective of the establishment attended.

Can there be a value placed to a degree or any formal certificate obtained via the traditional education route when the quality at the end is demonstrably lower than it was even 10 years back? The popularity of the BCS examinations with employers has been growing in the past 6 to 7 years as they are seen to still be at the same level of difficulty as they have been for many years. The likes of the BCS can do this as they are not restricted by the whims and caprices of the ruling party of the time.

Conclusion

If you've come this far, you're probably feeling somewhat incensed. Different political parties have done nothing other than destroy our once great education system. Since the inception of the GCSE in 1988, there has been (on average) 2 education bills or reforms each year; some are larger than others. The constant fiddling has eroded what we once had.

Sure, the examination system had to be changed as it had become recognised that the different style of learners [LearningStyles] responded differently to exams and that in order for everyone to have a fair crack, something had to change. However, since that time, the constant messing has changed the award of a GCSE as being nothing sellable in the employment market with even an A Level not having the clout it once had.

Credits and disclaimers

I must thank my friends at colleges and schools around the UK who have helped me with this feature. Without them, I would have not had half of the material needed to construct such a commentary.

References

[AQA] AQA, Mathematics (MODULAR) (SPECIFICATION B). Module 1 Intermediate Tier Section B.

[BBC] BBC website : http://news.bbc.co.uk/1/hi/education/5278774.stm

[C++] Possibly the worst C++ course in the world – EVER : http://www.all-the-johnsons.co.uk/cpp-worst-index.shtml

[DfES2002] 2002 Spending Review : http://www.dfes.gov.uk/2002spendingreview/06.shtml

[ECM] Every Child Matters : http://www.everychildmatters.gov.uk/

[FHEA 1992] Further and Higher Education Act 1992 (c.13) : http://www.opsi.gov.uk/acts/acts1992/Ukpga_19920013_en_1.htm

[LearningStyles] Learning styles : http://www.ldpride.net/learningstyles.MI.htm

[LitTrust] The link between poverty and exam results : http://www.literacytrust.org.uk/Database/stats/poorexam.html

[Luhrmann] Baz Luhrmann: *Everybody's free (to wear sunscreen)* – http://www.generationterrorists.com/quotes/sunscreen.html

[ProcLearn] Process learning : http://www.infed.org/biblio/b-learn.htm#process

[ProdLearn] Product learning : http://www.infed.org/biblio/b-learn.htm#product

[Times] QCA report on cheating (linked from the TES website) : http://www.qca.org.uk/2586_15523.html