contents

credits & contacts

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

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

Readers:

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Chris Lowe
ads@accu.org

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

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

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: The Value of What You Know

One of the things that constantly surprises me is the differences in value placed upon knowledge by those that have it and those that lack it. It often seems that anything that one knows is considered trivial or easy – and that anything one doesn't know is correspondingly complicated or hard.

Thus it is not uncommon to see a developer who spent days or weeks learning how to manage a technology expecting another to "pick it up" in a matter of minutes. Naturally, as developers are not a race apart, this also happens in other areas of endeavour: I've seen chess players run through the moves and rules in less than a minute and expect the explanation to be understood. Or those versed in cooking giving explanations of a recipe that would only make sense to someone that already knew most of the answer.

On the other end of such discussions the slightest confusion or ambiguity becomes a major setback and an obstacle to understanding. However, to my bemusement, the enigmatic mutterings are not seen as a failure of explanation but as a failure of understanding.

This is the context in which Overload operates: we all have pieces of knowledge that may be of use to others – but we fail to see the value in them and often lack the expertise to explain them. There can be very few of you reading Overload that do not have some knowledge or expertise to share, and the "readers" are here to provide assistance in conveying such expertise in a manner that is comprehensible. One only has to note the number of authors that feel the need to acknowledge their assistance to realise that this assistance is both necessary and valued. But most importantly it is available.

Why am I telling you all this? It is because Overload is dependent upon the willingness of ACCU members to write for it. And, despite the increasing number of ACCU members the number writing for Overload is not healthy. We get by but, on this occasion, it was only achieved by the last minute efforts of a number of contributors who responded to a plea from the editor. To avoid placing that pressure on them again, I'm making a plea now: please make a contribution to Overload. *This means you!*

Think back over the last week: how many things have you explained to other developers? How many of these do you understand well enough to think, "no-one would be interested in that"? Well, those are the things that you are expert enough on to write about. Try it – most of the authors find that the feedback they get makes the effort worthwhile.

And speaking of feedback: I'd like to thank all of those that helped with this issue, especially those that contributed to the last minute effort to fill the pages. I know that this time of year there are other demands on your time.

Three Perplexing Properties

There is rhetorical value in the number three ("The Three Bears", "The Three Billy Goats Gruff", or any number of political speeches). And it is also said that accidents happen in threes. I'm sure that the following three incidents don't quite qualify as accidents – there was a certain amount of intent involved. But they certainly represent missteps, did come as a triplet, and reflect some of the difficulties that occur when working in our field.

There is often a need to store arbitrary elements of configuration information and developers in many programming languages have come up with the same approach: store an associative collection of strings mapping keys to values. The keys provide convenient tags for values to be retrieved without the collection needing to be written with any knowledge of the information stored. And, since most languages allow values of various types to be represented as string values this affords the necessary degree of flexibility. Sometimes other types of value are used as a key (see "The tale of a struggling template programmer" [Overload 61] or its sequels [Overload 61, 62]), sometimes it is possible to store the values without converting their type.

In Java I've use the `Properties` class for this purpose and wrote a translation of this design in C++ for a recent project. The translation wasn't exact – there are a number of design decisions in the Java libraries that I find questionable. For example, the Java library treats `Properties` as a specialisation of `HashMap` whereas my implementation used delegation to a `std::map`. In a strongly typed language why allow a `Properties` class (which specifically contains `String`s for both keys and values) to be treated as a `HashMap` (that can contain arbitrary objects). Anyway, I did things my way and refused to expose the container interface.

The implementation language forced another difference – C++ doesn't allow null references, so I had to decide what to do when an invalid key was supplied. My decision caused a lot of discussion during the code review (yes, this client has code reviews; no, the moderators don't stop digression into solutions). What did I decide to do? Well, as I intend to cover the design options later, I'll avoid that discussion at this point.

First Motif

The code went into production and I didn't look at it for several months. I had no reason to until I happened to revisit some code that had used it – when I noticed that the classname had changed. Curiosity aroused I went to have a look. During the project lifecycle the original properties class had been renamed to `foo_properties` and "improved" by giving it a constructor that parsed a domain specific string format and a member function to produce such a string. This format (similar to field value pairs in a URL) contains embedded key/value pairs. The developer in question needed this functionality and was evidently convinced that this was the right class to support this functionality. (After all there was no other class to which these functions belonged!) There is even prior art: the Java library `Properties` class can serialise and deserialise itself.

Personally I didn't (and don't) see why these functions belonged as part of this class. It already did one thing well, and adding a

second role makes it less, not more usable. The class didn't need these functions to perform its role: they could be implemented efficiently via its public interface. And, if cohesion isn't a strong enough argument then consider the coupling introduced: the `properties` class was now attached to this `foo` string format.

There are several points to be considered here: when the change was made there was only one client of the `properties` class, so it was simple to make the change. It is also a good pocket example of how adding functionality to a component reduces its reusability. For many of our colleagues this is counter-intuitive and a good supply of examples is needed to convince them otherwise.

By the time I saw it, the code was in production (changes manage to achieve that without being reviewed) and there was enough work to do without fixing things that were not manifesting problems to the user. (Like fixing things that didn't work.)

Second Motif

"How do I return a `NULL` string?" came the voice from behind me. Like most of these questions it is worth finding out a bit of the context: I could guess that the language was C++, but why would anyone want to return a `NULL` `std::string`?

The answer to that wasn't illuminating: "because that's what I'd do in C". My colleague knew that in C a string is represented as a `char*` and that can be `NULL`. But he also knew that a C++ `std::string` cannot be `NULL`. Stop. Rewind. What was he trying to do?

It turns out that the problem is how to indicate a lookup failure in an associative map of `string` key to `string` value. And, as with the way my colleague would have implemented it in C, returning `NULL` is exactly the choice made in the Java library's `Properties` class:

```
public String getProperty(String key)
```

Searches for the property with the specified key in this property list. If the key is not found in this property list, the default property list, and its defaults, recursively, are then checked. The method returns null if the property is not found.

So there are precedents for returning `NULL`. Also, in the C++ standard library is a similar solution in a different but analogous situation. From "Associative container requirements" (23.1.2/7):

`a.find(k)`

returns an iterator pointing to an element with the key equivalent to k , or a.end() if such an element is not found.

This may not be `NULL`, but it is definitely a special value – with all the difficulties that this causes the client code.

Returning to Java the `Properties` class also provides an alternative solution:

```
public String getProperty(String key,
                          String defaultValue)
```

Searches for the property with the specified key in this property list. If the key is not found in this property list, the default property list, and its defaults, recursively, are then checked. The method returns the default value argument if the property is not found.

While we are considering possible solutions there is another one in the C++ standard library. (From 23.1.1/12)

The member function `at()` *provides bounds-checked access to container elements.* `at()` *throws* `out_of_range` *if* `n >= a.size()`.

Lets recap: we have seen three possible behaviours in the face of a request for the value associated with an unknown key:

1. Return a special value to indicate failure
2. Return a default value supplied by the caller
3. Throwing an exception

While these may all be reasonable solutions they are not all appropriate to every situation. And, there are additional options: "fallible" return types; stipulating the result as "undefined behaviour"; aborting the program; user-supplied callbacks and status flags (of various scopes) indicators are all possibilities (but less commonly used).

In order to recommend a solution it is necessary to know even more about the problem. It turns out that my questioner was in the process of designing a class to hold the configuration parameters for an application. Hey! That sounds familiar, maybe there is some prior art – even an existing implementation somewhere?

But before I could look for prior art or a library to use my questioner was gone with: "the configuration values should be there – I'll throw an exception. Thanks!"

Third Motif

When I started this editorial I had three discussions of implementations of "Property" classes to discuss. But between then and now one of them has evaporated into the mists of memory and been dispersed by the force of ongoing commitments. I'll have to trust that you have your own story to tell.

I fear that you will.

I wonder: how many times has this particular wheel been invented?

Alan Griffiths
alan@octopull.demon.co.uk

Copy Deadlines

All articles intended for publication in *Overload 63* should be submitted to the editor by September 1st 2004, and for *Overload 64* by November 1st 2004.

Letters to the Editor(s)

Dear Editor

I was reading Stefan Heinzmann's paper "The Tale of a Struggling Template Programmer" in June 2004 Overload, and I could not help thinking that a 2 page long code listing cannot possibly be a proper solution to such a simple problem!

To make the following discussion clearer, this is Stefan's declaration of the `lookup` function:

```
template<class EKey, class EVal, unsigned n,
         class Key, class Val>
EVal lookup(const Pair<EKey, EVal>(&tbl)[n],
            const Key & key, const Val & def)
```

As it is clearly stated by Phil Bass in his solution, the real problem in this declaration is the fact that we do not really want the types of the `key` and `def` function arguments to be automatically deduced by compiler. What instead we want is to force the compiler to deduce the types of the `EKey` and `EVal` template arguments by looking at the type of the `tbl`'s `Pair` elements, and then use these deduced types as the types of the `key` and `def` function arguments.

Using pseudo code this is how it would look:

```
template<class K, class V, int N>
V lookup(const Pair<K, V>(&tbl)[N],
         typeof(K) key,
         typeof(V) const & def)
```

Now in order to translate this pseudo code into real code the only thing we need is a way to name the types of the `Pair`'s `K` and `V` template arguments. And by far the simplest way to create a name for the type of a template argument is by creating a `typedef` within the definition of the template itself:

```
template<class K, class V>
struct Pair {
  typedef K key_type;
  typedef V mapped_type;
  key_type key;
  mapped_type value;
};
```

Once this is done we can rewrite the signature of the `lookup` function this way:

```
template<class K, class V, int N>
typename Pair<K, V>::mapped_type
    const & lookup(const Pair<K, V>(&tbl)[N],
           typename Pair<K, V>::key_type key,
           typename Pair<K, V>::mapped_type
                        const & default_value);
```

Which solves pretty much all of Stefan's problems related to the `lookup` function (no more ugly casts, function can return the result by reference).

I am attaching the code for the complete solution at the bottom of this mail.

In order to keep the code as simple and as clean as possible, I have decided to provide global definitions of the `<` and `==` operators for the `Pair` type instead of resorting to a custom function object. Given that the `Pair` type is a very simple type whose usage is entirely under our control I feel this is not at all inappropriate.

Kind Regards

Renato Mancuso

http://www.renatomancuso.com

```
#include <iostream>
      // for std::cout, std::cerr, std::endl
#include <algorithm> // for std::lower_bound
#include <cassert>   // for the assert macro

// This is the definition of the Pair
// template class.
// We do not declare a constructor since
// we want this struct to be a POD type.
template<class K, class V>
struct Pair {
  typedef K key_type;
  typedef V mapped_type;
  key_type key;
  mapped_type value;
};

// These are the global operator < and == for
// the Pair template class. They define a weak
// ordering relationship based on the value of
// the Pair's key data member. NOTE: Comeau
// 4.3.3 STL requires the declaration of the
// complete set of relational operators. This
// is not correct according to the Standard.
template<class K, class V>
inline bool operator<(const Pair<K, V> & lhs,
                      const Pair<K, V> & rhs) {
  return lhs.key < rhs.key;
}
template<class K, class V>
inline bool operator==(const Pair<K, V> & lhs,
                       const Pair<K, V> & rhs) {
  return lhs.key == rhs.key;
}

// This is the lookup function. It assumes
// that tbl's elements are sorted according
// to the // global < and == operators.
template<class K, class V, int N>
typename Pair<K, V>::mapped_type const &
      lookup(const Pair< K, V >( &tbl )[ N ],
          typename Pair<K, V>::key_type key,
          typename Pair<K, V>::mapped_type
                       const & default_value) {
  typedef Pair<K, V> pair_type;
  typedef const pair_type * iterator_type;
  const pair_type target = { key };
  iterator_type first = tbl;
  iterator_type last  = tbl + N;
  iterator_type found =
        std::lower_bound(first, last, target);
  if((found != last) && (*found == target)) {
    return found->value;
  }
  return default_value;
}
```

[concluded at foot of next page]

A Template Programmer's Struggles Revisited

by Chris Main

Overload 61 included a couple of lengthy articles ([1] and [2]) which demonstrated how difficult it is to undertake a small, realistic and well defined programming task using function templates and C++. The afterwords of the authors and of the editor of Overload suggested that C++ is too difficult to use.

The solution in the second article does indeed look verbose. Surely, I said to myself, there must be a better way. I wondered what I would have done if I had been faced with the same task.

What's Required?

A lookup table.

My initial reaction is to just use `std::map` unless there is a good reason not to.

Is there a good reason not to? In this case, yes, because it is also required to hold the table in non-volatile memory, which requires the table to be POD (a C-style array of a C-style struct). `std::map` does not fulfil this requirement.

We need something that behaves like `std::map` but is implemented with POD.

First Pass: Defining the Interface

Let's borrow the bits of the interface we need from `std::map`:

```
namespace rom {
  template<typename T1, typename T2>
  struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
  };

  template<typename Key, typename T,
          typename Cmp = std::less<Key> >
  class map {
  public:
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef Cmp key_compare;
  };
}
```

I'm sure if I had been doing this from scratch I would have tried to use `std::pair`, then realised like Stefan that this wouldn't work because it is not an aggregate. However I've used the hindsight I gained from reading his article to go straight to using a pair that supports aggregate initialisation.

Our new `rom::map` does not need a template parameter for allocation, since the whole point of it is that it uses a statically initialised array, so we discard that parameter of `std::map`.

The constructor of `rom::map` seems to be the obvious way to associate it with an array. The constructor would also be an ideal place to check that the array is sorted. Stefan used template argument deduction to obtain the size of the array but, as this fails on some compilers, I pass the size as a separate argument. The arguments of the constructor suggest the member variables the class requires:

[continued on next page]

```
// This template function checks that the
// table is sorted and that the key values
// are unique.
// Since this is a template function, it is
// only instantiated if it is called.
template<class T, int N>
bool is_sorted(T(&tbl)[N]) {
  for(int i = 0; i < N - 1; ++i) {
    if((tbl[i+1] < tbl[i])
        || (tbl[i+1] == tbl[i])) {
      std::cerr << "Element at index " << i+1
                << " is not greater than its "
                << "predecessor.\n";
      return false;
    }
  }
  return true;
}

// This is our test array mapping error codes
// to error messages.
const Pair<int, char const *> table[] = {
  { 0, "OK" },
  { 6, "minor glitch in self-destruction module" },
  { 13, "Error logging printer out of paper" },
  { 101, "Emergency cooling system inoperable" },
  { 2349, "Dangerous substance released" },
  { 32767, "Game over, you lost" }
};

// This is how we check that the array is
// sorted. It is done only in DEBUG builds.
#if !defined(NDEBUG)
namespace {
  struct check_sorted {
    check_sorted() { assert(is_sorted(table)); }
  };
  check_sorted checker;
}
#endif /* !defined(NDEBUG) */

int main() {
  // no need to cast the third argument to a
  // (char const*) since now the type of the
  // default_value argument is deduced from
  // the type of the elements of table[]...
  const char* result = lookup( table, 6, 0 );

  std::cout << (result ? result : "not found")
            << std::endl;
  std::cout << lookup(table, 5, "unknown error")
            << std::endl;
  return 0;
}
```

```
template<typename Key, typename T,
         typename Cmp = std::less<Key> >
class map {
public:
  typedef pair<const Key, T> value_type;
  map(const value_type array[],
      unsigned int array_size)
        : values(array), size(array_size) {}
private:
  const value_type * const values;
  unsigned int size;
};
```

The only member function we need is `find()`. For `std::map` this returns an iterator, but we can simply return a value because we are supplying a default value to use if none can be found. At this stage I want to verify that the interface is sound, so to get something that I can try out as early as possible I implement `find()` with a linear search rather than a more efficient binary search:

```
template<typename Key, typename T,
         typename Cmp = std::less<Key> >
class map {
public:
  const T &find(const Key &k, const T &def) const {
    for(unsigned int n = 0; n != size; ++n) {
      if(!Cmp()(k, values[n].first)
         && !Cmp()(values[n].first, k)) {
        return values[n].second;
      }
    }
    return def;
  }
};
```

Testing the Interface

Let's try it out. We know that the `rom::map` should behave like a `std::map`, so we write a utility to populate a `std::map` with the same table as a `rom::map` and check that every entry in the `std::map` can be found in the `rom::map`. Additionally we check that if an entry cannot be found in the `rom::map` the supplied default value is returned. (For brevity, I have implemented my tests with plain C `assert`s rather than use a unit test framework.)

```
namespace {
typedef rom::map<unsigned int,
                 const char *> RomLookup;

RomLookup::value_type table[] = {
  {0,"Ok"},
  {6,"Minor glitch in self-destruction module"},
  {13,"Error logging printer out of paper"},
  {101,"Emergency cooling system inoperable"},
  {2349,"Dangerous substances released"},
  {32767,"Game over, you lost"}
};

typedef std::map<RomLookup::key_type,
           RomLookup::mapped_type> StdLookup;

void PopulateStdLookup(
         const RomLookup::value_type table[],
         unsigned int table_size,
         StdLookup &stdLookup) {
  for(unsigned int n=0; n != table_size; ++n) {
    stdLookup[table[n].first] = table[n].second;
  }
}

class CheckFind {
public:
  CheckFind(const RomLookup &romLookup,
            const RomLookup::mapped_type
                                &def_value)
      : lookup(romLookup), def(def_value) {}
  void operator()(const StdLookup::value_type
                                &value) const {
    assert(lookup.find(value.first, def)
           == value.second);
  }
private:
  const RomLookup &lookup;
  const RomLookup::mapped_type &def;
};
}

int main(int, char**) {
  const unsigned int table_size
            = sizeof(table)/sizeof(table[0]);
  RomLookup romLookup(table, table_size);
  StdLookup stdLookup;
  PopulateStdLookup(table, table_size,
                    stdLookup);
  std::for_each(stdLookup.begin(),
                stdLookup.end(),
                CheckFind(romLookup, 0));
  assert(romLookup.find(1, 0) == 0);
  return 0;
}
```

This is all fine. We have a usable interface and set of test cases. Note that I didn't need to do any type casting to pass `0` as the default argument to `romLookup.find()`, it just compiled straight away with no problems.

Second Pass: Implementing the Binary Search

Now we need to refine `find()` to use a binary search, which requires `std::lower_bound`. My first attempt is:

```
template<typename Key, typename T,
         typename Cmp = std::less<Key> >
class map {
public:
  const T &find(const Key &k, const T &def) const {
    const value_type *value = std::lower_bound(
             values, values+size, k, Cmp());
    if(value == values+size
       || Cmp()(k, value->first)) {
      return def;
    }
    else {
      return value->second;
    }
  }
};
```

This gives me a compiler error saying it can't pass `value_types` to `less<unsigned int>`. It isn't too hard to work out that this is because I am passing a `key_type` comparison function to `std::lower_bound` which attempts to use it to compare `value_types`. So in the private part of the map I write a function object that adapts the key comparison function to work with `value_types`. Normally I do not bother to derive private function objects from `std::unary_function` or `std::binary_function`, but as this raised problems in the original article I did so on this occasion:

```
template<typename Key, typename T,
         typename Cmp = std::less<Key> >
class map {
public:
  const T &find(const Key &k,
                const T &def) const {
    const value_type *value =
         std::lower_bound(values, values+size,
                          k, value_compare());
    // rest of member function as before
  }
private:
  struct value_compare
    : public std::binary_function<value_type,
                                  value_type, bool> {
    bool operator()(const value_type &v1,
                    const value_type &v2) const {
      return Cmp()(v1.first, v2.first);
    }
  };
};
```

Still a compiler error, this time that I am trying to pass an `unsigned int` as an argument to `value_compare::operator()`. Again, it is not too difficult to spot that I am passing a `key_type` as the third argument of `std::lower_bound` where a `value_type` is required. We use the elegant fix employed in [2]:

```
template<typename Key, typename T,
         typename Cmp = std::less<Key> >
class map {
public:
  const T &find(const Key &k,
                const T &def) const {
    const value_type key = { k };
    const value_type *value =
       std::lower_bound(values, values+size,
                        key, value_compare());
    // rest of member function as before
  }
};
```

Now everything compiles cleanly (including the use of `std::binary_function`) and the test code also executes successfully.

Third Pass: Considering the Disadvantages

We have reached a solution that works. We reached it by a less painful route, with less code and with simpler code. But does this solution have some disadvantages the original did not have?

Most obviously, it does not provide a mechanism that can be used equally well for any map-like container: it is a less general solution. I'm not convinced this is a disadvantage. "Why restrict ourselves to arrays?" asks [2]. I'm tempted to reply "Why not?"

Another difference is that our `rom::map`s have two member variables that take up memory which the original solution did not. This may be insignificant, but since the context of the task is an embedded system it is conceivable that we may be required to conserve memory. If this is the case there is a simple refactoring that can be applied to the `rom::map`. The array can be passed directly to the `find()` member function, which can be made static, and the constructor and member variables removed. (If we had implemented a check that the array is sorted in the constructor, that code could also be refactored into a static member function).

At this stage, if I had a smart enough compiler, I could try to use template argument deduction to determine the array size rather than pass it as an explicit parameter. Personally, I don't think I would go to that trouble.

Fourth Pass: Things Get Nasty

If we find it necessary to eliminate the constructor and member variables, leaving only a static member function, the next obvious refactoring is to turn it into a standalone function. But if we do that, we run into the problems experienced in [1]. So we are faced with a choice: proceed with the refactoring and introduce the necessary traits class as in [2], or abandon the refactoring and stick with what we have. I'd go for the latter. The syntax is a little less elegant, but overall it's simpler.

Conclusion

Why did things run more smoothly with the approach I took? It is because my solution uses a class template rather than a function template. It therefore does much less template argument deduction, which avoids a whole host of problems.

This suggests a design guideline: if you are struggling to implement a function template, consider re-implementing it as a class template (as an alternative to introducing traits).

Chris Main
chris@chrismain.uklinux.net

Afterword

Is C++ too difficult? I'm not so sure. I think I've demonstrated that the code which provoked comments to that effect was unnecessarily complicated. I think I did so not because I am a C++ expert but because I followed strategies that are generally useful when programming: follow the pattern of a known working solution to a similar problem (in this case `std::map`), work incrementally towards the solution, try to keep things as simple as possible.

How would the problem be solved in other programming languages? In C you could use the standard library `bsearch()`. I have used it, but it is quite fiddly to get the casting to and from `void *` right, so in my experience it is not significantly easier to use than C++. What other languages could be used?

References

[1] S. Heinzmann, "The Tale of a Struggling Template Programmer", *Overload 61*, June 2004

[2] S. Heinzmann and P. Bass, "A Template Programmer's Struggles Resolved", *Overload 61*, June 2004

Handling Exceptions in `finally`

by Tony Barrett-Powell

Recently I was reviewing some old Java code that performs SQL queries against a database and closes the resources in `finally` blocks. When I examined the code I realized that the handling of the resources was flawed if an exception occurred. This article looks at how the handling of the resources and exceptions was problematic and some approaches to solving these problems.

The Problems

The code in question was made up of static methods where each method used a `Connection` parameter and performed the necessary actions to create a query, perform the query and process the results of the query. My problem came from the handling of the query and results resources, i.e. the instances of `PreparedStatement` and `ResultSet`.

The `PreparedStatement` and `ResultSet` were created in the main `try` block of the method and were closed in the associated `finally` block. The `close()` method of these classes can throw a `SQLException` and in the `finally` block each `close()` method was wrapped in a `try`/`catch` where the `SQLException` was caught and converted into a `RuntimeException` to be thrown. The outline of the original code is shown in the following listing:

```
public static ArrayList foo(Connection conn)
                        throws SQLException {
  ArrayList results = null;
  PreparedStatement ps = null;
  ResultSet rs = null;
  try {
    // create a query, perform the query and
    // process the results
  }
  finally {
    try {
      rs.close();
    }
    catch(SQLException ex) {
      throw new RuntimeException(ex);
    }
    try {
      ps.close();
    }
    catch(SQLException ex) {
      throw new RuntimeException(ex);
    }
  }
  return results;
}
```

There are a number of problems with this code:

1 If an exception is thrown in the `try` block and a subsequent exception is thrown in the `finally` block the original exception is lost.

The problem where an exception is hidden by a subsequent exception is well known and is discussed in a number of books: Thinking in Java [Eckel] 'the lost exception', Java in Practice [Warren] and Practical Java - Programming Language Guide [Hagger] to name a few. All discuss the problem and I will present a trivial version here with some example code:

```
public void foo() {
    try {
      throw new RuntimeException("Really
                                  important");
    }
    finally {
      throw new RuntimeException("Just
                                  trivial");
    }
  }
```

A caller of this function would receive the "Just trivial" exception, there would be no evidence that the "Really important" exception ever occurred at all. In the original code if an exception occurred in the finally block after a `SQLException` had been thrown in the `try` block, the `SQLException` would be lost.

2 The use of `RuntimeException`s to throw the `Exception`s caught in the `finally` block when the method would throw a `SQLException` from the `try` block is inconsistent, `SQLException` should be used for both.

3 If an exception is thrown by the closing of the `ResultSet`, no attempt is made to close the `PreparedStatment`, that may cause a possible resource leak.

Solutions

We can fix some of the problems very easily by nesting the handling of the resources in `try`/`finally` blocks (as demonstrated in [Griffiths]) and to remove the conversion to `RuntimeException`s. This would be implemented in the method as follows:

```
// assign query to ps
try {
  // perform the query and assign result to rs
  try {
    // process the results
  }
  finally { rs.close(); }
}
finally { ps.close(); }
```

This solves the second problem, as the method is already declared to throw a `SQLException` no conversion is required, and the third problem, because even if a exception is thrown by `rs.close()` the `ps.close()` will always be called. However this leaves the first problem of the lost exception.

The suggested approach in [Warren] is to *"Never let exceptions propagate out of a* `finally` *block"*, this would be implemented in the `finally` block as follows:

```
finally {
  try {
    rs.close();
  }
  catch(SQLException ex) {
    /* exception ignored */
  }
  try {
    ps.close();
  }
  catch(SQLException ex) {
    /* exception ignored */
  }
}
```

This approach only solves the hidden exception problem in the original code but as a consequence adds an additional problem: it is possible for the `rs.close()` to be the original exception and this is ignored. Ignoring an exception is likely to make recovery in higher levels of the code more difficult, if not impossible. It is also likely to mislead a user trying to determine the cause of a failure; a later related exception may be mistakenly diagnosed as the source of the problem. The consequences of ignoring exceptions are discussed further in [Bloch] "Item 47: Don't ignore exceptions".

[Hagger] offers a different solution to this problem by collecting up the exceptions thrown during processing of a method. This is achieved by the creation of a derived exception class containing a collection of other exceptions (a slightly modified version follows):

```
class FooException extends Exception {
  private ArrayList exceptions;
  public FooException(ArrayList exs) {
    exceptions = exs;
  }
  public ArrayList getExceptions() {
    return exceptions;
  }
}
```

And the original code is modified to make use of this exception:

```
public static ArrayList foo(Connection conn)
            throws FooException {
  ArrayList exceptions = new ArrayList();
  ArrayList results = null;
  PreparedStatement ps = null;
  ResultSet rs = null;
  try {
    // create a query, perform the query and
    // process the results
  }
  catch(SQLException ex) {
    exceptions.add(exception);
  }
  finally {
    try {
      rs.close();
    }
    catch(SQLException ex) {
      exceptions.add(ex);
    }
    try {
      ps.close();
    }
    catch(SQLException ex) {
      exceptions.add(ex);
    }
    if(exceptions.size() != 0) {
      throw new FooException(exceptions);
    }
  }
  return results;
}
```

This approach doesn't lose any of the exceptions thrown and the `PreparedStatement` will be closed even if the close of the `ResultSet` throws an exception, but now the method throws a user-defined `Exception` instead of `SQLException`. It is better to use standard exceptions where possible as discussed in [Bloch] *Item 42: Favor the use of standard exceptions. More importantly the exceptions are collected as peers not as causes, and so is not idiomatic (at least not since JDK1.4) where the* `Throwable` *class allows nesting of another* `Throwable` *as a cause* [JDK14].

`SQLException` was written before JDK1.4 and has its own mechanism for nesting other `SQLException`s, this is supported by methods `setNextException()` and `getNextException()`. This mechanism, being limited to `SQLException`, is not generally idiomatic for all `Throwable` types and so will be not be considered for the purposes of this article.

A More Idiomatic Approach?

So a `Throwable` (and its derived classes) can be constructed with a cause (if this support has been implemented), or can be initialized with a cause using the `initCause()` method. Nesting exceptions at different levels of abstraction has been idiomatic even before support was added to `Throwable`, an implementation of this can be found at `http://www.javaworld.com/javaworld/javatips/jw-javatip91.html`. So to be more idiomatic the same approach should be taken within the original method.

We can use a modified version of Hagger's solution, combining this with nested `try`/`finally` blocks from the first solution and nest the `SQLExceptions` using `initCause()`, if required. Thus the original code is rewritten:

```
public static ArrayList foo(Connection conn)
          throws SQLException {
  SQLException cachedException = null;
  ArrayList results = null;
  PreparedStatement ps = null;
  ResultSet rs = null;
  // assign query to ps
  try {
    // perform query and assign result to rs
    try {
      // process the results
    }
    catch(SQLException ex) {
      cachedException = ex;
      throw ex;
    }
    finally {
      try {
        rs.close();
      }
      catch(SQLException ex) {
        if(cachedException != null) {
          ex.initCause(cachedException);
        }
        cachedException = ex;
        throw ex;
      }
    }
  }
  catch(SQLException ex) {
    if(cachedException != null) {
      ex.initCause(cachedException);
    }
    cachedException = ex;
    throw ex;
  }
```

```
    finally {
      try {
        ps.close();
      }
      catch(SQLException ex) {
        if(cachedException != null) {
          ex.initCause(cachedException);
        }
        throw ex;
      }
    }
    return results;
  }
```

This solves the three problems of the original code, no exception is lost, the exception thrown is a `SQLException` and the `PreparedStatement` is closed even if the attempt to close the `ResultSet` results in an `Exception`. Unfortunately this isn't a general solution, the `initCause()` method is used to set the cause of a `SQLException` if an existing `SQLException` had been caught, but `initCause()` has some restrictions:

> *"`public Throwable initCause(Throwable cause)` Initializes the cause of this throwable to the specified value. (The cause is the throwable that caused this throwable to get thrown.) This method can be called at most once. It is generally called from within the constructor, or immediately after creating the throwable. If this throwable was created with `Throwable(Throwable)` or `Throwable(String,Throwable)`, this method cannot be called even once." [JDK14]*

This means that if the exceptions caught in the `finally` block already have a cause then the `initCause()` method call will fail with a `java.lang.IllegalStateException`. To explain further this example demonstrates how to provoke the failure:

```
  void AnotherThrowingMethod() {
    throw new RuntimeException();
  }
  void ThrowingMethod() {
    try {
      AnotherThrowingMethod();
    }
    catch(RuntimeException ex) {
      throw new RuntimeException(ex);
    }
  }
  void foo() throws Exception {
    Exception cachedException = null;
    try {
      ThrowingMethod();
    }
    catch(Exception ex) {
      cachedException = ex;
      throw ex;
    }
    finally {
      try {
        ThrowingMethod();
      }
      catch(Exception ex) {
        if(cachedException != null) {
          ex.initCause(cachedException);
          // error: IllegalStateException
          // Exception ex already has a cause
        }
        throw ex;
      }
    }
  }
```

The idiomatic approach could be written to check for this situation, for example the handling of the `PreparedStatement` could become:

```
  if(ps != null) {
    try {
      ps.close();
    }
    catch(SQLException ex) {
      if(ex.getCause() == null) {
        if(cachedException != null) {
          ex.initCause(cachedException);
        }
      }
      throw ex;
    }
  }
```

But this will mean that the original exception is lost, as discussed above, making Hagger's approach better in this case.

Summary

Handling exceptions thrown while in a `finally` block is problematic in the context of an existing exception. This article has presented some approaches that solve at least some of the problems discovered in the example but no approach is entirely satisfactory. For the example presented the idiomatic solution works and is the best solution.

In the wider context of a general solution each approach has drawbacks or will not work, for example the idiomatic approach will fail if the exception being handled already has a cause. Of the approaches presented I would use, in order of preference, the idiomatic version, then Hagger's approach (if the exceptions being handled could already have a cause). I would resist using the approach in [Warren] as ignoring exceptions is a particularly bad idiom and should be avoided under any circumstances.

Tony Barrett-Powell
`tony.barrett-powell@blueyonder.co.uk`

Bibliography

[Bloch] Joshua Bloch, *Effective Java - Programming Language Guide*, Addison-Wesley 0-201-31005-8

[Eckel] Bruce Eckel, *Thinking in Java, 3rd Edition*, Prentice-Hall 0-131-002872

[Griffiths] Alan Griffiths, "More Exceptional Java," *Overload 49* and also available at `http://www.octopull.demon.co.uk/java/MoreExceptionalJava.html`

[Hagger] Peter Hagger, *Practical Java - Programming Language Guide*, Addison-Wesley 0-201-61646-7

[JDK14] `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Throwable.html`

[Warren] Nigel Warren and Philip Bishop, *Java in Practice - Design Styles and Idioms for Effective Java*, Addison-Wesley 0-201-36065-9

ACCU Mentored Developers XML Project

Exercise 1: Validate XML Files and Display Element Structure

by Paul Grenyer and Jez Higgins

This article was originally written in December 2002 as part of the ACCU Mentored Developers [MDevelopers] XML [XMLRec] project. It has now been revised, with considerable help from Jez Higgins, for publication in Overload.

The first exercise set for the project students by the project mentors was as follows:

Incorporate either the Xerces[Xerces] or Microsoft XML[MSXML] parsers into a C++ project and use it to:

1. Parse XML strings and files.
2. Output the element structure as an indented tree.

As most of my development experience has been on Windows I followed the MSXML route.

Downloading and Installing MSXML

The MSXML parser can be downloaded from the Microsoft website. The latest version at the time of writing is version 4.0 and requires the latest Windows installer, which is incorporated into Windows XP and comes with Windows service pack 3. The installer can also be downloaded as single executable [InstMsi].

Assuming the latest Windows Installer is present on your system installing MSXML is simply a case of running the installer package. As MSXML is Component Object Model (COM) based this will register the MSXML dynamic link library (`msxml4.dll`). The installer also creates a directory with all necessary files needed to use the parser in a C++ project.

Testing MSXML

Although there are the usual Microsoft help files incorporated with MSXML there aren't any examples, so I used Google to try and find some and found the PerfectXML[PerfectXML] website. The website includes a number of MSXML C++ examples and one in particular, Using DOM [UsingDOM], that downloads an XML file from an Internet location, parses it, modifies it and writes it to the local hard disk. I used this example as a template for the following simple MSXML console application test program:

```
#include <iostream>
#include <string>
#include <windows.h>
#include <atlbase.h>
#import "msxml4.dll"
int main() {
  std::cout << "MSXML DOM: Simple Test 1: Creating"
     << " of COM object and parsing of XML.\n\n";
  ::CoInitialize(0);
  {
    MSXML2::IXMLDOMDocument2Ptr pXMLDoc = 0;
    // Create MSXML DOM object
    HRESULT hr = pXMLDoc.CreateInstance(
                  "Msxml2.DOMDocument.4.0");
    if (SUCCEEDED(hr)) {
      // Load the document synchronously
      pXMLDoc->async = false;
      _variant_t varLoadResult((bool)false);
      const std::string xmlFile("poem.xml");
      // Load the XML document
      varLoadResult = pXMLDoc->load(xmlFile.c_str());
      if(varLoadResult) {
        std::cout << "Successfully loaded XML file: "
                  << " file: " << xmlFile << "\n";
      }
```

An XML Mini-Glossary

Attributes – XML elements can have attributes. An attribute is a name-value pair attach to the element's start tag. Names are separated from their values by an equals sign, and values are enclosed in single or double quotes. Attribute order is not significant.

```
<bigbrain invented="SGML">Charles Goldfarb</bigbrain>
```

DOM – The Document Object Model is a W3C recommendation which an application programming interface well-formed XML documents [DOMRec], defining the logical structure of documents and the way a document is accessed and manipulated. The DOM is defined in programming-language neutral terms. This leads to some slightly clumsy looking code, but that aside the DOM is widely used (if not necessarily well-loved). Its in-memory representation makes it well suited to document editing, navigation and data retrieval applications.

DTD – Document Type Definition, the original XML schema language described in the XML recommendation. A Document Type Definition defines the legal building blocks of an XML document. It defines the document structure with a list of legal elements, each element's allowed content and so on.

Elements & Tags – Here's a tiny XML document

```
<bigbrain>Charles Golbfarb</bigbrain>
```

It consists of a single element named `bigbrain` and the element's content, the text string `Charles Goldfarb`. The element is delimited by the start tag `<bigbrain>` and the end tag `</bigbrain>`.

Valid – Documents which conform to a particular XML application are said to be *valid.* In the early days of XML (all of five years ago) validity meant conforming to a DTD. With the development and widespread adoption of other schema languages, valid has come to mean *valid to whatever schema you happen to be using*.

Well-formed – Not all, quite probably most, XML documents are not valid, nor do they need to be. However they are all *well-formed*. An XML document is well-formed if it satisfies the basic XML grammar – the elements are properly delimited, start and end tags match and so on. A document which is not well-formed is like a C++ program with a missing semi-colon, no good for anything.

XML Application – A set of XML elements and attributes for a particular purpose – for instance DocBook, SVG, WSDL, Open Office file format – is called an *XML application.* An XML application is often expressed in one of the many available schema languages – DTD, XML Schema, RelaxNG, Schematron, etc. An XML application is *not* an application which uses XML.

```
      else {
        std::cout << "Failed to load XML file: "
                  << xmlFile << "\n";
        // Get parseError interface
        MSXML2::IXMLDOMParseErrorPtr pError = 0;
        if(SUCCEEDED(pXMLDoc->get_parseError(
                                    &pError))) {
          USES_CONVERSION;
          std::cout << "Error: "
                    << W2A(pError->reason) << "\n";
        }
      }
    }
    else {
      std::cout << "Failed to create MS XML COM "
                << "object.\n";
    }
  }
  ::CoUninitialize();
  return 0;
}
```

This program takes the following XML file and parses it:

```
<?xml version="1.0" encoding="UTF-8"?>
<poem>
  <line>Roses are red,</line>
  <line>Violets are blue.</line>
  <line>Sugar is sweet,</line>
  <line>and I love you</line>
</poem>
```

If the parse fails an error message is written to `std::cout` giving the reason. Although this code snippet does the intended job, it is a bit rough and needs some work in order to achieve the objective of this exercise. Among other things it would benefit from wrapping of MSXML and some proper exception handling.

It is worth noting `#import` is specific to Microsoft Visual C++ and is not supported by other Win32 compilers.

Engineering the Exercise Solution: Part 1

I'm going to look at the exercise solution in two parts. The first part will reengineer the PerfectXML example into a more general solution with a clean interface, proper runtime handling and exception handling. The second part will look at writing the element structure to a stream.

COM Runtime

As MSXML is COM based, the COM runtime must be started before any COM objects can be instantiated. The COM runtime is started by the `CoInitializeEx` API function and stopped with `CoUninitialize`. MSDN states that every call to `CoInitializeEx` must be matched by a call to `CoUninitialize`, even if `CoInitializeEx` fails.

`CoUninitialize` must not be called until all COM objects have been released. For instance in the example above there is an extra scope wrapping the MSXML code so that the `IXMLDOMDocument2Ptr` smart pointer destructor is called, destroying the DOM, before `CoUninitialize` is called.

The easiest way to achieve this, even in the presence of exceptions, is to take advantage of C++'s RAII (Resource Acquisition Is Initialization) and place `CoInitialiseEx` in the constructor of a class and `CoUninitialize` in the destructor and to create an instance of the class on the stack, at the beginning of the program before anything else. `COMRuntimeInit`, shown below, is just such a class. The copy constructor and copy-assignment operator are both private and undefined, to prevent copying. A `COMRuntimeInit` object has no state and therefore it does not make sense to copy it. This method of preventing copying and some more of the reasons behind it are discussed by Scott Meyers in Effective C++[EC++].

```
#include <stdexcept>
#include <string>
#include <windows.h>
class COMRuntimeInit {
public:
  COMRuntimeInit() {
    HRESULT hr = ::CoInitializeEx(0,
                          COINIT_APARTMENTTHREADED);
    if(FAILED(hr)) {
      UnInitialize();
      std::string errorMsg = "Failed to start COM "
                             "Runtime: ";
      switch(hr) {
        case E_INVALIDARG:
          errorMsg += "An invalid parameter was "
                      "passed to the returning "
                      "function.";
          break;
        case E_OUTOFMEMORY:
          errorMsg += "Out of memory.";
          break;
        case E_UNEXPECTED:
          errorMsg += "Unexpected error.";
          break;
        case S_FALSE:
          errorMsg += "The COM library is already "
                      "initialized on this "
                      "thread.";
          break;
        default:
          errorMsg += "Unknown.";
          break;
      }
      throw std::runtime_error(errorMsg);
    }
  }
  ~COMRuntimeInit() {
    UnInitialize();
  }
private:
  void UnInitialize() const {
    ::CoUninitialize();
  }
  COMRuntimeInit(const COMRuntimeInit&);
  COMRuntimeInit& operator=(const COMRuntimeInit&);
};
```

There are of course times when the initial call to `CoInitialiseEx` may fail. The cause of the failure can be ascertained from its return value. The obvious way to communicate

the cause of the failure to the user is via an exception. This has the drawback that the destructor will not be called when the constructor throws and therefore `CoUninitialize` must be called manually. For now `std::runtime_error` will be thrown when `CoInitializeEx` fails, later on we'll look at a custom exception type.

As stated above, the `COMRuntimeInit` instance must be declared before any other object on the stack. The instance cannot be put at file scope as it throws an exception if it fails, so the obvious place is at the top of `main`'s scope. A `try`/`catch` block is also needed to detect the failure.

```
#include <iostream>
#include "comruntimeinit.h"
int main() {
  try {
    COMRuntimeInit comRuntime;
  }
  catch( const std::runtime_error& e) {
    std::cout << e.what() << "\n";
  }
  return 0;
}
```

Instantiating the MSXML DOM

Code that uses COM, as with most Microsoft API code, is just plain ugly and really should be hidden behind an interface. Exercise 1 of the XML project states that either the Xerces parser or the MSXML parser can be used. Ideally they should be easily interchangeable and their use completely hidden from the user. Hiding the ugly code *and* making the parsers easily interchangeable can be achieved with the Pimpl Idiom, as discussed by Herb Sutter in Exceptional C++ [ExC++].

The first stage in the exercise is to create the MSXML DOM parser. This is achieved with the DOM class:

```
// dom.h
// Forward declaration so that implementation
// can be completely hidden.

class DOMImpl;
class DOM {
private:
  DOMImpl *impl_;
public:
  DOM();
  ~DOM();
private:
  DOM(const DOM&);
  DOM& operator=(const DOM&);
};
```

The `DOM` class will form a basic wrapper for the `DOMImpl` class which will do all the work. `DOMImpl` is forward declared, so that its implementation can be completely hidden.

The `DOM` class implementation is shown below. It creates an instance of the `DOMImplclass` on the heap in the constructor and deletes it in the destructor.

```
// dom.cpp
#include "dom.h"
#include "domimpl.h"
DOM::DOM() : impl_(new DOMImpl) {}
DOM::~DOM() { delete impl_; }
```

`DOMImpl` creates the MSXML DOM parser in the same way as the PerfectXML example:

```
// domimpl.h

#import "msxml4.dll"
class DOMImpl {
private:
  MSXML2::IXMLDOMDocument2Ptr xmlDoc_;
public:
  DOMImpl() : xmlDoc_(0) {
    xmlDoc_.CreateInstance(
                    "Msxml2.DOMDocument.4.0");
  }
private:
  DOMImpl(const DOMImpl&);
  DOMImpl& operator=(const DOMImpl&);
};
```

Both `DOM` and `DOMImpl` have private copy constructors and copy assignment operators, again to prevent copying.

The above code does not include any error checking. It is possible for the call to `CreateInstance` to fail. The `msxml4.dll` may not be registered, for example. The success or failure of the `CreateInstance` call can be detected by its return value.

```
DOMImpl() : xmlDoc_(0) {
  HRESULT hr = xmlDoc_.CreateInstance(
                   "Msxml2.DOMDocument.4.0");
  if(FAILED(hr)) {
    std::string errorMsg = "Failed to start "
                           "create MSXML "
                           "DOM: ";
    switch(hr) {
      case CO_E_NOTINITIALIZED:
        errorMsg += "CoInitialize has not "
                    "been called.";
        break;
      case CO_E_CLASSSTRING:
        errorMsg += "Invalid class string.";
        break;
      case REGDB_E_CLASSNOTREG:
        errorMsg += "A specified class is "
                    "not registered."
        break;
      case CLASS_E_NOAGGREGATION:
        errorMsg += "This class cannot be "
                    "created as part of an "
                    "aggregate.";
        break;
      case E_NOINTERFACE:
        errorMsg += "The specified class "
                    "does not implement the "
                    "requested interface";
        break;
      default:
        errorMsg += "Unknown error.";
        break;
    }
    throw std::runtime_error(errorMsg );
  }
}
```

NonCopyable

We now have three classes which are "copy prevented", with a private copy constructor and copy assignment operator. There is a clearer way to document the fact that a class is not intended to be copied. When used by a number of different classes it also reduces the amount of code.

The `NonCopyable` class, show below, has a private copy constructor and assignment operator to prevent prevent copying. When another class inherits from `NonCopyable`, the private copy constructor and assignment operator are also inherited. This both prevents the subclass from being copied and documents the intention. The relationship between `NonCopyable` and its subclass is not IS-A and therefore the inheritance can be private.

As `NonCopyable` is intended only to provide behaviour to a derived class, rather than act as a class in its own right, its default constructor is protected, preventing a free `NonCopyable` object being created. Its destructor too, is protected to prevent a subclass being deleted via a pointer to `NonCopyable`. To further document this intention, the destructor is not virtual.

```
class NonCopyable {
protected:
  NonCopyable() {}
  ~NonCopyable() {}
private:
  NonCopyable(const NonCopyable&);
  NonCopyable& operator=(const NonCopyable&);
};
```

The `NonCopyable` class was written by Dave Abrahams for the boost [boost] library. I have recreated it here so that a dependency on the boost library is avoided.

Now that the `NonCopyable` class is in place the copy constructors and assignment operators can be removed from `COMRuntimeInit`, `DOM` and `DOMImpl`. They can then be changed to privately inherit from `NonCopyable`.

```
class COMRuntimeInit : private NonCopyable {
  ...
};

class DOM : private NonCopyable {
  ...
};

class DOMImpl : private NonCopyable {
  ...
};
```

Loading and Validating the XML

The MSXML DOM has a method that loads and parses an XML file. While parsing the file it is checked to make sure it is well formed and if there is a DTD or Schema specified it is also validated. If the file cannot be opened, is not well formed or cannot be validated the call fails.

The method is called `load` and takes a single parameter which is the full path to the XML file. To load and parse an XML file, a similar method can be added to `DOMImpl` and a corresponding forwarding function added to `DOM`.

```
class DOMImpl : private NonCopyable {
public:
  ...
  void Load(const std::string& fullPath) {
    xmlDoc_->load(fullPath.c_str());
  }
};
```

`main` can then be modified to call the new function with the path to an XML file.

```
try {
  COMRuntimeInit comRuntime;
  DOM dom;
  dom.Load("poem.xml");
}
catch(const std::runtime_error& e) {
  std::cout << e.what() << "\n";
}
```

Once again there is no way of detecting failure and the return value of the MSXML DOM `load` method must be tested to find out if it failed. If a failure has occurred an exception should be thrown.

```
void Load(const std::string& fullPath) {
  if(!xmlDoc_->load( fullPath.c_str())) {
    throw std::runtime_error(ErrorMessage());
  }
}
```

The method of extracting an error message from an MSXML DOM is a little fiddly, so I have placed it in its own function, `ErrorMessage`.

```
class DOMImpl : private NonCopyable {
public:
  ...
  std::string ErrorMessage() const {
    std::string result = "Failed to extract "
                         "error.";
    MSXML2::IXMLDOMParseErrorPtr pError =
                         xmlDoc_->parseError;
    if(pError->reason.length()) {
      result = pError->reason;
    }
    return result;
  }
};
```

A parse error is extracted from an MSXML DOM as an `XMLDOMParserError` object. The error description is fetched from the `reason` property. If no description is available, the `bstr_t` returned by `reason` has a length of 0. `bstr_t` is a wrapper class for COM's native `unsigned short*` string type. It provides a conversion to `const char*`, and thus can be assigned to a `std::string`.

Custom Exception Types

Our `main` function's body is

```
try {
  COMRuntimeInit comRuntime;
  DOM dom;
  dom.Load("poem.xml");
}
catch(const std::runtime_error& e) {
  std::cout << e.what() << "\n";
}
```

Currently the example throws a `std::runtime_error` if the COM runtime fails to initialise or if there is an XML failure. In both cases the error message is prefixed with a description of the type of error. Exceptions thrown as a result of the COM runtime failing to initialise are probably fatal and it may be appropriate for the program to exit, while for exceptions thrown due to an XML parse fail it might be more appropriate to log the error and move on to the next file.

These different categories of error would be better communicated by the exception's actual type and it is easy to add custom exceptions. Throwing different types of exceptions helps to maintain the context in which the exception was thrown and enables the behaviour of a program to change based on the type of exception that is thrown.

Deriving from `std::exception` not only means that custom exception types can be caught along with other standard exception types in a single `catch` statement if necessary, but also provides an implementation for the custom exception object.

```
class BadCOMRuntime : public std::exception {
public:
  BadCOMRuntime(const std::string& msg)
        : exception(msg.c_str()) {}
};
```

`std::exception`'s constructor takes a `char*`, but I know that I will be building exception messages with strings and following the model of `std::runtime_error`, `BadCOMRuntime`'s constructor takes a `std::string`.

`COMRuntimeInit`'s constructor must be modified for the new exception:

```
COMRuntimeInit() {
  HRESULT hr = ::CoInitialize(0);
  if(FAILED(hr)) {
    UnInitialize();
    std::string errorMsg = "Unknown.";
    switch(hr) {
      case E_INVALIDARG:
        errorMsg = "An invalid parameter was "
                   "passed to the returning "
                   "function.";
        break;
      ...
      default:
        break;
    }
    throw BadCOMRuntime(errorMsg);
  }
}
```

and `main` must be modified to catch the new exception:

```
try {
  COMRuntimeInit comRuntime;
  DOM dom;
  dom.Load("poem.xml");
}
catch(const BadCOMRuntime& e) {
  std::cout << "COM initialisation error: "
            << e.what()
            << "\n";
}
...
```

The exceptions thrown by `DOMImpl` are a little more complicated. `DOMImpl` throws exceptions when two *different* things happen and therefore requires two different exception types, which should be in some way related. One way to solve this is to have a common exception type for `DOMImpl` from which two other exception types derive.

`DOMImpl` is the implementation of `DOM` and any exception thrown by `DOMImpl` is most likely to be caught outside `DOM`. Therefore, to the user of `DOM`, who is unaware of `DOMImpl`, it is more logical for `DOM` to be throwing exceptions of type `BadDOM` rather than `BadDOMImpl`.

```
#include <stdexcept>
#include <string>

class BadDOM : public std::exception {
public:
  BadDOM(const std::string& msg)
        : exception(msg.c_str()) {}
};

class CreateFailed : public BadDOM {
public:
  CreateFailed(const std::string& msg)
        : BadDOM(msg) {}
};

class BadParse : public BadDom {
public:
  BadParse(const std::string& msg)
        : BadDOM(msg) {}
};
```

The constructor and `Load` function in `DOMImpl` can now be modified to use the new exception types and `main` modified to catch a `BadDOM` exception. For completeness sake, we also need a third `catch` block. The COM smart pointers generated by `#import` raise a `_com_error` if a function call fails.

```
try {
  COMRuntimeInit comRuntime;
  DOM dom;
  dom.Load("poem.xml");
}
catch(const BadCOMRuntime& e) {
  std::cout << "COM initialisation error: "
            << e.what() << "\n";
}
catch(const BadDOM& e) {
  std::cout << "DOM error: "
            << e.what() << "\n";
}
catch(const _com_error& e) {
  std::cout << "COM error: "
            << e.ErrorMessage() << "\n";
}
```

Engineering the Exercise Solution: Part 2

Now that the DOM is loading and validating XML the next part of the exercise is write the elements to an output stream as an indented tree.

Writing the Element Structure

The first step in enabling the elements to be written to an output stream is to pass one in. The obvious way to do this is to is to add a function to `DOMImpl`, and a forwarding function to `DOM`, which takes a `std::ostream` reference.

```
#include <ostream>
class DOMImpl : private NonCopyable {
...
public:
  void WriteTree(std::ostream& out) {}
...
};
```

Modifying `main` to call the new function means that results can be seen straight away as the `WriteTree` implementation is developed.

```
try {
  COMRuntimeInit comRuntime;
  DOM dom;
  dom.Load("poem.xml");
  dom.WriteTree(std::cout);
}
...
```

In order to write the complete tree, every element must be visited. Starting with the root element, the rest of the elements can then be visited in a depth-first traversal. I wrote the following function, based on some Delphi written by Adrian Fagg, which gets a pointer to the root element and then calls the function `WriteBranch` which recurses the rest of the tree.

```
void WriteTree(std::ostream& out) {
  MSXML2::IXMLDOMElementPtr root =
                    xmlDoc_->documentElement;
  WriteBranch(root, 0, out);
}
```

The `WriteBranch` function is also based on Adrian Fagg's Delphi code. The code is self explanatory, but basically it:

1. Gets the tag name of the element passed to it.
2. Writes tag names to the supplied `std::ostream` at twice the specified indentation.
3. The supplied element is then used to get a pointer to its first child.
4. If the child pointer is not 0, it is used to get the node type.
5. If the node is of type `NODE_ELEMENT` the `WriteBranch` method is called again (recursion).
6. The child pointer is then used to get the next sibling.
7. If there are no more siblings, the method returns.

```
void WriteBranch(
            MSXML2::IXMLDOMElementPtr element,
            unsigned long indentation,
            std::ostream& out) {
  bstr_t cbstr element->tagName;
  out << std::string(2 * indentation, ' ')
      << cbstr << std::endl;
  MSXML2::IXMLDOMNodePtr child =
                          element->firstChild;
  while(child != 0) {
    if(child->nodeType ==
                    MSXML2::NODE_ELEMENT) {
      WriteBranch(child,
                  indentation + 1, out);
    }
    child = child->nextSibling;
  }
}
```

The result of running the program is now that the following is written to the console:

```
poem
  line
  line
  line
  line
```

With that the exercise is complete.

Next Step

The logical next step would of course be exercise 2. However, as well as completing the exercises which help the students learn about XML, one of the aims of the ACCU Mentored Developers XML Project is to write a standard interface behind which any parser, such as MSXML or Xerces can be used. Therefore, the next step is to design a common interface to the DOM.

Paul Grenyer and Jez Higgins
paul@paulgrenyer.co.uk
jez@jezuk.co.uk

Thank You

Thanks to all the members of the ACCU Mentored Developers XML Project, especially Adrian Fagg, Rob Hughes, Thaddaeus Frogley and Alan Griffiths for the proof reading and code suggestions.

References

[boost] The boost library: `http://www.boost.org`
[DOMRec] W3C Document Object Model (DOM): `http://www.w3.org/DOM/`
[EC++] Scott Meyers, *Effective C++: 50 Specific Ways to improve Your Programs and Designs*. Addison Wesley: ISBN 0-201-9288-9
[ExC++] Herb Sutter, *Exceptional C++*. Addison Wesley: ISBN 0201615622
[InstMsi] Windows Installer 2.0: `http://www.microsoft.com/downloads/details.aspx?FamilyID=4b6140f9-2d36-4977-8fa1-6f8a0f5dca8f&displaylang=en`
[MDevelopers] ACCU Mentored Developers: `http://www.accu.org/mdevelopers/`
[MSXML] Microsoft XML parser: `http://www.microsoft.com/downloads/details.aspx?FamilyID=3144b72b-b4f2-46da-b4b6-c5d7485f2b42&displaylang=en`
[PerfectXML] PerfectXML: `www.perfectxml.com/msxml.asp`
[UsingDOM] Using DOM: `http://www.perfectxml.com/CPPMSXML/20020710.asp`
[Xerces] Xerces XML parser: `http://xml.apache.org/xerces-c`
[XMLRec] Extensible Markup Language (XML): `http://www.w3.org/XML/`

The Curious Case of the Compile-Time Function (An Exercise in Template Meta-Programming)

by Phil Bass

A Crime Has Been Committed

18 months ago I described a version of my Event/Callback library in an Overload article [1]. This library is used extensively in my employer's control systems software. A typical use looks like this:

```
// A class of objects that monitor some event.
class Observer {
public:
  Observer(Event<int>& event)
    : callback(bind_1st(memfun(
                  &Observer::handler), this))
    , connection(event, &callback) {}
private:
  void handler(int);  // the event handler
  typedef Callback::Adapter<
              void (Observer::*)(int)>::type
     Callback_Type;
  Callback_Type callback; // a function object
  Callback::Connection<int> connection;
                        // event <-> callback
};
```

Exhibit 1: The event/callback library in action.

The key feature in this example is that a callback and an event/callback connection are both stored in the `Observer` as data members. Some attempt has been made to support this idiom by providing various helpers (the `bind_1st()` and `memfun()` function templates[1] and the `Callback::Adapter<Pmf>` class template). However, there is still quite a lot of rather verbose boilerplate code. And that's a crime.

It has been clear for some time that we should be able to improve on this. There seems to be no fundamental reason, for example, why we can't combine the callback and its connection into a single class template (`Bound_Callback`, say) and use it like this:

```
// A class of objects that monitor some event.
class Observer {
public:
  Observer(Event<int>& event)
     : callback(event, &Observer::handler,
                this) {}
private:
  void handler(int);  // the event handler
  Bound_Callback<void (Observer::*)(int)>
     callback;
};
```

Exhibit 2: The goal.

The question is how should we write the `Bound_Callback<Pmf>` template?

1 These are not-quite-standard variations of `std::bind1st()` and `std::mem_fun()` developed in-house for reasons that are not important here.

Suspects and Red Herrings

The first thing that comes to mind is Boost [2]. There's bound to be a Boost library that provides what we need. The trouble is I can't find one.

Boost.Bind provides a lovely family of `bind()` functions that generate all kinds of function objects. Unfortunately, their return types are unspecified, so we can't declare data members of those types.

Then there's Boost.Function, which was designed for a very similar job and does provide types we can use as data members. I believe we could, in fact, use the `boost::function<>` template as the callback part of our `Bound_Callback`. What I haven't told you, though, is that an `Event<Arg>` can only be connected to callbacks derived from `Callback::Function<Arg>`. Clearly, as `boost::function<>` isn't derived from this base class it doesn't provide everything we need. And, of course, it doesn't know how to make the event/callback connection, either.

So, what about Boost.Signals? Well, yes, we could replace the whole of our event/callback library with `boost::signals`, but I'm reluctant to do that for several (not very good) reasons. First of all, I don't like the names: "signal" is already used for something else in Unix operating systems, and "slot" is a truly bizarre word for a callback function. Secondly, Boost.Signals does more than we need or want. Specifically, I'm not convinced that a general-purpose event/callback library should do its own object lifetime management and, anyway, we couldn't use that feature in common cases like Exhibit 1. Finally, if we were to use Boost.Signals the crime would be reduced to a misdemeanour and there would be little or no motivation for this article!

A Promising Lead

The astute reader may have spotted a clue in the first exhibit. The `typedef` isn't there just to provide a reasonably short name for the callback type – it also shows a template meta-function in action.

A meta-function in C++ is a compile-time analogue of an ordinary (run-time) function. Well-behaved run-time functions perform an operation on a set of values supplied as parameters and generate a new value as their result. Meta-functions typically perform an operation on a set of types supplied as parameters and generate a new type as their result.

In its simplest form, a meta-function taking a single type parameter and returning another type as its result looks like this:

```
template<typename Arg>
struct meta_function {
  typedef <some type expression involving Arg>
       type;
};
```

Exhibit 3: A simple meta-function.

In C++, a meta-function always involves a template. The meta-function's parameters are the template's parameters and the meta-function's result is a nested type name or integral constant. The Boost Meta-Programming Library adopts the convention that a meta-function's result is called `type` (if it's a type) or `value` (if it's an integral constant) and that same convention is used here.

Now, suppose we had a meta-function that takes a pointer-to-member-function type and returns the function's parameter type.

```
template<typename Pmf>
                // Result (Class::*Pmf)(Arg)
struct argument {
  typedef <magic involving Pmf> type;
                // type == Arg
};
```

Exhibit 4: A magical meta-function.

Similarly, we can imagine meta-functions that extract from a pointer-to-member-function the function's result type and the class of which the function is a member. We could now write a `Bound_Callback<Pmf>` template along the lines of Exhibit 5.

```
// A callback bound to an event.
template<typename Pmf>
class Bound_Callback
  : public Callback::Function<typename
argument<Pmf>::type> {
public:
  typedef typename argument<Pmf>::type Arg;
  typedef typename result<Pmf>::type Result;
  typedef typename class_<Pmf>::type Class;
  Bound_Callback(Event<Arg>& event, Pmf f,
                 Class* p)
    : pointer(p), function(f)
    , connection(event, this) {}
  Result operator()(Arg value) {
    return (pointer->*function)(value);
  }
private:
  Class* pointer;
  Pmf function;
  Callback::Connection<Arg> connection;
};
```

Exhibit 5: Using a meta-function.

This would be exactly what we need to implement the sort of class illustrated in Exhibit 2. As Sherlock Holmes himself might say, "Well done, Watson. Now, how can we implement the `argument<Pmf>`, `result<Pmf>` and `class_<Pmf>` meta-functions?"

Reviewing the Evidence

The `argument<Pmf>` meta-function shown in Exhibit 4 works perfectly, but only if your name is Harry Potter. Plodding detectives (and C++ compilers) can't be expected to perform magic. I was puzzled. Then I spotted something odd among the evidence:

```
template<typename Result, typename Class,
         typename Arg>
struct argument {
  typedef Arg type;
};
```

Exhibit 6: A meta-function for clairvoyants.

Here's a meta-function that extracts the parameter type without using magic. It just needs a little clairvoyance. If you know in advance what the parameter type is you can use this meta-function to generate the type you need. The heroic sleuth in detective novels may seem to be clairvoyant at times but programmers are not that clever (not even pizza-stuffed, caffeine-soaked *real* programmers).

My search for the `argument<Pmf>` meta-function had run up a blind alley. It was late. I was tired. I was getting desperate. And then it hit me. We were looking for a meta-function with one parameter (like the magical one), but to implement it we need three parameters (like the one for clairvoyants). We need a *specialisation*.

```
// Declaration of general template
template<typename Pmf> struct argument;
// Partial specialisation for pointers to
// member functions
template<typename Result, typename Class,
         typename Arg>
struct argument<Result (Class::*)(Arg)> {
  typedef Arg type;
};
```

Exhibit 7: Extracting the parameter type.

The specialisation tells the compiler how to instantiate `argument<Pmf>` when `Pmf` is a pointer to a member function of any class, taking a single parameter of any type and returning a result of any type.

The same technique works for the `result<Pmf>` and `class_<Pmf>` meta-functions, too. In each case, the general template takes one parameter, but the specialisation takes three. The compiler performs a form of pattern matching to break down a single pointer-to-member-function type into its three components. For example:

```
typedef result<
        void (Observer::*)(int)>::type Result;
Result* null_pointer = 0;  // Result is void
```

Exhibit 8: Using the `result<Pmf>` meta-function.

When it sees the `result<Pmf>` template being used the compiler compares the template argument (pointer-to-member-of-Observer) with the template parameter of the specialisation (any pointer-to-member-function). In this case the argument matches the parameter and the compiler deduces `Result = void`, `Class = Observer`, `Arg = int`. The compiler then instantiates the specialisation which defines `result<void (Observer::*)(int)>::type` as `void`.

The Case is Closed

So that's it. The crime is solved. All that's left is to prepare a case for presentation in court and let justice take its course. I've had enough for one day. "I'm off to the pub, anyone want to join me?", I called across the office.

"Well, that was the usual warm, friendly response", I thought, as I sat on my own with a pint. "No thanks", "Sorry, can't", "Too busy" they said. But something was still bothering me. Does `Bound_Callback<Pmf>` still work if we try to connect a handler function taking an `int` to an `Event` that publishes a `short`? And what if we need to connect an `Event<Arg>` to something other than a member function – like a non-member function or a function object?

These thoughts were still churning over in my mind when, sometime after midnight, I tumbled into bed and soon fell into a fitful sleep.

Phil Bass
phil@stoneymanor.demon.co.uk

References

[1] Phil Bass, "Implementing the Observer Pattern in C++", *Overload 53*, February 2003.
[2] See `www.boost.org`

C++ Interface Classes – An Introduction

by Mark Radford

Class hierarchies that have run-time polymorphism as one of their prominent characteristics are a common design feature in C++ programs, and with good design, it should not be necessary for users of a class to be concerned with its implementation details. One of the mechanisms for achieving this objective is the separation of a class's interface from its implementation. Some programming languages, e.g. Java, have a mechanism available in the language for doing this. In Java, an *interface* can contain only method signatures. In C++ however, there is no such first class language feature, and the mechanisms already in the language must be used to emulate interfaces as best as can be achieved. To this end, an *interface class* is a class used to *hoist* the polymorphic interface – i.e. pure virtual function declarations – into a base class. The programmer using a class hierarchy can then do so via a base class that communicates only the interface of classes in the hierarchy.

Example Hierarchy

The much used *shape* hierarchy example serves well here. Let's assume for the sake of illustration, that we have two kinds of shape: arc and line. The hierarchy therefore, contains three *abstractions*: the `arc` and `line` concrete classes, and the generalisation `shape`. From now on, I'll talk mainly about `shape` and `line` only – the latter serving as an illustration of an implementation. These two classes, in fragment form, look like this:

```
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;
  //...
};

class line : public shape {
public:
  virtual ~line();
  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);
private:
  point end_point_1, end_point_2;
  //...
};
```

The `shape` abstraction is expressed here as an *interface class* – it contains nothing but pure virtual function declarations. This is as close as we can get in C++ to expressing an interface. Adding to the terminology, classes such as `line` (and `arc`) are known as *implementation classes*.

Now let's assume this hierarchy is to be used in a two dimensional drawing package. It seems reasonable to suggest that in this package, `drawing` may be another useful abstraction. `drawing` could be expressed as an interface class, like in this fragment:

```
class drawing {
public:
  virtual ~drawing();
  virtual void add(shape* additional_shape)
                                                = 0;
  //...
};
```

Besides the virtual destructor, only one member function of `drawing` – the `add()` virtual function – is shown. Note that `drawing` does not collaborate with any implementation of `shape`, but only with the interface class `shape`. This is sometimes known as *abstract coupling* – `drawing` can talk to any class that supports the `shape` interface.

Benefits

Having explained the technique of *hoisting* a class's interface, I need to explain why developers should be interested in doing this. There are three points:

1 Hoisting the (common) interface of classes in a run-time polymorphic hierarchy affords a clear separation of interface from implementation. Further, doing so helps to underpin the use of abstraction, because the interface class expresses only the capabilities of the abstracted entity.

2 It follows on from the above, that new implementations can be added without changing existing code. For example, it is most likely that drawing will initially have only one implementation class, but because other code is dependent only on its interface class, new implementations *can* easily be added in the future.

3 Consider the physical structure of C++ code with regard to the interface class, its implementation classes, and classes (such as `drawing`) that use it. Assuming common C++ practice is followed, the definition of `shape` will have a header file – let's assume it's called `shape.hpp` – all to itself, as will `drawing` (i.e. `drawing.hpp`, using the same convention). Now, owing to the physical structure of C++ (that is, the structure it inherited from C), if anything in the `shape.hpp` header file is changed, anything that depends on it – such as `drawing.hpp` – must recompile. In large systems where build times are measured in hours (or even days), this can be a significant overhead. However, because `shape` is an interface class, `drawing` (for example) has no physical dependency on any of the implementation detail, and it is in the implementation detail that change is likely to occur (assuming some thought has been put into the design of `shape`'s interface).

Strengthening the Separation

Returning to the first point above for a moment, there is a way by which we can strengthen the logical separation further: we can make `shape`'s implementation classes into *implementation only* classes. This means that in the implementation classes, all the virtual member functions are made *private*, leaving only their constructors publicly accessible. The `line` class then looks like this:

```
class line : public shape {
public:
  line(point end_point_1, point end_point_2);
  //...
private:
  virtual ~line();
  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);
  //...
};
```

Now, the only thing users can do with `line` is create instances of it. All usage must be via its interface – i.e. `shape`, thus enforcing a stronger interface/implementation separation. Before leaving this topic, it is important to get something straight: the point of enforcing the interface/implementation separation is *not* to tell users what to do. Rather, the objective is to underpin the logical separation – the *code* now explains that the key abstraction is `shape`, and that `line` serves to provide an implementation of `shape`.

Mixin Interfaces

As a general design principle, all classes should have responsibilities that represent a primary design role played by the class. However, sometimes a class must also express functionality representing responsibilities that fall outside its primary design role. In such cases, the need for partitioning of this functionality is pressing, and interface classes have a part to play.

A class that expresses this kind of extra functionality is called a *mixin*. For example, it is easy to imagine there might be a requirement to store and retrieve the state of shape objects. However, storage and retrieval functionality is not a responsibility of shape in the application domain model. Therefore, a feasible design is as follows:

```
class serialisable {
public:
  virtual void load(istream& in) = 0;
  virtual void save(ostream& out) = 0;
protected:
  ~serialisable();
};

class shape : public serialisable {
public:
  virtual ~shape();
  virtual void move_x(distance x) = 0;
  virtual void move_y(distance y) = 0;
  virtual void rotate(angle rotation) = 0;
  // No declarations of load() or save() in
  // this class
  // ...
};

class line : public shape {
public:
  line(point end_point_1, point end_point_2);
  //...
private:
  virtual ~line();
  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);
  virtual void load(istream& in);
  virtual void save(ostream& out);
  //...
};
```

This approach is intrusive to a degree because `serialisable`'s virtual member functions must be declared in `line`'s interface. However, at least there is a separation in that `serialisable` is kept separate from the crucial `shape` abstraction.

Note that `serialisable` does not have a public virtual destructor – its destructor is protected and non-virtual. It is not intended that pointers to `serialisable` are held and passed around in a program – i.e. it is not a *usage* type, that's the role of the `shape` class. Making the destructor non-virtual and not publicly accessible allows the code to state this explicitly, without recourse to any further documentation.

Often mixin functionality is added to a class using multiple inheritance. Here there is an analogy with Java, in which there is direct language support for interfaces. In Java, a class can inherit from one other class, but can implement as many interfaces as desired. The same thing can be emulated in C++ using interface classes, but in C++ there is an added twist – C++ has *private* inheritance to offer. This approach comes in handy particularly when the usage type is outside the control of the programmer – for example, because it is part of a third party API. For example, consider a small framework where notifications are sent out by objects of type `notifier`, and received by classes supporting an interface defined by `notifiable`. The two interface classes (or fragment of, in the case of `notifier`) are defined as follows:

```
class notifiable {
public:
  virtual void update() = 0;
protected:
  ~notifiable();
};
class notifier {
public:
  virtual void register_client(notifiable* o)
                                              = 0;
  // ...
};
```

Now consider using a GUI toolkit that provides a base class called `window`, from which all window classes are to be derived. The programmer wishes to write a class called `my_window` that receives notifications from objects of type `notifier` – such a class could look like this:

```
class my_window : public window,
                  private notifiable {
public:
  void register_for_notifications(
                                 notifier& n) {
    n.register_client(this);
  }
  // ...
};
```

Using private inheritance has rendered the notifiable interface inaccessible to clients, but allows `my_window` use of it, because

like anything else that's private to `my_window`, its private base classes are accessible in its member functions. This approach helps to strengthen the separation of concerns which the use of mixin functionality seeks to promote.

Interface Class Emulation Issues

The fact that we have to consider emulation issues at all is owing to the fact that interfaces are being *emulated* rather than being a first class language feature – all part of the fun of using C++! I think there are issues in two areas, i.e. those concerned with:

- An interface class's interface
- Deriving from an interface class

An Interface Class's Interface

Consider the interface class `shape`:

```
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;
  // other virtual function declarations...
};
```

If we write only the above, the compiler will step in and provide: a copy assignment operator, a default constructor, and a copy constructor. I think we can safely say that, an interface class' run time polymorphic behaviour points to assignment semantics being inappropriate and irrelevant. Therefore, the assignment operator should be private and not implemented:

```
class shape {
public:
  // ...
private:
  shape& operator=(const shape&);
};
```

Interface classes are stateless by their nature, so allowing assignment is harmless, but prohibiting it is a simple contribution to avoiding errors.

What about the default constructor and a copy constructor? Here we should just thank the compiler and take what is on offer, as this is the easiest way to avoid any complications. Note that declaration of constructors by the programmer has potential pitfalls. For example, if a copy constructor only is declared, then the compiler will not generate a default constructor.

Deriving From an Interface Class

Consider the following fragment that shows `line` being derived from `shape` (as one would expect):

```
class line : public shape {
public:
  line(int in_x1, int in_y1,
       int in_x2, int in_y2)
    : x1(in_x1), y1(in_y1),
      x2(in_x2), y2(in_y2) {}
  // ...
private:
  int x1, y1, x2, y2;
};
```

The programmer has declared a constructor that initialises `line`'s state, but not specified which of `shape`'s constructors is to be called. As a result the compiler generates a call to `shape`'s default constructor. So far this is fine. Because `shape` is stateless it doesn't matter how it gets initialised.

However, that's not the end of the story ...

It is a common design re-factoring in C++ (and several other languages), to hoist common state out of concrete classes, and place it in a base class. So if common implementation is found between `shape`'s derived classes `line` and `arc`, rather than have a two tier hierarchy, it is reasonable to have a three tier hierarchy. For the sake of an example, let's assume that it is necessary for all `shape`s to maintain a *proximity rectangle* – i.e. if a point falls within the rectangle, the point is considered to be in close proximity to the `shape`. This functionality can then, for example, be used to determine if a `shape` object should be selected when the user clicks the mouse near by.

I'm going to assume a suitable `rectangle` class is in scope, and introduce `shape_impl` to contain the common implementation.

```
class shape_impl : public shape {
private:
  virtual ~shape_impl() = 0;
  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);
  //...
protected:
  shape_impl();
  shape_impl(
          const rectangle& initial_proximity);
  //...
private:
  rectangle proximity;
  // ...
};
```

The implementation class `shape_impl` is abstract, as shown by the pure virtual destructor. As a brief digression, it is also an *implementation only* class – its implementation of `shape`'s interface has been declared as private so clients can create instances, but can't call any of the member functions.

Now look what happens if `line`'s base class is changed, but changing the constructor used to initialise the base class gets forgotten about.

```
class line : public shape_impl {
public:
  line(int in_x1, int in_y1,
       int in_x2, int in_y2)
    : x1(in_x1), y1(in_y1),
      x2(in_x2), y2(in_y2) {}
  // ...
private:
  int x1, y1, x2, y2;
};
```

From Mechanism to Method: The Safe Stacking of Cats

by Kevlin Henney

In spite of some obvious differences – and the similarity that neither can be considered a normal practice – curling and throwing have something in common: curling is a bizarre sport played on ice; throwing in C++ is often played on thin ice. It is the thin ice that has most often caused consternation, and the balanced art of not falling through that has attracted much attention.

By coincidence, curling is also something in which cats both indulge and excel, putting the *pro* into *procrastination*. But more on cats later.

Taking Exception

Exceptions are disruptive but modular. The common appeal to consider them as related to the `goto` is more than a little misleading ("considering `goto`" considered harmful, if you like). That they are both discontinuous is one of the few features they share. It is an observation that although true is not necessarily useful: `break`, `continue`, and `return` also share this description of behavior. A quick dissection exposes the differences:

- **Transferred information:**
  a `goto` can associate only with a label whereas a `throw` communicates with respect to both type and any information contained in the type instance. In this sense, the `throw` acts more like a `return`, communicating an exceptional rather than a normal result.

- **Locality:**
  a `goto` has meaning only within a function, labels being the only C++ entity with function scope. By contrast, exception handling is primarily about transferring control out of a function. It shares this with `return`, but potentially has the whole of the call stack rather than just the immediate caller within its reach. It also shares with `break` and `continue` a relationship with an enclosing control flow primitive, so exception handling can also be used simply at the block level.

- **Destination coupling:**
  the target of a `goto` is fixed, hardwired at compile time. There is no way to express "the following has happened, so whoever can sort it out, please sort it out." Exceptions are independent of lexical scope and do not nominate their handlers explicitly. Instead, nomination is dynamic and by requirement – "the first one that can handle one of these gets to sort it out." Exceptions can be handled or ignored at different call levels without intervention from any of the levels in between. In many ways, the `try`/`catch` mechanism resembles an advanced selection control structure – an `if`/`else` with extreme attitude.

- **Block structure:**
  Taligent's Guide to Designing Programs pulls no punches in stating that *"a `goto` completely invalidates the high-level structure of the code"* [1]. Far from being merely a provocative statement, this is a concise summary of fact. C++ is essentially block structured: exceptions respect and work within this structure, whereas `goto`s largely ignore and disrespect it.

The differences are thrown (sic) into sharp relief when you attempt to refactor code. Say that you wish to factor a block out as a function (the Extract Method refactoring [2]); it is trivial to factor out the data flow: looking at the data that's used and affected in the block tells you what arguments and results you need to pass and how. With control flow, unless you flow off the bottom of a block or `throw`, you cannot factor the code simply. Traditional discontinuous control flow is non-modular and requires additional restructuring to communicate to the caller that a `break`, `return`, `continue`, or `goto` (especially) must be effected. This is not the case with `throw`: both the overall structure and the fine-grained detail remain unchanged.

State Corruption

This all sounds straightforward – or straight backward if you take the call stack's perspective – because we know about modularity, both structured programming and object-oriented programming are built on that foundation. However, there is still that one small

---

This will compile, and fail at run time. However, if in the first place the programmer had written:

```
class line : public shape {
public:
  line(int in_x1, int in_y1,
       int in_x2, int in_y2)
    : shape(), x1(in_x1), y1(in_y1),
      x2(in_x2), y2(in_y2) {}
  // ...
};
```

In the latter case, changing the base class to `shape_impl` would cause a compile error, because `shape` is no longer the immediate base class. This leads me to make the following recommendation: *always call an interface class's constructor explicitly*.

Finally

Interface classes are fundamental to programming with run time polymorphism in C++. Despite this, I'm all too frequently surprised by how little they are known about by the C++ programmers out there.

This article doesn't cover everything: for example, the use of virtual inheritance when deriving from mixins is something I hope to get around to covering in a future article. However, I hope this article serves as a reasonable introduction.

Mark Radford
mark@twonine.co.uk

Acknowledgements

Many thanks to Phil Bass, Thaddaeus Frogley and Alan Griffiths for their feedback.

matter of "disruption." When an exception is thrown, the only thing you want disrupted is the control flow, not the integrity of the program.

Any block of code may be characterized with respect to the program's stability in the event of an exception. We can guarantee different levels of safety, of which three are commonly recognized [3], plus the (literally) degenerate case:

- **No guarantee of exception safety:**
  ensures disruption, corruption, and chaos. Code written without exceptions in mind often falls into this category, leaking memory or leaving dangling pointers in the event of a thrown exception – converting the exceptional into the unacceptable. In short, all bets are off.

- **Basic guarantee of exception safety:**
  ensures that the thrower will not leak or corrupt resources. Objects involved in the execution will be in a stable and usable, albeit not necessarily predictable, state.

- **Strong guarantee of exception safety:**
  ensures that a program's state remains unchanged in the presence of exceptions. In other words, commit-rollback semantics.

- **No-throw guarantee of exception safety:**
  ensures that exceptions are never thrown, hence the question of how program state is affected in the presence of an exception need never be answered because it is purely hypothetical.

The stroke of midnight separates the first, degenerate category of exception unsafety from the last, Zen-like guarantee of benign order through the simple absence of disruption. Code written to achieve these guarantees may have the same structure, but will differ in the not-so-small detail of whether or not exceptions occur anywhere in their flow.

These guarantees apply to any unit of code from a statement to a function, but are most commonly applied to member functions called on objects. A point that is not often made relates exception safety to encapsulation: not so much that exception safety can be improved by better encapsulation, but that exception safety is one measure of encapsulation. Prominent OO propaganda holds that encapsulation is concerned with making object data private. Whilst this view is not strictly false, it misses some important truths.

Encapsulation is neither a language feature nor a practice; rather it is a non-functional property of code, and something that you can have more or less of. Encapsulation describes the degree to which something is self-contained, the degree to which its insides affect its outsides, the degree to which internal representation affects external usage. Encapsulation is about usability, about not imposing on the user. Language features and idiomatic design practices can be used to improve encapsulation, but of themselves they are not encapsulation. Thinking back to exceptions, you can see that without even thinking about internal representation, an object that offers the strong guarantee on a member function is more encapsulated than one that offers no guarantee.

Incorruptible Style

It is one thing to have a guarantee, but quite another to fulfill it. What is the style and mechanism of the code that allows a thrown exception to propagate out of a block in a safe manner? Including the degenerate case, there are essentially four approaches to achieving exception safety:

- **Exception-unaware code:**
  code that is not written with exceptions in mind is as easy to read as it is dangerous – going wrong with confidence.

- **Exception-aware code:**
  code may be scaffolded explicitly with `try`, `catch`, and `throw` to ensure that the appropriate stabilizing action is taken in the event of a thrown exception. Alas, it is not always obvious that exception-aware code is safe: such code is rarely clear and concise.

- **Exception-neutral code:**
  code that works in the presence of exceptions, but does not require any explicit exception-handling apparatus to do so (i.e., no explicit `try`/`catch` code). Not only is exception-neutral code briefer and clearer than exception-aware code, but it is also typically shorter than exception unaware code. So, exception safety and seamless exception propagation aside, such minimalism offers another strong motivation for reworking code in this style.

- **Exception-free code:**
  code that generates no exceptions offers the most transparent fulfillment of exception safety.

When Cats Turn Bad

There is a tradition – from Schrödinger to Stroustrup – of employing cats for demonstration purposes, and I see no reason to stand in the way of tradition. There appears to be sufficient prior art in the stacking of cats [4] that I will also adopt that practice. Of course, we are only dealing with abstractions – if you are concerned for the poor cat, keep in mind that unless we set it in concrete no act of cruelty actually occurs.

Assuming that we have an appropriate `cat` class definition, the following fragment demonstrates exception-unaware code:

```
{
  cat *marshall = new cat;
  .... // play with marshall
  delete marshall;
}
```

If an exception occurs during play, there will be a memory leak: you will forget about your scoped `cat`. The following fragment demonstrates exception-aware code:

```
{
  cat *marshall = new cat;
  try {
    .... // play with marshall
  }
  catch(...) {
    delete marshall;
    throw;
  }
  delete marshall;
}
```

Safe? Yes. Unreadable? Certainly. What it lacks in elegance it more than makes up for in verbosity. The code may be safe, but it is not obviously so [5]. The following fragment demonstrates exception-neutral code:

```
{
  std::auto_ptr<cat> marshall(new cat);
  .... // play with marshall
}
```

For all its faults (and they are many), this is one job that `std::auto_ptr` does do well. If we know that default `cat` constructors do not throw exceptions, and we recognize that `marshall` is always bounded by scope, the following fragment demonstrates exception-free code:

```
{
  cat marshall;
  .... // play with marshall
}
```

Clearly, for demo purposes, we are taking some liberties with the common understanding of cats and their care, treating them as disposable commodities. Taking further license with feline appreciation and object design, let us also assume that they are value-based rather than entity-based objects. This means that they support copying through construction and assignment, are generally not heap based, and are typically not deeply involved in class hierarchies.

Modern cloning technology is imperfect, so cat copy constructors are not always guaranteed to work. On failure they throw an exception, but they are well behaved enough to avoid resource leakage and to not corrupt the program's state.

Throwing Gauntlets

In 1994 Tom Cargill laid down a challenge – or extended an invitation to solution, depending on your point of view – concerning exception safety [6]. The challenge was based on a fairly typical stack class template. There were a number of elements to the challenge; the one I want to focus on here is how to write the `pop` member function.

Here is some code that demonstrates the challenge:

```
template<typename value_type>
class stack {
public:
  void push(const value_type &new_top) {
    data.push_back(new_top);
  }
  value_type pop() {
    value_type old_top = data.back();
    data.pop_back();
    return old_top;
  }
  std::size_t size() const {
    return data.size();
  }
  ....
private:
  std::vector<value_type> data;
};
```

I have used `std::vector` for brevity (performing manual memory management does nothing to make the problem clearer) and I am skipping issues related to assignment – I would recommend looking at Herb Sutter's thorough coverage of the challenge to see how this is addressed [3].

We can now recruit our favorite cat to demonstrate the issue. First of all, pushing cats is not problematic:

```
stack<cat> stacked;
stacked.push(marshall);
std::cout << "number of stacked cats == "
          << stacked.size() << std::endl;
```

The issue arises when we pop cats:

```
try {
  cat fender = stacked.pop();
  .... // play with fender
}
catch(...) {
  std::cout << "number of stacked cats"
            << " == " << stacked.size()
            << std::endl;
}
```

If the copy made in `pop`'s return statement fails, we have lost the top cat: the cat has been removed from `data` and `size` is one less than before. `pop`, therefore, cannot satisfy the strong guarantee of exception safety, because that requires everything to be left as it was before the exception was thrown. The stack is still usable and its resulting state is predictable, which means that we can promise marginally more than the basic guarantee ... but we've still got a missing cat.

Before setting about any solution, it is important to remember that designs – and therefore design problems – do not exist in a vacuum. Design is intimately bound up with purpose and context, and without understanding these we risk either solving the wrong problem or, as we so often do, solving the solution. Design is about balancing goals – as well as cats.

Unasking the Question

Looking at the class interface, we might ask why two actions are combined into one: Why does `pop` both return a queried value and modify the target object? We know that such a return causes an exception-safety problem, and we also know that it is potentially wasteful. What if you do not plan to use the return value? Even if you ignore it, the work that goes into copying and returning the value still happens. You are potentially paying both a cost and a penalty for something you didn't use.

The Command-Query Separation pattern [7] – sometimes referred to as a *principle* rather than a *pattern* [8] – resolves our concerns by making a separation with respect to qualification:

```
template<typename value_type>
class stack {
public:
  ....
  void pop() {
    data.pop_back();
  }
```

```
  value_type &top() {
    return data.back();
  }
  const value_type &top() const {
    return data.back();
  }
  ....
private:
  std::vector<value_type> data;
};
```

The separation of modifier from query operations ensures that we cannot make a change and lose the result. This separated interface also supports a slightly different usage model:

```
cat fender = stacked.top();
stacked.pop();
.... // play with fender
```

No copying exception can arise within the stack, so there is no need to deal with it. This separation of concerns (and member functions) can be seen in the design of the STL sequences and sequence adaptors.

Rephrasing the Question

It would seem that the problem is solved, except for one thing: we never fully established the context of execution. It is entirely possible that the basic guarantee of the original code was satisfactory for our purposes, so there was no problem – from our perspective – to be solved. Either we accept the loss of a cat or, more commonly, the element type of the stack has exception-free copying, which would be the case for built-in types as well as a number of user-defined types. So under some circumstances, the stack offers us the strong guarantee. If these are your circumstances, the original code does not strictly speaking need to be fixed. If they are not, there is indeed a problem to be fixed, and Command-Query Separation offers one solution.

But there are others. Command-Query Separation is attractive because it clarifies the role of interface functions. It could be said to offer better encapsulation and cohesion. However, such a statement is not universally true, and understanding why will demonstrate why we must consider Command-Query Separation a pattern (a design solution with consequences and a context) and not a principle (an idea that expresses a universal truth).

Consider a clarification in design context: the stack is to be shared between multiple threads. Ideally we would like to encapsulate synchronization detail within the stack, ensuring that primitives such as mutexes are used safely and correctly. Focusing just on the `push` member, an exception-unaware implementation would be as follows:

```
template<typename value_type>
class stack {
public:
  ....
  void push(const value_type &new_top) {
    guard.lock();
    data.push_back(new_top);
    guard.unlock();
  }
  ....
private:
  mutable mutex monitor;
  std::vector<value_type> data;
};
```

The exception-neutral approach is both shorter and safer:

```
template<typename value_type>
class stack {
public:
  ....
  void push(const value_type &new_top) {
    locker<mutex> guard(monitor);
    data.push_back(new_top);
  }
  ....
private:
  mutable mutex monitor;
  std::vector<value_type> data;
};
```

Where `locker` is a helper class template responsible for abstracting control flow [9]:

```
template<typename locked_type>
class locker {
public:
  explicit locker(locked_type &lockee)
    : lockee(lockee) {
    lockee.lock();
  }
  ~locker() {
    lockee.unlock();
  }
private:
  locker(const locker &); // no copying
  locked_type &lockee;
};
```

Making each public member function of `stack` self-locking would appear to preserve encapsulation. However, this works only for usage scenarios that are based on single function calls. For the Command-Query Separation solution, this would introduce a couple of subtle bugs:

```
cat fender = stacked.top();
stacked.pop();
.... // play with fender
```

First of all, `top` returns a reference. Consider the following simple action in another concurrent thread:

```
stacked.pop();
```

Assuming that all of the member functions we are talking about are self-locking, what is the problem? Imagine that the second thread executes `pop` just after the first thread completes the call to `top`: the reference result from `top` is now dangling, referring to a non-existent element. Undefined behavior. Oops. Poor `fender` gets a very bad start in life.

Returning references to value objects from thread-shared objects is a bad idea, so let's fix `stack`:

```
template<typename value_type>
class stack {
public:
  ....
  value_type top() const {
    locker<mutex> guard(monitor);
    return data.back();
  }
  ....
private:
  mutable mutex monitor;
  std::vector<value_type> data;
};
```

This solves the problem of undefined behavior, but leads us straight into the jaws of the second problem, which is that of "surprising" behavior. Idiomatically, we treat the following as a single unit:

```
cat fender = stacked.top();
stacked.pop();
.... // play with fender
```

However, this usage is not cohesive in its execution. It can be interrupted by another thread:

```
cat peavey;
stacked.push(peavey);
```

so that the `push` in the second thread occurs between the initialization of `fender` and the `pop` in the first thread. This means that the wrong element is popped from the stack. Oops, again.

We could expose the `lock` and `unlock` features in `stack` and let the user sort it all out:

```
template<typename value_type>
class stack {
public:
  void lock() {
    monitor.lock();
  }
  void unlock() {
    monitor.unlock();
  }
  ....
private:
  mutex monitor;
  std::vector<value_type> data;
};
```

Giving rise to the following somewhat clunky usage:

```
cat fender; {
  locker< stack<cat> > guard(stacked);
  fender = stacked.top();
  stacked.pop();
}
.... // play with fender
```

Let's compare this with the original usage:

```
cat fender = stacked.pop();
.... // play with fender
```

There's now more to write and more to remember – and therefore more to forget. In addition to being more tedious and error prone, it is easy to make the code pessimistic by forgetting to enclose the `locker` in the narrowest scope possible, leaving waiting threads `locked` out of `stacked` for far longer than necessary.

Remember that the original design's only safety shortcoming was that it offered only the basic – rather than the strong – guarantee of exception safety. It would take a leap of orthodoxy to say, hand on heart, that Command-Query Separation has produced a more cohesive and encapsulated solution – the opposite is true in this context.

The Combined Method pattern [7] is one that sometimes finds itself in tension with Command-Query Separation, combining separate actions into a single, transactional whole for the benefit of simplicity and correctness in, principally, multithreaded environments. The original `pop` was an example of this tactical pattern, but suffered from weakened exception safety. An alternative realization that achieves strong exception safety in an exception-neutral style is to overload the pure `pop` function with a Combined Method that takes a result argument:

```
template<typename value_type>
class stack {
public:
  ....
  void pop() {
    locker<mutex> guard(monitor);
    data.pop_back();
  }
  void pop(value_type &old_top) {
    locker<mutex> guard(monitor);
    old_top = data.back();
    data.pop_back();
  }
  ....
private:
  mutable mutex monitor;
  std::vector<value_type> data;
};
```

This design tightens the screws a little on the element type requirements, additionally requiring assignability as well as copy constructibility. In practice this often means that we also demand default constructibility of the target because the overloaded `pop` cannot be used in an assignment:

```
cat fender;
stacked.pop(fender);
.... // play with fender
```

Another consequence of the assignment-based approach is that the result variable must be an exact type match for the element type (i.e., it cannot rely on implicit conversions that would have worked if `pop`'s result had been returned by value).

A Transactional Approach

Staying with Combined Method, but for brevity leaving aside the code for thread synchronization, it turns out that it is possible to write an exception-neutral version of `pop` that preserves the original value-returning interface and satisfies the strong guarantee of exception safety in slightly different circumstances to the original:

```
template<typename value_type>
class stack {
public:
  ....
  value_type pop() {
    popper karl(data);
    return data.back();
  }
  ....
private:
  class popper {
  public:
    popper(std::vector<value_type> &data)
      : data(data) {}
    ~popper() {
      if(!std::uncaught_exception())
        data.pop_back();
    }
  private:
    popper(const popper &);
    std::vector<value_type> &data;
  };
  std::vector<value_type> data;
};
```

Here a small helper object, `karl`, is created to commit a `pop` action if the copying of the return value is successful. The `popper` object is passed the representation of the surrounding stack, and on destruction, it will cause a `pop_back` to be executed. If the copy is unsuccessful, the `popper` destructor will not commit the intended change, skipping the `pop_back`.

This approach has the benefit of preserving the signature interface and typically reducing the number of temporaries involved in copying. However, there is an important precondition that must be publicized and satisfied for `popper` to work as expected: `pop` should not be called from the destructor of another object. Why? What if the destructor is being called because the stack is being unwound by an exception? The call to `std::uncaught_exception` in `popper`'s destructor will return true even if the copy is successful.

How you respond to this scenario is a matter of context-driven requirement. Either you state that the behavior of a `stack` is undefined in these circumstances or you define behavior for it. One definition of behavior is shown above – in the presence of existing exceptions, don't pop – but could be considered unsatisfactory because of its pessimism. An alternative, more optimistic approach is to say that our `pop` offers a strong guarantee of exception safety if there is no unhandled exception present when it is executed, but only the basic guarantee otherwise:

```
template<typename value_type>
class stack {
  ....
  class popper {
  public:
    popper(std::vector<value_type> &data)
      : data(data),
        unwinding(
            std::uncaught_exception()) {}
    ~popper() {
      if(unwinding
         || !std::uncaught_exception())
        data.pop_back();
    }
  private:
    popper(const popper &);
    std::vector<value_type> &data;
    const bool unwinding;
  };
  ....
};
```

`std::uncaught_exception` is a function that is generally not as useful as it first appears. It often leads to false confidence in code [10], but with an understanding of its limitations, there are a few situations in which we can press it into useful service.

A Lazy Approach

It is possible to take the transactional idea a step further using a technique that I first saw Angelika Langer present at *C++ World* in 1999:

```
template<typename value_type>
class stack {
public:
  ....
  value_type pop() {
    try {
      --length;
      return data[length];
    }
    catch(...) {
      ++length;
      throw;
    }
  }
  ....
private:
  std::size_t length;
  std::vector<value_type> data;
};
```

Here the size of the stack is tracked in a separate variable that is incremented and decremented accordingly. It uses an exception-aware style to implement commit-rollback semantics, bumping the length count back up again if the copy from the last element fails with an exception.

The obvious benefit of this approach is that it will work independently of whether or not the stack is already unwinding

because of an exception. However, the disadvantage with this approach is not so much with the extra piece of state that has been introduced but that popped elements are never actually popped. They continue to exist in the `data` member long after they have been popped: at least up until another modification operation requires rationalization of `data` with `length`, such as a `push`. A couple of minor refinements address this issue by introducing a deferred but amortized commit operation:

```
template<typename value_type>
class stack {
public:
  stack()
    : uncommitted(false) {}
  void push(const value_type &new_top) {
    commit();
    data.push_back(new_top);
  }
  value_type pop() {
    commit();
    try {
      uncommitted = true;
      return data.back();
    }
    catch(...) {
      uncommitted = false;
      throw;
    }
  }
  std::size_t size() const {
    commit();
    return data.size();
  }
  ....
private:
  void commit() const {
    if(uncommitted) {
      data.pop_back();
      uncommitted = false;
    }
  }
  mutable bool uncommitted;
  mutable std::vector<value_type> data;
};
```

Internally the committed state will be at most one element different from the uncommitted state, but externally any attempt to determine the state by calling an operation will ensure that the books are kept balanced. This constraint requires that all public functions call the `commit` function as their first act, which requires that the object's state to be qualified as `mutable` to permit updates in query functions. Thus, this design affects all member functions and imposes a little more on the class developer. The class user is, however, unaffected.

Conclusion

It is time to declare a moratorium on these exceptional experiments on abstracted cats. They have served to demonstrate that no design can be perfect, and that encapsulation is related to usability; it is not just a matter of data hiding. Although we may strive for absolute recommendations, there are times when only relative ones can be made with confidence (and caveats). Design is about compromise and about context, and therefore it is about understanding consequences. Weigh up the benefits and liabilities for a particular usage and then make your decision – what is workable in one context may be unworkable in another, and so what is "good" in one situation may be "bad" in another.

On the compromise of design in other fields I will leave you with this quote from David Pye [11]:

> *It follows that all designs for use are arbitrary. The designer or his client has to choose in what degree and where there shall be failure. Thus the shape of all design things is the product of arbitrary choice. If you vary the terms of your compromise – say, more speed, more heat, less safety, more discomfort, lower first cost-then you vary the shape of the thing designed. It is quite impossible for any design to be "the logical outcome of the requirements" simply because, the requirements being in conflict, their logical outcome is an impossibility.*

Kevlin Henney
kevlin@curbralan.com

References

[1] *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*, (Addison-Wesley, 1994), http://pcroot.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_1.html

[2] Martin Fowler. *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999), www.refactoring.com

[3] Herb Sutter. *Exceptional C++* (Addison-Wesley, 2000).

[4] Bjarne Stroustrup. "Sixteen Ways to Stack a Cat," *C++ Report*, October 1990, www.research.att.com/~bs

[5] To quote C. A. R. Hoare:
"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."

[6] Tom Cargill. "Exception Handling: A False Sense of Security," *C++ Report*, November-December 1994.

[7] Kevlin Henney. "A Tale of Two Patterns," *Java Report*, December 2000, www.curbralan.com

[8] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition* (Prentice Hall, 1997).

[9] Kevlin Henney. "C++ Patterns: Executing Around Sequences," *EuroPLoP 2000*, July 2000, www.curbralan.com

[10] Herb Sutter. *More Exceptional C++* (Addison-Wesley, 2002).

[11] Henry Petroski. *To Engineer is Human: The Role of Failure in Successful Design* (Vintage, 1992).

This article was originally published on the C/C++ Users Journal C++ Experts Forum in February 2002 at http://www.cuj.com/experts/documents/s=7986/cujcexp2002Henney/
Thanks to Kevlin for allowing us to reprint it.