ISSN 1354-3172

Overload

Journal of the ACCU C++ Special Interest Group

Issue 25

April 1998

Contents

Editorial

Treats

We've got such a vast array of treats for you this issue that I've been squeezed off my usual page. Amongst the glittering delights are:

- The Colour Cover.
- Kevlin's revelations whilst skip diving over the Christmas holidays.
- An article from Bjarne Stroustrup on C++2000 features.
- And, the unveiling of the ACCU Overload web pages.

Web Pages

The Overload web pages are now online and available for your enjoyment. Slightly sparse, but we hope to expand its breadth as each issue is published. At present there's a bibliography of articles from past issues, and we'll be posting some complete issues, articles, and source code in the future.

If you're interested in being the Overload web-meister then please contact us. It'd be a great excuse to learn about web publishing technologies.

Submissions

As ever, we are always looking for contributions from new authors, and are particularly interested in shorter pieces of work. With many 4 and 5 page articles it becomes tricky getting just the right number of pages. The occasional one or two page text really helps.

We'd be particularly interested to hear of your experiences deploying patterns in your current project. Have you found the Design Patterns book useful?

Ray Hall would like to hear from you if you are interested in reviewing books. Overload book reviews tend to have more depth than their C-Vu counterparts, and we'd like to keep that differentiation. In fact, a survey of books covering a single topic would be of interest to many. Perhaps Patterns, yet again, or the under explored STL.

Copy Deadline

All articles intended for publication in *Overload 26* should be submitted to the editor, by May 5th.

John Merrells
merrells@netscape.com

Software Development in C++

Counted Body Techniques
By Kevlin Henney

Reference counting techniques? Nothing new, you might think. Every good C++ text that takes you to an intermediate or advanced level will introduce the concept. It has been explored with such thoroughness in the past that you might be forgiven for thinking that everything that can be said has been said. Well, let's start from first principles and see if we can unearth something new....

And then there were none...

The principle behind reference counting is to keep a running usage count of an object so that when it falls to zero we know the object is unused. This is normally used to simplify the memory management for dynamically allocated objects: keep a count of the number of references held to that object and, on zero, delete the object.

How to keep a track of the number of users of an object? Well, normal pointers are quite dumb, and so an extra level of indirection is required to manage the count. This is essentially the PROXY pattern described in *Design Patterns* [Gamma, Helm, Johnson & Vlissides, Addison-Wesley, ISBN 0-201-63361-2]. The intent is given as

Provide a surrogate or placeholder for another object to control access to it.

Coplien [*Advanced C++ Programming Styles and Idioms*, Addison-Wesley, ISBN 0-201-56365-7] defines a set of idioms related to this essential separation of a handle and a body part. The *Taligent Guide to Designing Programs* [Addison-Wesley, ISBN 0-201-40888-0] identifies a number of specific categories for proxies (aka surrogates). Broadly speaking they fall into two general categories:

Hidden: The handle is the object of interest, hiding the body itself. The functionality of the handle is obtained by delegation to the body, and the user of the handle is unaware of the body. Reference counted strings offer a transparent optimisation. The body is shared between copies of a string until such a time as a change is needed, at which point a copy is made. Such a COPY ON WRITE pattern (a specialisation of LAZY EVALUATION) requires the use of a hidden reference counted body.

Explicit: Here the body is of interest and the handle merely provides intelligence for its access and housekeeping. In C++ this is often implemented as the SMART POINTER idiom. One such application is that of reference counted smart pointers that collaborate to keep a count of an object, deleting it when the count falls to zero.

Attached vs detached

For reference counted smart pointers there are two places the count can exist, resulting in two different patterns, both outlined in *Software Patterns* [Coplien, SIGS, ISBN 0-884842-50-X]:

COUNTED BODY or ATTACHED COUNTED HANDLE/BODY places the count within the object being counted. The benefits are that countability is a part of the object being counted, and that reference counting does not require an additional object. The drawbacks are clearly that this is intrusive, and that the space for the reference count is wasted when the object is not heap based. Therefore the reference counting ties you to a particular implementation and style of use.

DETACHED COUNTED HANDLE/BODY places the count outside the object being counted, such that they are handled together. The clear benefit of this is that this technique is completely unintrusive, with all of the intelligence and support apparatus in the smart pointer, and therefore can be used on classes created independently of the reference counted pointer. The main disadvantage is that frequent use of this can lead to a proliferation of small objects, i.e. the counter, being created on the heap.

Even with this simple analysis, it seems that the DETACHED COUNTED HANDLE/BODY approach is ahead. Indeed, with the increasing use of templates this is often the favourite, and is the principle behind the common – but not standard – `counted_ptr`.

A common implementation of COUNTED BODY is to provide the counting mechanism in a base class that the counted type is derived from. Either that, or the reference counting mechanism is provided anew for each class that needs it. Both of these approaches are unsatisfactory because they are quite closed, coupling a class into a particular framework. Added to this the non-cohesiveness of having the count lying dormant in a non-counted object, and you get the feeling that excepting its use in widespread object models such as COM and CORBA the COUNTED BODY approach is perhaps only of use in specialised situations.

A requirements based approach

It is the question of openness that convinced me to revisit the problems with the COUNTED BODY idiom. Yes, there is a certain degree of intrusion expected when using this idiom, but is there anyway to minimise this and decouple the choice of counting mechanism from the smart pointer type used?

In recent years the most instructive body of code and specification for constructing open general purpose components has been the Stepanov and Lee's STL (Standard Template Library), now part of the C++ standard library. The STL approach makes extensive use of compile time polymorphism based on well defined operational requirements for types. For instance, each container, contained and iterator type is defined by the operations that should be performable on an object of that type, often with annotations describing additional constraints. Compile time polymorphism, as its name suggests, resolves functions at compile time based on function name and argument usage, i.e. overloading. This is less intrusive, although less easily diagnosed if incorrect, than runtime poymorphism that is based on types, names and function signatures.

This requirements based approach can be applied to reference counting. The operations we need for a type to be *Countable* are loosely:

An `acquire` operation that registers interest in a *Countable* object.

A `release` operation unregisters interest in a *Countable* object.

An `acquired` query that returns whether or not a *Countable* object is currently acquired.

A `dispose` operation that is responsible for disposing of an object that is no longer acquired.

Note that the count is deduced as a part of the abstract state of this type, and is not mentioned or defined in any other way. The openness of this approach derives in part from the use of global functions, meaning that no particular member functions are implied; a perfect way to wrap up an existing counted body class without modifying the class itself. The other aspect to the openness comes from a more precise specification of the operations.

For a type to be *Countable* it must satisfy the following requirements, where `ptr` is a non-null pointer to a single object (i.e. not an array) of the type, and `#`*`function`* indicates number of calls to *`function`*`(ptr)`:

Expression	
`Acquire(ptr)`	no requirement for return type post: `acquired(ptr)`
`Release(ptr)`	no requirement for return type pre: `acquired(ptr)` post: `acquired(ptr) == #acquire >`

	#release
Acquired(ptr)	Return type convertible to bool return: #acquire > #release
Dispose (ptr, ptr)	no requirement for return type pre: !acquired(ptr) post: *ptr no longer usable

Note that the two arguments to dispose are to support selection of the appropriate type safe version of the function to be called. In the general case the intent is that the first argument determines the type to be deleted, and would typically be templated, while the second selects which template to use, e.g. by conforming to a specific base class.

In addition the following requirements must also be satisfied, where null is a null pointer to the *Countable* type:

Expression	Semantics and notes
acquire(null)	No requirement for return type action: none
release(null)	No requirement for return type action: none
acquired(null)	Return type convertible to bool return: false
Dispose (null, null)	No requirement for return type action: none

Note that there are no requirements on these functions in terms of exceptions thrown or not thrown, except that if exceptions are thrown the functions themselves should be exception safe.

Getting smart

Given the *Countable* requirements for a type, it is possible to define a generic smart pointer type that uses them for reference counting:

```
template<typename countable_type>
class countable_ptr
{
public: // construction and destruction

 explicit countable_ptr(countable_type*);
 countable_ptr(const countable_ptr &);
 ~countable_ptr();

public: // access

 countable_type *operator->() const;
 countable_type &operator*() const;
 countable_type *get() const;

public: // modification

 countable_ptr &clear();
 countable_ptr &assign(countable_type *);
 countable_ptr &assign
                  (const countable_ptr&);
 countable_ptr &operator=
                 (const countable_ptr &);

private: // representation

 countable_type *body;
};
```

The interface to this class has been kept intentionally simple, e.g. member templates and throw specs have been omitted, for exposition. The majority of the functions are quite simple in implementation, relying very much on the assign member as a keystone function:

```
template<typename countable_type>
countable_ptr<countable_type>::
countable_ptr(countable_type *initial)
  : body(initial)
{
  acquire(body);
}

template<typename countable_type>
countable_ptr<countable_type>::
countable_ptr
  (const countable_ptr &other)
  : body(other.body)
{
  acquire(body);
}

template<typename countable_type>
countable_ptr<countable_type>::
~countable_ptr()
{
  clear();
}
```

```
template<typename countable_type>
countable_type
*countable_ptr<countable_type>::
operator->() const
{
  return body;
}

template<typename countable_type>
countable_type
&countable_ptr<countable_type>::
operator*() const
{
  return *body;
}

template<typename countable_type>
countable_type
*countable_ptr<countable_type>::
get() const
{
  return body;
}

template<typename countable_type>
countable_ptr<countable_type>
&countable_ptr<countable_type>::
clear()
{
  return assign(0);
}

template<typename countable_type>
countable_ptr<countable_type> &
countable_ptr<countable_type>::
assign(countable_type *rhs)
{
  // set to rhs (this sequence
  // is self assignment safe)
  acquire(rhs);
  countable_type *old_body = body;
  body = rhs;

  // tidy up
  release(old_body);
  if(!acquired(old_body))
  {
    dispose(old_body, old_body);
  }

  return *this;
}

template<typename countable_type>
countable_ptr<countable_type> &
countable_ptr<countable_type>::
assign(const countable_ptr &rhs)
{
  return assign(rhs.body);
}

template<typename countable_type>
countable_ptr<countable_type> &
countable_ptr<countable_type>::
operator=(const countable_ptr &rhs)
{
  return assign(rhs);
}
```

Public accountability

Conformance to the requirements means that a type can be used with `countable_ptr`. Here is an implementation mix-in class (*mix-imp*) that confers countability on its derived classes through member functions. This class can be used as a class adaptor:

```
class countability
{
public: // manipulation

  void acquire() const;
  void release() const;
  size_t acquired() const;

protected:
  // construction and destruction

  countability();
  ~countability();

private: // representation

  mutable size_t count;

private: // prevention

  countability(const countability &);
  countability &operator=
               (const countability &);
};
```

Notice that the manipulation functions are `const` and that the `count` member itself is `mutable`. This is because countability is not a part of an object's abstract state: memory management does not depend on the `const`-ness or otherwise of an object. I won't include the definitions of the member functions here as you can probably guess them: increment, decrement and return the current count, respectively for the manipulation functions. In a multithreaded environment you should ensure that such read and write operations are atomic.

So how do we make this class *Countable*? A simple set of forwarding functions does the job:

```
void acquire(const countability *ptr)
{
  if(ptr)
  {
    ptr->acquire();
  }
}

void release(const countability *ptr)
{
  if(ptr)
  {
    ptr->release();
  }
}

size_t acquired(const countability *ptr)
{
  return ptr ? ptr->acquired() : 0;
}

template<class countability_derived>
void dispose
  (const countability_derived *ptr,
   const countability *)
{
  delete ptr;
}
```

Any type that now derives from `countability` may now be used with `countable_ptr`:

```
class example : public countability
{
  ...
};

void simple()
{
  countable_ptr<example> ptr(new
example);
  countable_ptr<example> qtr(ptr);
  // set ptr to point to null
  ptr.clear();
  // allocated object deleted when
  //qtr destructs
}
```

Runtime mixin

The challenge is to apply COUNTED BODY in a non-intrusive fashion, such that there is no overhead when an object is not counted. What we would like to do is confer this capability on a per object rather than on a per class basis. Effectively we are after *Countability* on any object, i.e. anything pointed to by a `void *`! It goes without saying that `void` is perhaps the least committed of any type.

The forces to resolve on this are quite interesting, to say the least. Interesting, but not insurmountable. Given that the class of a runtime object cannot change dynamically in any well defined manner, and the layout of the object must be fixed, we have to find a new place and time to add the counting state. The fact that this must be added only on heap creation suggests the following solution:

```
struct countable_new;
extern const countable_new countable;

void *operator new
        (size_t, const countable_new &);
void operator delete
        (void *, const countable_new &);
```

We have overloaded `operator new` with a dummy argument to distinguish it from the regular global `operator new`. This is comparable to the use of the `std::nothrow_t` type and `std::nothrow` object in the standard library. The placement `operator delete` is there to perform any tidy up in the event of failed construction. Note that this is not yet supported on all that many compilers.

The result of a `new` expression using `countable` is an object allocated on the heap that has a header block that holds the count, i.e. we have extended the object by prefixing it. We can provide a couple of features in an anonymous namespace (not shown) in the implementation file for for supporting the count and its access from a raw pointer:

```
struct count
{
    size_t value;
};

count *header(const void *ptr)
{
  return const_cast<count *>
   (static_cast<const count *>(ptr) - 1);
}
```

An important constraint to observe here is the alignment of `count` should be such that it is suitably aligned for any type. For the definition shown this will be the case on almost all platforms. However, you may need to add a padding member for those that don't, e.g. using an anonymous `union` to coalign `count` and the most aligned type. Unfortunately, there is no portable way of specifying this such that the minimum alignment is also observed – this is a common problem when specifying your own allocators that do not directly use the results of either `new` or `malloc`.

Again, note that the count is not considered to be a part of the logical state of the object, and hence the conversion from `const` to non-`const` – `count` is in effect a `mutable` type.

The allocator functions themselves are fairly straightforward:

```
void *operator new
     (size_t size, const countable_new &)
{
  count *allocated = static_cast<count *>
  (::operator new(sizeof(count) + size));
  // initialise the header
  *allocated = count();
  // adjust result to point to the body
  return allocated + 1;
}

void operator delete
       (void *ptr, const countable_new &)
{
  ::operator delete(header(ptr));
}
```

Given a correctly allocated header, we now need the *Countable* functions to operate on `const void *` to complete the picture:

```
void acquire(const void *ptr)
{
    if(ptr)
    {
        ++header(ptr)->value;
    }
}

void release(const void *ptr)
{
    if(ptr)
    {
        --header(ptr)->value;
    }
}

size_t acquired(const void *ptr)
{
    return ptr ? header(ptr)->value : 0;
}

template<typename countable_type>
void dispose(const countable_type *ptr,
const void *)
{
  ptr->~countable_type();
  operator delete(
    const_cast<countable_type *>(ptr),
    countable);
}
```

The most complex of these is the `dispose` function that must ensure that the correct type is destructed and also that the memory is collected from the correct offset. It uses the value and type of first argument to perform this correctly, and the second argument merely acts as a strategy selector, i.e. the use of `const void *` distinguishes it from the earlier `dispose` shown for `const countability *`.

Getting smarter

Now that we have a way of adding countability at creation for objects of any type, what extra is needed to make this work with the `countable_ptr` we defined earlier? Good news: nothing!

```
class example
{
  ...
};

void simple()
{
  countable_ptr<example>
    ptr(new(countable) example);
  countable_ptr<example> qtr(ptr);
  // set ptr to point to null
  ptr.clear();
}   // allocated object deleted when qtr
destructs
```

The `new(countable)` expression defines a different policy for allocation and deallocation and, in common with other allocators, any attempt to mix your allocation policies, e.g. call `delete` on an object allocated with `new(countable)`, results in undefined behaviour. This is similar to what happens when you mix `new[]` with `delete` or `malloc` with `delete`. The whole point of *Countable* conformance is that *Countable* objects are used with `countable_ptr`, and this ensures the correct use.

However, accidents will happen, and inevitably you may forget to allocate using `new(countable)` and instead use `new`. This error and others can be detected in most cases by extending the code shown here to add a check member to the `count`, validating the check on every access. A benefit of ensuring clear separation between header and implementation source files means that you can introduce a checking version of this allocator without having to recompile your code.

Conclusion

There are two key concepts that this article has introduced:

The use of a generic requirements based approach to simplify and adapt the use of the COUNTED BODY pattern.

The ability, through control of allocation, to dynamically and non-intrusively add capabilities to fixed types using the RUNTIME MIXIN pattern.

The application of the two together gives rise to a new variant of the essential COUNTED BODY pattern, UNINTRUSIVE COUNTED BODY. You can take this theme even further and contrive a simple garbage collection system for C++.

The code for `countable_ptr`, `countability`, and the `countable new` are included on the ACCU web site, and next month's disk.

Kevlin Henney
kevlin@acm.org

counted_ptr<type> By Jon Jagger

Introduction

In my previous article I introduced the general idea of a pointer class. This time I'm going to focus on a specific pointer. A counted pointer. Or to be more specific, what is often called a detached counted pointer [1,2]. Counted pointers are the things that do reference counting. The idea behind reference counting is very simple. I'll use string as my example. Suppose two string objects exist and happen to contain the same value.

```
void peri()
{
  // string::string(const char
*literal)
  string theory("hello");

  // string::string(const string &rhs)
  string vest(theory);
}
```

The question is can theory and vest share the state that represents what after all is a common value. Clearly they can, the issue is what are the consequences of this sharing.

Naïve and broken

```
class string
{
public:
  string( const char *literal );
  string( const string &rhs );
  string &operator=( const string &rhs
);
  ~string();
  char &operator[]( size_t index );
  ...
```

```
private: // state
  char *rep;
};

string::string( const char *literal )
  : rep(new char[strlen(literal) + 1])
{
  strcpy(rep, literal);
}

string::string( const string &rhs )
  : rep(rhs.rep)
{
  // empty
}

string &string::operator=( const string
&rhs )
{
  rep = rhs.rep;
  return *this;
}

string::~string()
{
  delete rep;
}

char &string::operator[]( size_t index
)
{
  return rep[index];
}
```

The first consequence of naive sharing is that it breaks the Law of Least Astonishment. If I change theory I do not expect vest to change and when it does I'm more than a little annoyed.

```
// theory == "Jello"
theory[0] = 'J';

// !!! prints Jello
cout << vest << endl;
```

There are various ways to solve this. For example a copy on write pointer, which I'll look at in another pointer article.

The second consequence of sharing is that it introduces an associative relationship. Or to be blunt a pointer. theory and vest now hold a pointer to, and thus share, the state that represents the value "hello". Whenever we have an associative relationship we are implicitly talking about separate objects with separate lifetimes. We must somehow manage the shared relationship. Custody. The naïve code above is fatally flawed because it utterly fails to manage the shared relationship. Theory and vest share the same scope; the string destructor will be called twice at the end of the peri function and the code will do a double deletion. The alternative of making the destructor empty is no good either. This will just cause a memory leak. A solution to this problem is to introduce extra information which is used to manage the shared relationship. A reference count.

The Basic Idea

The extra information required is simply the number of objects that are participating in the sharing. If you think about it you'll quickly realise that this number must also be shared. Here's a first cut at a working version of string.

```
namespace accu
{
  class string
  {
  public:
    ...
  private:
    char *rep;
    int *count;
  };
}
```

The plain constructor allocates some memory to hold a replica of the literal, and some more memory to hold the shared reference count. It initialises the count to one to indicate that it (the object that the constructor is constructing) does not share the state with any other string objects. Not yet.

```
namespace accu
{
  string::string( const char *literal )
    : rep(new char[strlen(literal) +
1])
    , count(new int(1))
```

```
  { strpcy(rep, literal);
  }
}
```

The copy constructor simply makes two pointer initialisations. In other words shallow copies. Normally a shallow copy is dangerous. However in this case the idea is to do just that. It's not dangerous because we have a count that is managing the sharing. The vital line is the count increment.

```
namespace accu
{
  string::string( const string &rhs )
    : rep(rhs.rep), count(rhs.count)
  {
    ++*count;
  }
}
```

With this copy constructor then after...

```
string theory(“hello”);
string vest(theory);
```

...theory.rep and vest.rep both point to the same shared state and, vitally, theory.count and vest.count both point to the same shared integer which holds the value two indicating that two objects (theory and vest) are sharing the state.

The destructor can now use the shared count to determine whether it is the last object referring to the state. If it is it must do the deletions, if not it must decrement the count since the string object that is dying holds one of the references and it is, well, dying.

```
string::~string()
{
  if (--*count == 0)
  {
    delete rep;
    delete count;
  }
}
```

The copy assignment operator is basically just a combination of the code in the destructor plus the code in the copy constructor. The self assignment trap is handled by reordering the statements rather than via an explicit (this != &rhs) test.

```
string &string::operator=( const string
&rhs )
{
  ++*rhs.count;
  if (--*count == 0)
  {
    delete rep;
    delete count;
  }
  rep = rhs.rep;
  count = rhs.count;
  return *this;
}
```

counted_ptr<type>

Clearly there is a lot of code in this that is generic and not specific to string. Let's abstract string away as a template argument to create counted_ptr<>. This will allow us to rewrite string like this.

```
namespace accu
{
  class string
  {
  public:
    string( const char *literal );
    ...
  private:
    class body;
    counted_ptr<body> ptr;
  };
}


// accu/counted_ptr.hpp

#ifndef ACCU_COUNTED_PTR_INCLUDED
#define ACCU_COUNTED_PTR_INCLUDED

namespace accu
{
  template<typename type>
  class counted_ptr
  {
  public: // create/copy/destroy
    counted_ptr( type *p );
    counted_ptr(const counted_ptr &rhs
);
    counted_ptr &operator=
            ( const counted_ptr &rhs
);
    ~counted_ptr();
  public: // access
    type *operator->() const;
    type &operator*() const;
    type *raw() const throw();
  private: // precondition for -> an *
    void check_not_null_ptr() const;
  private: // plumbing
    void increment() const;
    void decrement() const;
  private: // state
    type *ptr;
    int *count;
  };
}

//include-all compilation model
#include "accu/counted_ptr-
template.hpp"
```

```
// accu/counted_ptr-template.hpp

#if !defined ACCU_COUNTED_PTR_INCLUDED
\
  || defined
ACCU_COUNTED_PTR_TEMPLATE_INCLUDED
#error "include
<accu/counted_ptr.hpp>:"\
  "counted_ptr-template.hpp must not “\
  “be included directly"
#endif

#define
ACCU_COUNTED_PTR_TEMPLATE_INCLUDED
...
namespace accu // counted_ptr - cre-
ate/copy/destroy
{
  template<typename type>
  counted_ptr<type>::counted_ptr( type
*p )
    : ptr(p), count(new int(1))
  {
    // empty
  }

  template<typename type>
  counted_ptr<type>::counted_ptr(
                 const counted_ptr &rhs
)
    : ptr(rhs.ptr), count(rhs.count)
  {
    increment();
  }

  template<typename type>
  counted_ptr<type> &
  counted_ptr<type>::operator=(
                 const counted_ptr &rhs
)
  {
    rhs.increment();
    decrement();  // this IS
    ptr = rhs.ptr;// self-assignment
safe
    count = rhs.count;
    return *this;
  }

  template<typename type>
  counted_ptr<type>::~counted_ptr()
  {
    decrement();
  }
}

namespace accu // counted_ptr - access
{
  template<typename type>
  type *counted_ptr<type>::operatr->()

const
  {
    check_not_null_ptr();
    return ptr;
  }

  template<typename type>
  type &counted_ptr<type>::operator*()
const
  {
    check_not_null_ptr();
    return *ptr;
  }

  template<typename type>
  type *counted_ptr<type>::raw()
                            const
throw()
  {
    return ptr;
  }
}

// counted_ptr – preconditions
namespace accu

{
  template<typename type>
  void counted_ptr<type>::
  check_not_null_ptr()const
  {
    if (ptr == 0)
    {
      throw logic_error(
            "counted_ptr: null
pointer");
    }
  }
}

namespace accu // counted_ptr - plumb-
ing
{
  template<typename type>
  void counted_ptr<type>::
  increment() const
  {
    ++*count;
  }

  template<typename type>
  void counted_ptr<type>::
  decrement() const
  {
    if (--*count == 0)
    {
      delete ptr;
      delete count;
    }
  }
}
```

Belt and Braces

Let's take a look at some of the subtler issues of counted_ptr that don't seem to be discussed much in the C++ literature. The first one is creating a full blown reference_count class to replace the raw integer pointer. This is what Barton and Nackman do in their book [3]. It is a pity they do not say why they do it. It concerns the code inside decrement()

```
if (--*count == 0)
{
  delete ptr;
  delete count;
}
```

The question is what happens if the destructor called via the delete ptr expression throws an exception. The answer is that the memory pointed to by count won't be reclaimed. This is because count is a raw pointer and not a fully constructed class object. One way to solve this is simply to delete the integer first...

```
if (--*count == 0)
{
  delete count;
  delete ptr;
}
```

However, it might be useful to create a general reference_count class. There is also an issue concerning the nature of ++ and --. If you are working with multi-threading you need to be sure that ++ and -- are atomic. So let's create a reference_count class.

```
// accu/reference_count.hpp

namespace accu
{
  class reference_count
  {
  public: // create/copy/destroy
    reference_count();
    reference_count(
            const reference_count &rhs
);
    reference_count &operator=(
            const reference_count &rhs
);
    ~reference_count();
  public: // query
    bool is_unique() const;
  private: // plumbing
    void increment() const;
    void decrement() const;
  private: // state
    int *count;
  };
}

// accu/reference_count.cpp

namespace accu
{
  reference_count::reference_count()
    : count(new int(1))
  {
    // empty
  }

  reference_count::reference_count
```

```
           ( const reference_count &rhs
)
    : count(rhs.count)
  {
    increment();
  }

  reference_count &
reference_count::operator=
           ( const reference_count &rhs
)
{
  rhs.increment();
  decrement();    // this IS self-
  count = rhs.count;// assignment safe
  return *this;
}

  reference_count::~reference_count()
  {
    decrement();
  }
}

namespace accu
{
  bool reference_count::is_unique()
const
  {
    return *count == 1;
  }
}

namespace accu
{
  void reference_count::increment()
const
  {
    ++*count;
  }

  void reference_count::decrement()
const
  {
    if (--*count == 0)
    {
      delete count;
    }
  }
}
```

With this class counted_ptr simplifies somewhat.

```
// accu/counted_ptr.hpp
. . .
namespace accu
{
  class reference_count;

  template<typename type>
  class counted_ptr
  {
  public:
    counted_ptr( type *p );
    // default copy constructor
    counted_ptr &operator=(
                const counted_ptr &rhs
);
    ~counted_ptr();
  public:
    // ...
  private:
    type *ptr;
    reference_count count;
  };
}

// accu/counted_ptr-template.hpp
. . .
namespace accu // counted_ptr - cre-
ate/copy/destroy
{
  template<typename type>
  counted_ptr<type>::counted_ptr(type
*p)
    : ptr(p), count()
  {
    // empty
  }

  template<typename type>
  counted_ptr<type> &
  counted_ptr<type>::
  operator=( const counted_ptr &rhs )
  {
    if (this != &rhs)
    {
      if (count.is_unique())
      {
        delete ptr;
      }
      ptr = rhs.ptr;
      count = rhs.count;
    }
    return *this;
  }

  template<typename type>
  counted_ptr<type>::~counted_ptr()
  {
    if (count.is_unique())
    {
      delete ptr;
    }
  }
}
```

Continuing the theme of exception safety, let's look at the constructor for counted_ptr. A typical use will be something like...

```
namespace accu
{
  string::string( const char *literal )
    : ptr(new body(literal))
  {
    // empty
  }
}
```

Now ptr inside counted_ptr<> is a raw pointer. That means if the reference_count constructor throws an exception (which it could) we will have a resource leak because the argument to the counted_ptr constructor was a raw pointer created via the new body(literal) expression. I've thought about this and I can't see an elegant solution. The best I can come up with is to split off the initialisation of the reference_count into a separate method. Like this...

```
namespace accu
{
  template<typename type>
  counted_ptr<type>::counted_ptr( type
*p )
    : ptr(p), count()
  {
    auto_ptr<type> resource(p);
    count.initialise();
    resource.reset(0);
  }
}

namespace accu
{
  class reference_count
  {
  public:
    reference_count() throw();
    void initialise();
  private:
    int *count;
  };
}

namespace accu
{
  reference_count::
  reference_count() throw()
    : count(0)
  {
    // empty
  }

  void reference_count::initialise()
  {
    count = new int(1);
  }
}
```

If anyone can see a "better" solution I'd appreciate an email.

Final Polish

Now that we have a nicely separated reference_count into a new class we have the opportunity of making the integer type a template parameter.

```
namespace accu
{
  template<typename integer>
  class reference_count;

  template<typename type,
                  typename integer =
int>
  class counted_ptr
  {
  public:
    // ...
  private:
    type *ptr;
    reference_count<integer> count;
  };
};

namespace accu
{
  template<typename integer>
  class reference_count
  {
  public:
    // ...
  private:
    integer *count;
  };
}
```

Remembering the problems hinted at with ++ and -- and multithreading you could create a refence_count class based on an atomic integer for example.

```
class atomic_integer
{
public:
  // ...
  atomic_integer &operator++();
  const atomic_integer operator++(int);
  atomic_integer &operator--();
  const atomic_integer operator--(int);
  // ...
};
```

Or perhaps an integer with a limited range. For example one where an attempt to decrement zero to minus one would cause an exception.

That's almost it for now. There is one thought I'll leave you with though. Right at the very start I declared operator[](size_t index) inside the string definition. How does this sit with the reference_counting?

```
string theory("hello");
string vest(theory);
theory[0] = 'J';
cout << vest << endl;  // must print
"hello" not "jello"
```

Errata

To finish I'd like to correct a serious bug that crept into my previous article. Then, as now I used the include-all model for template compilation. In particular in the file accu/pointer-template.hpp I wrote this

```
namespace // unnamed
```

```
{
  template<typename type>
  void check_not_null( type *ptr )
  {
    …
  }
}
```

My motivation for not making this a method of pointer<type> was reasonable enough: to keep the interface of pointer<type> “clean”. However, thanks to the include-all template compilation model, each translation unit that includes this file will get its own copy of the check_not_null function and each copy will live in its own unnamable namespace. This will violate the One Definition Rule. My thanks to Kevlin for spotting this.

The code from this article is available on the ACCU Overload web pages.

Jon Jagger
jjagger@qatraining.com

References (also counted 1,2,3 :-)

1. Ruminations on C++, Andrew Koenig and Barbara Moo, Addison Wesley, ISBN 0-201-4233-1, Chapter 7. Handles: Part 2. Page 67.

2. Scientific and Engineering C++, John Barton and Lee Nackman, Addison Wesley, ISBN 0-201-53393-6, Chapter 14 Pointer Classes, page 419

3. The C++ Programming Language, 3rd edition, Bjarne Stroustrup, Addison Wesley, ISBN 0-201-88954-4, 25.7 Handle Classes, page 782.

UML – Parameterised Classes (Templates) and Utilities
By Richard Blundell

Introduction

So far we have covered a number of commonly-used areas of the Unified Modelling Language. We have discussed using the UML for designing and documenting classes and objects, and patterns as collaborations of classes, in *static structure diagrams*. We have also looked at ways of representing the dynamic behaviour of a single class or object in a *state-transition diagram*. Rather than continue with other techniques for depicting dynamic behaviour, I shall turn our attention this time to *templates* (which are called *parameterised classes* in UML parlance) and utilities (used to deal with ‘global’ functions). Both of these can be used on static structure diagrams. I shall also mention the use of dependencies, which are used on a number of different types of diagram.

Parameterised Classes

Templates are a facility to represent classes or functions at an additional level of abstraction. They are commonly used in (although not limited to) collection classes, because this is a common domain where you can easily abstract the behaviour of the container and the operations you wish to perform on it without knowledge of the underlying data type. You can consider a linked list class that holds integers and a linked list class that holds strings. A list template allows you to code the basic functionality of a list without worrying about the type of variable that it will hold. Later, this template can be instantiated to generate automatically a list class that holds integers, strings or whatever. In the UML, template classes are known as *parameterised classes*, because they define templates for classes that require one or more parameters before they become real classes that can be used in a system.

Let us look at the list template from the C++ standard library (I have omitted the data members and all but the most-used methods):

```
template <typename T, class A =
allocator<T> >
class list
{
  public:
    explicit list(const A& al = A());
    iterator begin();
    iterator end();
    size_type size() const;
    bool empty() const;
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    iterator insert(iterator it,
      const T& x = T());
    iterator erase(iterator it);
    void sort();
    void reverse();
    // etc...
};
```

This template class is represented using UML notation in figure 1. Parameterised classes are shown using a normal rectangular class symbol, but with a dotted box at the top right that holds the parameters for the class, in our case, the value type T, and the (optional) allocator class A.

The use of default template parameters is not shown in the UML 1.1 Notation Guide. I am not sure if this is an intentional omission or an oversight, because as far as I can see the semantics for template parameters are the same as for the parameters of operations. If so, then the syntax I have shown above should be OK.

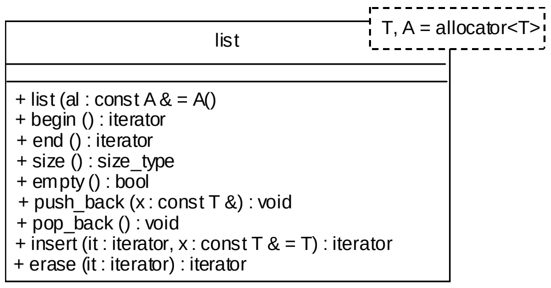
list
T, A = allocator<T>
+ list (al : const A & = A()
+ begin () : iterator
+ end () : iterator
+ size () : size_type
+ empty () : bool
+ push_back (x : const T &) : void
+ pop_back () : void
+ insert (it : iterator, x : const T & = T) : iterator
+ erase (it : iterator) : iterator

Figure 1 – The list template from the standard library, with attributes and some operations suppressed.

It is worth remembering that the free parameters in a parameterised class need not be types, but may be values as well, or a mixture of types and values. Although types are the most common kind of template parameter, integers and other values are occasionally used. I'm not aware of many cases of this in the standard library, but Microsoft's Active Template Library (ATL) uses these quite a lot, with integer and pointer template parameters occurring with some frequency.

To show non-type parameters in action, figure 2 depicts a class that Kevlin Henney described in an earlier Overload article on templates [1]. His template involved a struct with a single static data member that referenced a recursively-instantiated version of its own containing templatised struct! The result was a compile-time constant that was equal to x raised to the power of y. The template had two free integer parameters. I have shown the implementation in a note for those who do not have access to the original reference. I have also seen references to arrays with in-built size checking that use integers as template parameters (also, the ATL contains a fixed-size array template called CComUnkArray, for example), although I don't know how genuinely useful such constructs are in practice.

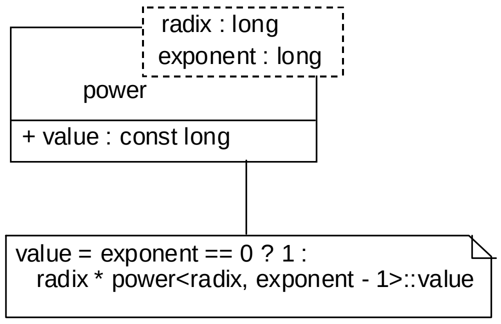
radix : long
exponent : long
power
+ value : const long
value = exponent == 0 ? 1 :
radix * power<radix, exponent - 1>::value

Figure 2 – Kevlin's circular template, using parameterisation on integer values!

Stereotypes – a digression

Before we get to utilities, I should mention the basic extension mechanisms of the UML, because we will require one of them next. If the UML were too complex, no one would ever learn it all, or enough of it, and so no one would use it. If it were too simplistic, then it would not be sufficiently powerful to be of much use in anything but the most straightforward systems. The result is that the UML has built-in support for quite a wide range of concepts, but also possesses a number of techniques to allow it to be consistently extended when the need arises. The three main methods, or *extensibility mechanisms* as they are often called, are *constraints*, *tagged values* and *stereotypes*. I shall only discuss the latter at the moment – the others can wait for another time.

Stereotypes are modelling elements that are, in a sense, specialised or derived forms of existing elements. Take an existing element, and add a stereotype symbol to it, and it becomes a refined version with additional properties and nuances that distinguish it from the original. A number of stereotypes are predefined by the UML, but new ones can be minted as and when the need arises. (Care should, of course, be exercised because the essential communication enabled by the UML will break down if no one knows what your 57 new stereotypes mean!)

A stereotype symbol consists of a keyword enclosed in guillmets like this: «wibble». The keyword names the new stereotype that you have created, and it is assumed that the extended features of this new entity are documented somewhere accessible. I will only be using pre-defined stereotypes here, but the option of creating your own exists, as long as you can justify this on the grounds that the existing semantics of the UML could not easily cover your new case. [End of digression!]

Utilities

Sometimes you may find that you have a number of static functions that are related in operation or task, but do not seem to belong to any of your existing classes. You may additionally have some static attributes (usually constants) that may be related to these functions. There are a number of reasons why these they may not seem to fit in anywhere:

First, you may not have analysed your problem domain accurately enough (or you may be insufficiently familiar with it) so that you cannot work out where the functions belong. Perhaps they seem to relate to two or more classes and you can't decide which, if any, is the 'correct' one.

Second, you may have a function that, although closely related to a particular class, cannot be made a class member for language reasons. Examples of this include things like overloaded operator functions, which (in C++) often need to be made friends of the class so that they can work with implicit conversions to either of their arguments (e.g. operator+(...) for strings or complex numbers).

Third, you may wish to preserve the 'calling' syntax of a function from another domain. For example, the *sin* method in Java could have been made a member of the Number or Double classes, but then programmers would need to write code of the form: y = x.sin(), rather than the more familiar and natural style with the parameter on the right: y = sin(x) (see fig. ???).

A *utility* can be used to 'package' these static functions and data together into a single entity. A utility is a special type of class, and is denoted using the usual class rectangle modified with the stereotype «utility». This stereotype symbol is displayed at the top of the name compartment of the class, just above the class name, as shown in figure 3. Because you cannot have an instance of this class, all attributes and operations within it are assumed to be static. A utility can be implemented in C++ using either a class or a namespace.

An example of a utility is the Math class in Java. This class has a number of mathematical functions and constants packaged within it. Packaging these functions up avoids the problems of polluting the global namespace, especially because many of the functions have short common names. To call one of these methods in Java, you need to prefix the function name with Math (so you call Math.sin(x), or Math.sqrt(y), for example). The Math class is illustrated in figure 3. Note that the attributes E and PI are declared *final* in the source.

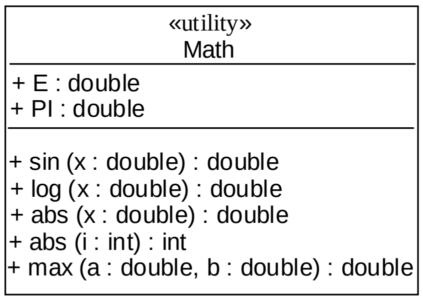
«utility»
Math
+ E : double
+ PI : double
+ sin (x : double) : double
+ log (x : double) : double
+ abs (x : double) : double
+ abs (i : int) : int
+ max (a : double, b : double) : double

Figure 3 - An excerpt from the Java Math utility, showing constant attributes and static methods.

Parameterised Utilities

There is no reason, of course, why you can't have a parameterised utility class. Such a class would be a collection of (related) static functions with one or more free types that have yet to be specified. We could therefore define a utility class that contains the common *min* and *max* template functions as shown in figure 4. Because you cannot instantiate it, the functions in a parameterised utility are template functions. The C++ compiler will automatically bind the free parameters (and hence instantiate the template function) when you call one of the utility functions.

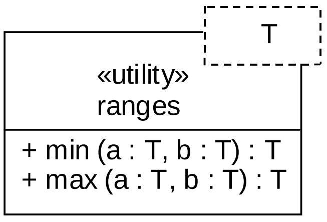

Figure 4 - A parameterised utility class.

Dependencies

An apparently unrelated concept is that of *dependencies*, but we shall see how these are used with parameterised classes below. Dependencies show what you might expect – relationships in which one entity depends upon another entity in some way. A dependency is shown as a dashed arrow from the dependent element to the element it depends upon (i.e. from the client to the server, if you like). The nature of the dependency is not necessarily specified, although a number of predefined types have been defined (including «bind» (see below), «refines», «instantiates», «uses», «calls», «friend», «becomes» and «supports»). Dependencies can denote relationships such as a source-code dependency, if you are interested in minimising your build time, or an object dependency if one object uses the services of another. Because dependencies are quite a general concept, they can be used in many different ways, and on many different types of UML diagram. Don't forget that the UML has been designed to be quite a generalised modelling language, and is not necessarily limited to software development.[1] You can sit down and work out dependency graphs for many different problem domains, and doing so can often help you to determine what are the really fundamental variables in your system upon which all else ultimately depends. Figure 5 shows a dependency diagram from a distinctly non-software field!

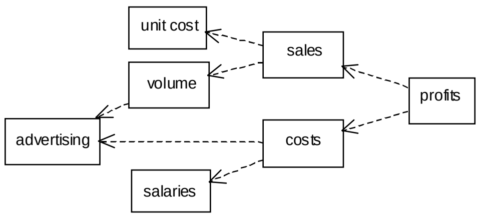

Figure 5 – A simple network of financial and non-financial entities with some hypothetical dependencies.

Instantiating Parameterised Classes

So, how do we actually use a parameterised class once we have designed it? Well we need to instantiate it by *binding* the free parameters of the class to actual types and values. There are two syntaxes for this. Either you can use the C++ angle-bracketed style syntax of template instantiation and draw a class with the name template<arg, arg, ...>. Alternatively you can draw a

[1] To quote from the UML specification, the UML "is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, *as well as for business modelling and other non-software systems*." (sic, but italics added)

dependency arrow, from the instance to the parameterised class, labelled with the stereotype «bind», which binds the values given to the free parameters of the class. Both of these forms are shown in figure 6.

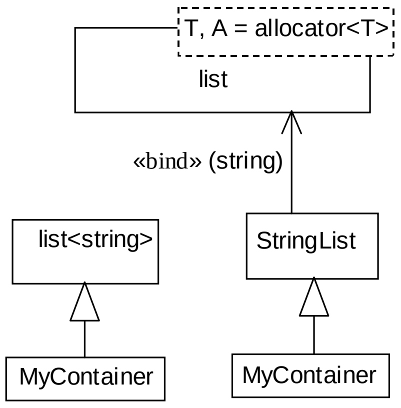

Figure 6 – Two ways to instantiate a parameterised class. The C++-style method on the left uses an unnamed temporary from which MyContainer is then derived. The second method on the right explicitly binds the type to the free parameter to create the intermediate StringList class

Summary

We now know how to document templates using the UML notation, and how to tidy up static functions into related groups called utilities in order to avoid namespace pollution. We have also seen how dependencies are denoted and some examples of how they are used. We still have a number of standard UML diagrams to cover, including diagrams to show detailed dynamic behaviour for collections of classes, as well as component and deployment diagrams that are concerned with the physical location and arrangement of, and relationships between, source and compiled software modules. I shall start to cover some of these next time.

Richard Blundell

References

1. Henney, Kevlin, */tmp/late/* Generating constants with templates*, Overload 11 pp 36-37

The Draft International C++ Standard

(Almost) No Casting Vote Standard's Report By Francis Glassborow

The recent meeting of WG21 & X3J16 in Sofia Antipolis (a few miles NW of Nice) was unusual to say the least. We always knew that this would be a relatively quiet meeting at which we looked forward and planned what to do next. In the event we were even further constrained by the prevarication of ISO who had yet to distribute the FDIS for vote by National Bodies. This meant that we were effectively unable to commit to anything. None-the-less the meeting was useful.

For those that have never been there, the Provence area of Southern France is typified by a lack of urgency and somewhat irrational plumbing and electrical systems. It took me several minutes to unravel the peculiarities of the switches in my hotel room (I even checked the bulbs to make sure they were not broken. I will leave descriptions of the plumbing till I share a pint with you in the local bar. It was not bad (in fact finding a clean, flushing public toilet half way up a mountain village was a welcome surprise) just inconsistent and sometimes more suited to Disneyland.

When eight of us decided to eat in the hotel on Saturday night we found ourselves having to search for staff in the restaurant. We probably felt that the couple of hours for that meal was leisurely but later experience suggested that we had bolted our meal by local standards. Sunday lunch in a pleasant square in Vence took something like three hours. Just as well the company was good. Latter that day I reflected that one of the particularly enjoyable things about ISO Standards meetings is the wide range of cultures involved. I should add that the a surprisingly large number of those attending C++ meetings are far from being single minded programming nerds. These are great people to be with and the sad thing about completing the C++ Standard is that quite a few will drop out.

A combination of priorities, injuries, job changes and accidents had severely reduced the attendance this time. Sean Corfield had had an accident that had prevented him from travelling and Steve Rumsby (one of the UK's outstanding experts) has changed his job with the result that he can no longer attend international meetings. That left me to be elevated to Head of Delegation and Principle UK Expert - just as well that there was little cause for technical expertise. Somehow I found myself shanghaied by WG21 into forming the drafting committee along with Herb Sutter. When I objected on the grounds that I had no experience I was told not to worry because there was unlikely to be any motions to draft. Actually there was one but it proved to be fairly easy and Herb did it by himself.

By now you may wonder what we actually did. Well there were a few things.

Possibly the most significant in the long run was the agreement to hold at least one meeting each year at the same place and approximately the same time as WG14+X3J11 (responsible for C). The idea being that interested people could attend relevant parts of both sets of meetings. More importantly it would provide an opportunity for informal face to face meetings between those who often have very different views as to the desirability and future of C and C++. Many members from both committees have been so involved with a specific language that they have lacked the time to track developments in the other. The main problem that we can foresee with our plans is that both groups include their share of bigots that view the other language with disdain as being inspired by the devil. Hopefully the sane majority and the more tolerant specialists will act as oil and we will not finish up with blood on the floor.

The first of these ‘collocated meetings’ will be somewhere in Silicon Valley, hosted by SGI from 5th-9th October this year. Strictly speaking the C++ meeting will only be the last three days (Tom Plum, our liaison with C had found it difficult to get the C committees to accept a five-day C++ meeting alongside theirs. In the event we agreed that the first two days would be used for technical presentations which are not part of our formal agendas and so individuals could decide whether they wanted to attend the whole five days or only for three of them.

The next ‘collocated’ meeting will be in Hawaii in October '99. This one is planned so that there will be an overlap of meeting days rather than both meetings being scheduled for exactly the same five days. Before that meeting we will have a C++ meeting in Dublin in late March or early April of ‘99. I gather that the C committees have tentative plans to return to London in the Summer of ‘99. This means that European, and particularly UK ‘experts’ have an excellent opportunity to get fully involved over the next couple of years (the next C meeting is in Copenhagen, 22nd-26th June).

The only substantial motion from WG21 relates to co-operation with C. C9X is being developed under a mandate to avoid gratuitous incompatibilities with C++. Actually they seem to have done quite a good job in keeping to this. What concerns C++ is that C9X plans to introduce a number of elements aimed at the world of numerically intense programming (hitherto the domain of (High Performance) FORTRAN). What we do not want is for ISO to require the next revision of C++ to include all these additions to C. Remember that C++ was required to be as close to being a superset of C89 as possible without seriously undermining its objectives. We do not believe that it would be reasonable to constrain the next revision of C++ to be similarly related to C9X. I wonder if readers have any opinions about this. If so please write in and share them with the rest of us.

The other issues for the week concerned the X3J16 part of the joint committees (though other NB's might be interested.)

The first issue was advice on changes to voting rights as regards NCITS (was ANSI) committees. Currently representatives must have attended two out of the last three meetings. As the amount of work gets progressively transferred to electronic consultation meeting frequencies are dropping. C++ currently plans on two per year but other less active committees anticipate dropping to only one per year. This could result in interested parties being disenfranchised by long qualification times (only those who have qualified by physical attendance can cast postal votes under the current ANSI/NITS rules). X3J16 decided to recommend that members should be eligible for postal votes from the end of their first physical attendance at a meeting. However they would only retain those rights so long as the actually attended at least one physical meeting each year. In other words you only get to vote if you have a demonstrable ongoing commitment.

I wonder if the UK might consider how it will respond to the growth of electronic consultation.

The next issue was probably the most important of the week. A Japanese delegation presented a request for WG21 to consider applying for a work item to standardise Embedded C++. The reason that we passed this to X3J16 for consideration is that an application for a new work item requires support from at least five NBs. We needed to explore the ramifications but clearly WG21 representatives could not vote on such an issue without consulting with their NBs.

The major issue is one of resource usage. As currently written C++ requires the Standard C++ Library to support such things as locales, alternative character sets etc. This makes some parts of the library very resource hungry. While this could be managed by compile time switches, the result would be the growth of proprietary dialects subtly incompatible with each other. We can also envision superb development tools that could strip out the fat. However we need to be realistic and explore ways in which we can provide a little more wriggle room (to use Bill Plauger's phrase).

This problem with resources is not confined to embedded systems and so at my instigation we drafted a motion that supported WG21 seeking a work item to produce a technical report on resource management in C++. Such a report would keep everyone focused on keeping together. I hope that sufficient other countries feel this is a constructive approach to the problem. I will certainly be urging the UK to support it. The Japanese seemed happy.

Much of the rest of our time was spent discussing mechanisms for handling defect reports on C++. More about these when we actually have a C++ Standard voted out by the NBs. I will be arranging for a report on this aspect for our Conference (11th & 12th September). For now, you should know that ACCU will be closely involved in this process.

Finally

A couple of more light-hearted items from the meeting. Bjarne Stroustrup and I were sitting together during most of the meetings. At one time his enthusiasm for my proposal to overload the semicolon operator (see this months issue of EXE Magazine, which you should receive) was such that Steve Clamage was heard to mildly rebuke the two of us for lack of attention to the business of the meeting. Quite like being back at school.

On another occasion I quoted a proposed price for something in pounds and added an aside that it would convert according to the normal commercial rules to the same number of dollars. That almost brought the house down. It was that kind of meeting.

When we were discussing the future of C++, I think I was ruled out of order when I suggested that we should first tackle the question ‘Has it got one?’

Just as well we did not have a meeting in France earlier on because I do not think we could have managed all the normal after hours work alongside four-hour dinners (the restaurants of Provence seem happy to fill each table just once per evening). Fortunately after hours was confined to enjoying the excellent company.

I can thoroughly recommend visiting Provence in the company of friends from other cultures so that you avoid the British habit of wanting to turn everywhere else into little Britains. Though I ate fish while there, chips never crossed my path.

Francis Glassborow
francis@robinton.co.uk

Generalizing Overloading for C++2000
By Bjarne Stroustrup

Abstract

This paper outlines the proposal for generalizing the overloading rules for Standard C++ that is expected to become part of the next revision of the standard. The focus is on general ideas rather than technical details (which can be found in AT&T Labs Technical Report no. 42, April 1,1998).

Introduction

With the acceptance of the ISO C++ standard, the time has come to consider new directions for the C++ language and to revise the facilities already provided to make them more complete and consistent.

A good example of a current facility that can be generalized into something much more powerful and useful is overloading. The aim of overloading is to accurately reflect the notations used in application areas. For example, overloading of + and * allows us to use the conventional notation for arithmetic operations for a variety of data types such as integers, floating point numbers (for built-in types), complex numbers, and infinite precision numbers (user-defined types). This existing C++ facility can be generalized to handle user-defined operators and overloaded whitespace.

The facilities for defining new operators, such as :::, <>, pow , and abs are described in a companion paper [B. Stroustrup: "User-defined operators for fun and profit," Overload. April, 1998].

Basically, this mechanism builds on experience from Algol68 and ML to allow the programmer to assign useful - and often conventional - meaning to expressions such as

```
  double d = z pow 2 + abs y;
```

and

```
  if (z <> ns:::2) // …
```

This facility is conceptually simple, type safe, conventional, and very simple to implement.

Basic Whitespace Overloading

Here, I describe the more innovative and powerful mechanism for overloading whitespace. Consider x*y. In programming languages (e.g. Fortran, Pascal, and C++), this is the conventional notation for multiplying two values. However, mathematicians and physicists traditionally do not use the operator *. Instead they use simple juxtaposition to indicate multiplication. That is, for variables x and y of suitable types,

```
  x y
```

means multiply x by y.

This is simply achieved by overloading the space operator for double-precision floating-point values:

```
double operator (double d1, double d2)
{
  return d1*d2;
}
```

Or - more explicitly - equivalently

```
double operator ' '(double d1, double d2)
{
  return d1*d2;
}
```

Given one of these definitions, a physicist can use his (or her) conventional notation rather than the notation that has become conventional among computer scientists:

```
double f(double x, double y, double z)
{
// using also a user-defined operator pow
  return a + x y pow z;
}
```

Clearly, the space operator has (by default) a precedence lower than pow and higher than +. The mechanism for assigning precedence to user-defined operators is described in detail in the companion article. The superscript operator allows a further improvement:

```
double operator super(double d1, double
d2)
{
  return d1 pow d2;
}

double f(double x, double y, double z)
{
  // using user defined
  // superscript operator
  return a + x y^z;
}
```

Naturally, this requires that overloading is allowed for built-in types. However, to avoid absurdities, it is (still) not allowed to provide new meanings for the built-in operators for built-in types. Thus, the language remains extensible but not mutable. In fact, generalizing the overloading rules allows us to provide a unified clean framework for built-in and user-defined types as well as for built-in and user-defined operators. This improvement furthermore opens the opportunity to eliminate many of the anarchic and error-prone traditional implicit conversions inherited from C in the next revision of the C++ standard.

Previous Work

The overloading mechanisms described here are partly inspired by the pioneering work of Bjørn Stavtrup [B. Stavtrup: "Overloading of C++ Whitespace." JOOP. April, 1992]. However, Dr. Stavtrup failed to take object types into account so that his system was far less flexible than the mechanisms described here. He also made the - not uncommon - mistake of tying his innovative linguistic mechanism up with a peculiar design methodology and a proprietary toolset.

FFPL [Francois French and Paul Lawson : "A language for Free Form Programming." POPL. 1992] and White [G. LeBlanc: "Whitespace overloading as a fundamental language design principle." JIR. Vol. 24, no. 3, May 1994] were academic projects that never had any users - except possibly their designers. It is not clear that White was ever implemented and Dr. Wimmelskaft of the university of Horsens , Denmark, have conjectured that it was, in fact, unimplementable [Wimmelskaft: "A refutation of White." JIR. Vol. 26, no. 4, March 1996].

The overload mechanism described here generalizes the built-in use of concatenation for string literals in C and C++. In particular, space is predefined to mean C-style string concatenation. For example,

```
  "this is" "a single " "C-style string"
```

is by the lexical analyzer turned into

```
  "this is a single C-style string"
```

Thus whitespace between two C-style string literals is interpreted as concatenation. The facility was missing in K&R C, introduced into ANSI C, and adopted by C++ in the ARM (Ellis and Stroustrup: "The Annotated C++ Reference Manual, " Addison-Wesley 1989).

Overloading Separate Forms of Whitespace

There are of course several forms of whitespace, such as space, tab, // comments, and /* */ comments. A comment is considered a single whitespace character. For example,

```
/* this comment is considered a single
   character for overloading purposes
*/
```

It was soon discovered that it was essential to be able to overload the different forms of whitespace differently. For example, several heavy users of whitespace overloading found overloading of newline ('\n'), tab ('\t'), and comments as the same arithmetic operator is counterintuitive and error prone. Consider:

```
double z1 = x y;    // obvious
double z2 = x
              y;    // obscure
double z3 = x /* asking for trouble */ y;
```

In addition, different overloading of different whitespace characters can be used to mirror conventional two-dimensional layout of computations (see below).

Stavtrup claimed that it was important to distinguish between a different number of adjacent whitespace characters, but we did not find that mechanism useful. In fact, we determined it to be error-prone and omitted for Standard C++.

Overloading Missing Whitespace

After some experimentation, it was discovered that the overloading mechanism described so far did not go far enough. When using the mechanism, the physicists tended to omit the space character and write

```
  xy
```

rather than

```
  x y
```

This problem persisted even after the overloading rules had been clearly and repeatedly explained. What was needed wasn't just the ability to overload explicit use of whitespace, but also implicit application. This is easily achieved by modifying the lexical analyzer to recognize

```
  xy
```

as the two tokens

```
  x y
```

when x and y are declared. The "missing whitespace" between two identifiers are assumed to be a space.

Deciding how to resolve the ambiguity that arise for xy when x, y, and xy are all declared was one of the hardest issues to resolve for the whitespace overloading design.

One obvious alternative is to apply the "Max Munch" rule (also known as the greedy parsing rule) to this so that xy means the single identifier xy rather than x y. However, this has the unfortunate effect that the declaration of xy can completely change the meaning of a conforming program. That is, adding "int xy;" to

```
  int x, y;
  // …
  int z = xy; // means x y
```

yields

```
  int x,y,xy;
  // …
  int z = xy; // means xy
```

when space is overloaded to mean multiplication. It was therefore decided that the "Max Munch" resolution was unsuitable for large-scale programming.

Instead, it was decided to limit identifiers to a single character, by default:

```
  // error: two-character identifier
  int xy;
```

This may seems Draconian at first. However, since we now have the full Unicode character set available, we don't actually need hard-to-read long names. Such long names only make code obscure by causing unpleasantly long lines and unnatural line breaks. Multi-character names are a relic of languages that relied heavily on a global namespace and encouraged overly-large scopes.

Mathematicians and physicists in particularly appreciate the ability to use Greek letters:

```
  double β = φλ;
```

This facility was also an instant success in China and Japan where the Chinese character set provides a much richer set of single characters than does the various Latin alphabets.

Less traditional symbols are also useful. For example:

```
  // take my phone (•) off hook (•)
  •->•();
```

This example become even more natural when - as is common - the whitespace operator is overloaded to mean -> for the telephone class:

```
class Phone
{
  // …
  Phone* operator ' ' ()
  { return this->operator->(); }
  void •();  // off-hook
  // …
};

  Phone •;

  // take phone (•) off hook (•)
  ••();
```

It is also common to overload newline to mean application without arguments, that is (), so that what used to be the long-winded and ugly

```
  my_phone->off_hook();
```

becomes plain and simple

```
  ••;
```

Finally, the semicolon is most often redundant as a statement terminator so the grammar has been improved to make it optional in most contexts. Thus, we get:

```
  ••
```

Extensive use of such special characters together with imaginative and thoughtful use of whitespace overloading has had an immense impact on maintenance cost.

Should you feel the need for longer names - for example, if you don't have a high-resolution screen with a suitable large character set available - you can explicitly specify one using the multi-character identifier operator $:

```
  // explicitly multi-character name
  double $xy = 0.0;
  double x, y;

  // xy times x times y
  double Φ = xy x y;
```

Naturally this is best avoided. For compatibility, a $ as the first character of a translation unit means that every identifier can be implicitly multi-character. This has proven immensely useful during transition from the old to the new rules. As an alternative to $ as the first character, the header <> can be included:

```
  #include<>
```

Overloading \\ (double backslash) to mean "everything before this is a comment" has proven another useful transition tool. It allows old-style and new-style code to coexist:

```
  my_phone->off_hook(); // \\ ••
```

Given a new-style compiler, everything up until the • is ignored whereas an old-style compile ignores everything after the ;

Composite Operators

As described in the companion paper, C++2000 adopts a variant of the overloading of composite operators described in the ARM. This implies that we can define the meaning of

```
  x = a*b + c;
```

directly by a single

```
  operator = * + (Vector&, const Matrix&,
const Vector&, const Vector&);
```

rather than achieving this indirectly though function objects as described in Stroustrup: The C++ Programming Language (3rd edition). Addison-Wesley 1997.

Naturally, a composite operator can contain whitespace operators. For example,

```
  x = ab + c;
```

can be handled by

```
  operator = ' ' + (Vector&, const
Matrix&, const Vector&, const Vector&);
```

where multiplication is as usual represented by concatenation (missing whitespace). Some people go further by representing addition by newline to match the common convention of listing numbers in a column before adding them. Doing that we can define:

```
  operator = ' ' '\n' (Vector&, const
Matrix&, const Vector&, const Vector&);
```

to handle

```
  x = ab
        c;    // old-style: x = a*b+c
```

This convention is not universally appreciated and more experience is needed to estimate its impact on maintainability.

Availability

The generalized overloading mechanism described here has been in experimental use for some time and it is expected that most major C++ compiler vendors will ship it as an integral part of new releases in the near future. A preprocessor that implements the facility for any current C++ implementation can be freely downloaded from http://www.research.att.com/~bs/whitespace. html.

In addition to the overloading of missing whitespace, etc., this distributed version includes overloading based on the color of identifiers. Due to the limitations of the printing process used for this article, I cannot give examples, but basically a red x is obviously a different identifier to a green x. This is most useful for making scope differences obvious. For example, I use black for keywords, red for global variables (as a warning), blue for member names, and green for local variables. In all, a given character can be of one of 256 colors. Naturally, this again reduces the need for multiple-character identifiers while increasing readability. The lack of universal availability of color printers and problems of color blind programmers caused me to leave this feature out of the standard.

Current and Future Work

In preparation for standardization, formal specifications of the overloading mechanism in VDF and Z are being constructed. In addition, a simplified teaching environment is being constructed where operators such as *, +, and -> have been eliminated in favor of overloaded whitespace. Initial results indicates that this immensely shortens the time needed to learn C++ and should possibly be compulsory for non-expert programmers. A tool to automatically convert of old-style programs to new-style programs is being constructed; the inverse tool will not be needed.

Naturally, whitespace overloading is essentially language independent. Consequently, we are looking for ways of applying it uniformly across several programming languages to achieve common semantics. In addition, whitespace overloading clearly fits the C9x effort to support traditional numeric programming. Consequently, I confidently predict that the basic whitespace overloading mechanism will be part of the revised C standard.

Finally, work is underway to extend the character set, language syntax, and overloading rules to take advantage of 3D display devices. This will allow us to naturally represent multiplication, addition, and exponentiation as spatial displacements along three different axis. Because this project relies of the ability to fool the brain into accepting a projected image as 3D and because we don't take delivery of the 3D projection device until next spring, this project is usually referred to as "Project April Fool."

Bjarne Stroustrup
AT&T Labs, Florham Park, NJ, USA

Whiteboard

Irrational Behaviour By Graham Jones

Having read “Rational Values Part 3” I would like to respond to the Harpist’s comments about implementation and interface. He claims that I am confusing these. I don't think so, but we do seem to have very different ideas about what an interface is. My definition of an interface of a class is something like “everything the user needs to know about the class in order to be able to use it with confidence”.

Most programmers, suggest the Harpist, will expect to be able to create a Rational from a double, and particularly mentions mathematical constants. I wonder what most programmers would expect the following code to do.

```
Rational x, e;
x = Rational(1.2);
e = Rational(exp(1));
cout << 5*x << " " << e*e << endl;
```

I think that the calculation of 5*x has three main kinds of behaviour, depending on the size of your ints and the precision of your doubles: it may produce 6, some fraction close to 6 with a huge denominator, or it may throw an overflow exception. As for e*e, it's too complex for me to analyse. Anyway, this code is useless: no-one can use the Rational class with confidence if they make rationals out of doubles. There's no point in shielding the user from the implementation details of your class if you then expose the user to an even lower level of implementation in unpredictable ways. The point is not that the Rational class fails to be of industrial strength, but that the conversion to doubles fails to provide any kind of behaviour that could sensibly be documented. A while back, in CVu, the Harpist said “Writing re-usable code is harder than you think”. I’d like to re-phrase that as “Separating implementation from interface is harder than you think”.

The Harpist still seems to think that ints could be replaced with BigInts without changing the interface. Ignoring conversion from doubles, this would cause major changes to the interface: the exceptions that may be raised, the memory used and the time taken, would all be changed drastically. These can all affect the way the user of the class must write his or her program.

A few months ago I implemented the Playpen class that the Harpist described in CVu. According to the Harpist the class definition was all I needed to know about. But when I started to write code I found all sorts of things which would affect the user of the class. Error handling was one issue. For example: How should the class behave if the user tried to draw a point outside the Playpen? How should operating system errors be reported? Other things concerned the appearance of the Playpen: there were several ways in which a Playpen might appear different on different machines (which would negate the point of having the Playpen in the first place). Is the origin top left or bottom left - or somewhere else? Does updatepalette() affect the screen? Will my RED^BLACK be the same colour as yours? It is no exaggeration to say that I spent more time examining Simon Wood's implementation than writing mine, trying (and almost certainly failing) to understand how a Playpen should behave.

In the designs of both the Playpen and Rational classes, the Harpist seems to be assuming that “class interface equals class definition”. I think this is wrong, and hope I have explained why. If I thought this was an issue that just applied to a couple of tutorial classes, I wouldn't bother writing. However it seems to me that too many authors of libraries I have to use have a similar mindset. At first sight the libraries present a nice clean interface, and they certainly hide implementation details in the sense that it is very difficult for me to find out what they are. But when I use them, all kinds of undocumented and incomprehensible behaviour leaks out from underneath. There seem to be no controversy over the fact that sorting and searching functions in the STL expose the algorithms they use: everyone seems to think this is an improvement over qsort() and bsearch() where you could never be quite sure what you were getting. Perhaps we should learn from that when designing our own classes.

And finally: It might not sound like it from some of the above, but I am grateful to the Harpist for his many contributions to ACCU. I for one hope that he spends most of his time learning what to write about rather than polishing his writing.

Graham Jones

Implementations & Interfaces By The Harpist

One of the more serious problems with computing is that we use the same terms with such different meanings. When the differences are gross it matters very little, but when it comes to shades of grey we can all finish up confused. Above you will find a letter from Graham Jones in which he raises a number of excellent points. In this article I shall attempt to address these points.

One reason why I am happy to spend time writing for ACCU publications is that I learn much from the effort, and letters such as Graham's add considerable value. Let me go back into the deep past and look at some of the design criteria for C and contrast them with those for Java. I promise you that this is not a sidetrack.

TINSTAAFL

Portability and constancy over time is probably one of the most challenging aspects of language design. C (and in particular ISO C) provides one concept of a language interface. It introduced the concept of a strictly conforming program as one that would exhibit the same behaviour wherever it was run. The requirements for strictly conforming code are so demanding that it is highly debatable that anyone has ever written such a program, certainly any attempt to do so requires a highly specialised approach to code writing. C introduced a lesser classification; that of a conforming program. A conforming program is one that is accepted by a conforming implementation. In simple terms a conforming program does not exhibit undefined behaviour. A footnote clarifies that a conforming program may rely on non-portable behaviour of a conforming implementation. To learn more you need to read clause 4 of ISO/IEC 9899-1990.

The idea was to allow compiler implementors as much liberty as possible to get the best from the hardware and operating systems that they were writing for. For example a program that opened a file called 'LPT1' should do something, but it would only normally be on MSDOS based systems that the result was to send data to a printer via an MSDOS reserved file name. The major focus was to support very efficient code generation from C source code. The many traps that inexperienced programmers fall into are the price that is paid. Experienced programmers can extract much of the critical data from the required header files (such as limits.h). Inexperienced programmers often assume that all implementations will have the same range of values for int, the same relationship between signed and unsigned values etc. as that of the first compiler they used. C trades efficiency for consistency. It makes demands on the professionalism of programmers that academic languages such as Pascal, Modula 2 etc. forgo.

While C provides a programming interface that is relatively reliable it does not ensure uniform behaviour under all circumstances. It sets limits within which variations may occur. If you do not understand these limits (and sadly, the overwhelming majority of programmers do not) then your program will behave surprisingly even if it is a conforming program.

For example consider:

```
Enum
{
  controller = 0xEAF0,
  off = 0,
  on = 1
};

int main()
{
  char volatile * port = controller;
  do {
    if (*port >= 0) *port = on;
    else *port = off;
  }
}
```

Assume that this program provides thermostatic control. Its successful behaviour certainly relies on the implementation of char. If char is implemented as unsigned, *port will always be greater than or equal to zero and the heater will never be switched off. Of course this is a 'bad' program that relies on a gross characteristic of an implementation. However that is not the point. What is the result of the following program:

```
#include <stdio.h>
int main()
{
  int j = -27;
  printf("%d", j & 27);
  return 0;
}
```

It will (I believe) always produce a result. Almost always it will produce the same result but you would be seriously mistaken to believe that it does not rely on implementation defined behaviour. For those that do not spot it, C allows at least three different representations of negative integers – 2's complement, 1's complement and sign and value – and the result certainly depends on which your system uses.

Turning briefly to Java: here the language specification is deliberately much more tightly drawn up. All implementations are supposed to behave the same way. There is a heavy price to pay for this consistency. You cannot use anything that is not universally available. I shuddered when I first came across a class to handle a three-button mouse in Java. Think about it, such code cannot run correctly on an Apple Mac. You see, the designers of Java forgot an essential element of a universal language; you must include a specification for the underlying hardware. For Java to work as described you can only use specified hardware. Your graphics must be confined to the common subset of resolution, palettes etc. that all hardware running Java must support.

In a real sense a computer language specification provides an interface between a programmer and a computer. In general we are fairly content with a language that compiles everywhere and expect that that guarantees that the resulting program produces the same behaviour everywhere. That latter belief is almost impossible to fulfil.

Consider qsort() in C. The library specification specifies exactly what you can expect. In order to use it the programmer must know how to call the function and what parameters must be passed to it. It says nothing about any performance guarantees or resource requirements. Now Graham maintains that C++ does better in the STL because it exposes the algorithms it uses. That is not so, nor should it be. What is true of the STL is that certain extra constraints are placed on its functions. There is no requirement that any specific algorithm be used for sorting, only that whatever is used shall meet certain specified performance requirements. Actually a new sorting algorithm was developed recently that has a demonstrably better performance than that required by the STL sort specifications. Nothing forbids implementors from using this new algorithm even though its performance will be different from earlier choices. Can you imagine the howls of anguish if an implementation was not allowed to provide a 'better' solution because that might result in different 'behaviour' (performance or resource requirements) from that of competitors?

The most important lesson to learn about interfaces is to recognise the limits of what you have been guaranteed. The second important feature is to recognise trade-offs.

Look at the STL containers. At first sight programmers may wonder why they would ever wish to use a deque rather than a vector unless the capacity to add elements to the beginning was important. vector trades certain advantages (elements being in contiguous address space, fast random access etc.) for disadvantages (catastrophic performance hits when the expansion of a vector requires re-allocation of space, high cost for inserting elements other than at the end etc.).

The question that can be posed is 'is performance part of the interface or part of the implementation?' Graham would answer one way and I another. I think I understand Graham's viewpoint on this but I also think that it means that it is impossible to change the underlying implementation. Any such change to a class would automatically result in some form of performance change else why bother. When I write my application for a container so that it carefully confines itself to the common interface between different STL containers I do so precisely because I do not consider performance to be part of the implementation. I want to be able to select the implementation that best meets my requirements at that time. I want to be able to change my mind at any time.

The Problem of Not Being Exact

Integer types have the advantage of providing exact representations within specified limits (I will get to BigInts in a moment). When we implement them in C/C++ (and most other languages) we also pay a price for efficient arithmetic by allowing overflow and underflow to go undetected. The usual mechanism is to allow 'wrap round' of values. Most of the time programmers are happy to accept this price. The cost of detecting overflow/underflow in integer arithmetic is very high when we try to implement it at high level. There is also a high price if we elect to allow detection at the hardware level. It is in the nature of the representations we use for most systems that underflow/overflow during an integer computation often still produces the correct answer. For example most 16-bit systems will arrive at the correct answer for: (32000 + 4000 – 5000) even though an intermediate result has overflowed.

You may be wondering what I have against BigInts (where these are an integer type whose capacity is only limited by memory resources). The problem is that they are terribly inefficient. As the amount of storage required by a value ebbs and flows memory is being allocated and deallocated dynamically. We can reduce the amount of re-allocation by being more profligate with our resources (generously allocating extra space against future usage and being reluctant to release it until absolutely necessary). In some circumstances that is a price I will cheerfully pay, but at other times I would hate both frequent re-allocation and greedy use of resources.

In my mind the designer of a Rational type faces exactly the same kind of problems. S/he wants a type that has well defined general behaviour that can be refined by implementation decisions as needed. There will be a core of usage that will have invariant behaviour but beyond that…

It may surprise some programmers to learn just how much has to be sacrificed in the conflict between accuracy and speed. Look at the guarantees that C makes for floating point values. Remember that these go in sort of quantum leaps. There is a smallest increment between values. You may ask what happens when a calculation would result in an intermediate value. Surely you will say, such cases must result in one or other of the two nearest representable values. Perfectly reasonable and completely wrong. The theory of computation shows that such a requirement would place at least an order of magnitude performance hit on all floating point computations as compared to that we get by allowing the result to be within two 'quanta' (either the nearest or next nearest representable value). Almost universally languages accept the lesser accuracy in exchange for the greater speed.

Now let us turn to Graham's example of constructing a Rational from 1.2. He is correct in saying that my interface says nothing about the result. Given that 1.2 cannot be exactly represented in binary notation with a finite number of bits we have a problem. We know that we want a result of 6/5 but can we guarantee that we will get one. More to the point, what guarantees can we provide that any floating point number that could be represented exactly as a Rational within the range of numerators and denominators available will in fact be so represented? This is a difficult question and one that the designer of the interface will need to discuss with a skilled implementor. Clearly the interface designer wants to make the best reasonable guarantee but this will depend on many things. I think that a skilled implementor should be able to provide some pretty strong guarantees by using continued fractions (one quality of these is that alternate steps are too high and too low unless and until an exact result is arrived at). But note that this is deep in the domain of implementation. The designer specifies an interface for both the implementor and the user. S/he negotiates specification details that are acceptable to the user and achievable by the implementor. These may vary from time to time. The resulting program code still compiles though it may do different things for some corner cases. If you care you will need to act, but mostly you will not. Note that statement carefully. Mostly you will not care about any change in behaviour but sometimes you will. If you really care about that extra quantum of accuracy when using floats you will have to use a higher precision type. If you care about overflow you will have to check the range of the integer type you use. If you want a practically unlimited range of values you will have to sacrifice a great deal of speed to get it via a BigInt type.

What I am saying is that a professional programmer looking at a class interface can and should ask what guarantees are being given and at what price. You know that floats are not exact represenstions so you know that there may be a conversion problem. If it matters you can check the full specification or you can do the job yourself.

In the words of the master, ‘You should not pay for what you do not use.’ Building Rationals on top of BigInts straight off would be a gross breach of that maxim.

I hope that Graham will agree that out of our disagreement – alternative viewpoints – a better understanding of the issues can arise.

The Harpist

Debuggable new and delete Part Two

By Peter Pilgrim

In this second article I will present my solution to debuggable C++ dynamic memory allocation integrity.

Here is fresh recap [1]. The basic idea of heap space integrity is to use an identification method within the memory block itself. In other words how do we find out if a block of memory is valid heap space or not? [7] A function allocates a block of heap memory larger than the user requested, and divides this memory block, say B, into three parts: the *prefix*, the *middle*, and the *suffix*. Some magic identifier bytes are written into the prefix, and another set of magic bytes into the suffix. Finally we simply return to the user a pointer to the middle of the memory block .

The C++ language allows us to override the default implementation of the `::new` and `::delete` operators. The ‘::’ denotes the global scope of the identifiers. This important *hook* enables the implementation of debuggable global new & delete operators. These operators respectively call functions, which `mark` and `unmark` the memory block. For example writing OVERL%AD identifies the prefix of the memory block. The string is reversed and a character is changed for make up another identifier DA$LREVO, which is written as the suffix.

The C++ language enables a developer to take over the management of memory allocation in such a way that it appears omnipresent. Once we have defined a special `::new` and `::delete` operators, our custom built functions will be linked against other translation units. [3] For example having written a debuggable new and delete module called D. If we link D against translation units U, V, and W. Any calls to `::new` in either U, V, and W will refer to the operator defined in D.

Let us look at the data structures for the prefix and suffix parts of the memory block.

```
#define PREFIX_HEADER_SIZE
     sizeof(PrefixHeader)
```

```
#define SUFFIX_HEADER_SIZE
      sizeof(SuffixHeader)

enum MemoryStatus {
  // Diagnose memory problems
  MEMORY_OK,// healthy memory!
  MEMORY_UNKNOWN,// unrecognised memory
  MEMORY_UPPERBOUND,// corrupted
  MEMORY_LOWERBOUND,// corrupted
  MEMORY_ALREADY_FREED // already freed
};

union Alignment {
  int                a1;
  unsigned int       a2;
  long               a3;
  unsigned long      a4;
  float              a5;
  double             a6;
  // maybe long double a7; // DEC Alpha
};

const int ALIGNMENT_SIZE =
sizeof(Alignment);

struct PrefixHeader
{
  // Store the size of the block.
  size_t  size;
  // Special Identifier Magic.
  unsigned long  magic_word;
  // Prefix identifier bytes.
  char ch[ALIGNMENT_SIZE];
  // alignment bytes.
  Alignment alignment;
};

struct SuffixHeader
{
  // Suffix identifier bytes.
  char ch[ALIGNMENT_SIZE];
};
```

The enumeration `MemoryStatus` is used in the source code. It provides the status of the memory block. I have improved upon the original C version, which appeared in C Vu [2], by using the union to get the best end- of-structure alignment independent of the C++ compiler and the operating system. The end of the `PrefixHeader` structure features this union as a field member. Why do we need to align the structure? It is simply for portability reasons. We enforce the alignment of the PrefixHeader data structure so that it ends on a word boundary. If one thinks about the memory block in terms of an array of PrefixHeader elements, say `P[N]`. The address of the first PrefixHeader record will equal to `&P[0]`, but the address of the beginning of the second PrefixHeader element `&P[1]` is actually exactly equal to the address of the first byte of heap space that user gets to use (void *B). In other words the beginning of the middle section of the allocated memory block ***B***. For the sake of repeating information that appeared in `Kernighan & Ritchie's` classic C Book [6], C++ likewise does not mention anything about the exact alignment of memory returned from `operator ::new()`. Alignment of pointers returned from `operator ::new()` is very much an implementation issue, some architectures (like Sun Microsystems Sparc RISC microprocessor and the SunSoft C++ Compiler for SunOS 4.x) can support misaligned data structures for which pointers to non-word boundaries are not a problem. If we assume that even if an architecture and available compiler supports misalignment then by definition it supports alignment. Hence the forced alignment of the `PrefixHeader` data structure.

Here's our alternative of `operator ::new()`

```
void * operator new( size_t size )
{
  PrefixHeader *ptr =
       _allocate_memory( "new", size );
  if (ptr == 0) return (0);
  return ( static_cast<void*>(ptr+1) );
  // The same as ((void*) &ptr[1])
}
```

The implementation is fairly straight forward, except for using a static cast to change the type between the *PrefixHeader* * to the *void* * type. Here are the rest of the global `operators new` and `delete` functions:

```
void * operator new [] ( size_t size )
{
  PrefixHeader * ptr =
     _allocate_memory("new []", size );
  if (ptr == 0) return (0);
  return ( static_cast<void*>( ptr+1));
}

void operator delete( void *input_ptr )
{
  if (input_ptr == 0) return;
  PrefixHeader *ptr =
       (static_cast<PrefixHeader*>
            (input_ptr)) - 1;
  _deallocate_memory("delete", ptr );
}

void operator delete []
                   ( void *input_ptr )
```

```
{
  if (input_ptr == 0) return;
  PrefixHeader *ptr =
       (static_cast<PrefixHeader*>
            (input_ptr)) - 1;
  _deallocate_memory("delete []",ptr );
}
```

These are, then, our replacement debuggable global `::new` and `::delete` operators. Notice that the code that allocates and deallocates dynamic memory is shared between the `new` operators likewise for the `delete` operators:

```
PrefixHeader *_allocate_memory(
 const char * func_name,
 size_t size)
{
  // `malloc(0)' is unpredictable
  if (size == 0) size = 1;

  // Allocate memory with real size
incorporating the
  // prefix header record and suffix
record
 size_t real_size = PREFIX_HEADER_SIZE +
size + SUFFIX_HEADER_SIZE ;
 PrefixHeader *ptr =
(PrefixHeader*)malloc( real_size );

 while (ptr == 0)
 {
   // Malloc failed call the new handler
   // to try to free up memory or
   // terminate the application by
   // throwing an exception or calling
   // `abort()'
   (*_dbgnew_handler)();
   ptr = (PrefixHeader*)malloc(
real_size);
  }

  // Mark Prefix and Suffix Memory Block
  // and record the size
  ptr->size = size;
  ptr->magic_word = PREFIX_MAGIC_ID;
  mark_memory( ptr, size );

  // Mark the middle
  memset( (void*)(ptr+1), 'U', size );

  return (ptr);
}
```

The allocator function follows the conventional implementation of the `new` operator (see [5]). It is no surprise that it calls `malloc()` to perform the dynamic allocation of block from the heap. The C functions `malloc()` and `free()` deal with uninitialised memory. The default implementation of `::new` & `::delete` in many native C++ compilers uses `malloc()` and `free()`, the C primitive functions. (And if you think hard about it the C++ compiler developer wants to remain compatible with C. Nevertheless mixing `malloc` and `new` in an ordinary C++ application is still very poor style.) On the other hand your compiler may call another internal function. See notes at the end of the article for details.

The default `new_handler()` function in the debuggable `::operator new()` throws an exception class `bad_alloc`. Here we do *not* deviate from the C++ standard. Other new exception classes augment the standard to indicate more failed memory conditions. [4]

The allocator stores a special unrecognised byte `U' (which is equivalent to 0x55 in hexadecimal) in the user part (the middle). If you are in the middle of a debugging session and happen to see a lot 'U' characters printed out, when you are examining variables, then you could be looking at uninitialised memory! You could also choose another byte value for your system if feel it would be more appropriate. If you happen to know assembly language and the machine code of your target system, then one can go further and use *a byte sequence*. The Editor suggests 0xCC, better known as "Int 3" for Intel machines.

```
Void _deallocate_memory(
  const char *func_name,
  PrefixHeader *ptr )
{
  if (ptr == 0)
  {
#ifdef CHECK_DELETE_ZERO
    cerr << "(*DbgNew*) " << func_name
    << "() : corrupted nil pointer"
    << endl;
#endif
    return;
  }

  if ( diagnose_memory( ptr )
        != MEMORY_OK )
    // If a failed diagnosis returned
    // then do not free pointer!
    return;

  // Unmark the memory.
  unmark_memory( ptr );

  free( ptr );  // Release it
}
```

The deallocator function rejects the input if the input pointer is null and it does *not* normally report this as an error otherwise the function returns immediately. If the input pointer is non-zero then the deallocator will check the actual block to see if it is invalid. The call to `diagnose_memory()` performs this operation. If the memory block is invalid, a diagnosis will be reported (and an exception class object thrown). If the input pointer is legitimate, it refers to previously allocated block, then the memory is safe to unmark and only then it is finally released by calling `free()`.

The check_memory() function does the hard work of examining the memory block. The function returns the memory status as the enumeration type. In particular we:

Look to see if the memory has already been released. If it was unmarked, then this is a double-deletion error.

If our new operator did not allocate the pointer then this is unrecognised pointer error.

Check the prefix header for corruption, if it does not conform then this is a lower bound error.

Check if the suffix header is unmarked, if it does not conform then this is a double-deletion error. This check is done to complete robustness, even though most of double-deletions will be detected in case.

Check the suffix header for corruption, if it does not conform then this is an upper bound error.

If we got through all of the above checks, then the memory block is certified in good health!

```
static MemoryStatus check_memory(
    PrefixHeader *  ptr
)
{
    // Check the memory by:
    register int j,k;

    for (j=0,k=0; j<ALIGNMENT_SIZE;++j)
    If (ptr->ch[j] ==
       free_prefix_string[j])
        ++k;
    if (k==ALIGNMENT_SIZE)
       return (MEMORY_ALREADY_FREED);

    if ( ptr->magic_word !=
       PREFIX_MAGIC_ID )
       return (MEMORY_UNKNOWN);

    for (j=0; j<ALIGNMENT_SIZE; ++j)
       if (ptr->ch[j] !=
       prefix_string[j])
           return (MEMORY_LOWERBOUND);

    SuffixHeader *sptr =
     (SuffixHeader*)( ((char*)ptr) +
     PREFIX_HEADER_SIZE + ptr->size );

    for (j=0,k=0; j<ALIGNMENT_SIZE;++j)
       if (sptr->ch[j] ==
       free_suffix_string[j])
           ++k;
    if (k==ALIGNMENT_SIZE)
       return (MEMORY_ALREADY_FREED);

    for (j=0; j<ALIGNMENT_SIZE; ++j)
       if (sptr->ch[j] !=
       sufix_string[j])
           return (MEMORY_UPPERBOUND);

    // Fine!
    return (MEMORY_OK);
}
```

This naturally leads us to the diagnose_memory() routine, which places a call to check_memory() to examine the returning pointer to heap block. The diagnosis function throws exceptions to correspond to the *test cases* of the first article.

```
static MemoryStatus diagnose_memory(
PrefixHeader *ptr )
{
    // Diagnose memory and raise any
exceptions
    bool            memory_error=false;
    MemoryStatus    status =
       check_memory( ptr );
    switch (status) {
      case MEMORY_ALREADY_FREED:
         cerr << "(*DbgNew*) memory
already freed at ptr:"
              << (void*)(ptr+1) << endl;
         throw AlreadyFreed();
         break;

      case MEMORY_UNKNOWN:
         // Any unknown memory should
         // be reported as an error.
         cerr << "(*DbgNew*) unrecognised
memory at ptr:"
              << (void*)(ptr+1) << endl;
         throw UnknownMemory();
         break;

      case MEMORY_LOWERBOUND:
         cerr << "(*DbgNew*) lower
boundary corruption at ptr:"
              << (void*)(ptr+1) << endl;
         throw LowerboundCorrupted();
         break;

      case MEMORY_UPPERBOUND:
         cerr << "(*DbgNew*) upper
boundary corruption at ptr:"
              << (void*)(ptr+1) << endl;
         throw UpperboundCorrupted();
         break;
    }
    return (status);
}
```

Lastly I will show the marking functions:-

```
static PrefixHeader *mark_memory(
        PrefixHeader *ptr, size_t size )
{
  // Mark the prefix
  ptr->size = size;
  register int j;
  for (j=0; j<ALIGNMENT_SIZE; ++j)
    ptr->ch[j] = prefix_string[j];

  // Mark the suffix
  SuffixHeader *sptr =
    (SuffixHeader*)( ((char*)ptr)+
           PREFIX_HEADER_SIZE + size );
  for (j=0; j<ALIGNMENT_SIZE; ++j)
    sptr->ch[j] = suffix_string[j];
  return (ptr);
}
```

The `mark_memory()` function writes the prefix and suffix identification strings in to the memory block B. The `unmark_memory()` function writes a completely different string, the free memory identification string, in memory once it has been released by the program. This is for the detecting double deletion.

```
static PrefixHeader *unmark_memory(
                   PrefixHeader *ptr )
{
  // Unmark the prefix
  register int j;
  for (j=0; j<ALIGNMENT_SIZE; ++j)
  ptr->ch[j] = free_prefix_string[j];

  // Unmark the suffix
  SuffixHeader *sptr =
    (SuffixHeader*)( ((char*)ptr) +
       PREFIX_HEADER_SIZE + ptr->size);
  for (j=0; j<ALIGNMENT_SIZE; ++j)
    sptr->ch[j]=
       free_suffix_string[j];
  return (ptr);
}
```

We have learned that the global `operator new()` and `operator delete()` can be powerfully overridden so that they can be used to debug memory allocation in C++. The fact that we can provide aternative new and delete functions is very important, because it provides us with a hook to implement garbage collection or some other method of managing dynamic memory for efficiency and productivity.

Caveats

All of the C++ compilers that I have used so far (g++ 2.7.2 and Borland C++ 4.52) use some form of the traditional C memory allocation function such as `malloc()`. Your mileage may indeed vary. If you have difficulty your compiler's manual may provide the answer. New compiler implementations are expected any day soon, once the standard is clarified.

The memory block technique is useful when combined with mix-in debuggable heap space classes that are described by Mr. Scott Meyers [7]. With C++ you need to know what class of object type directly corresponds a section of free space. The language does not provide this feature directly. You have to instrument your own base classes that reference blocks of memory to a particular object class. One idea is to use Run-Time Type Information (RTTI), when faced with the sight of raw memory, but it will only work on virtual object class though.

The article delivered the technique that enables us to solve case 1, 2, 3 and 4. We can detect when a pointer to a block is being freed again. We can detect the case when a pointer to block if it is not recognised. We can detect overrun and underwrites in dynamic memory. Only the fifth case is left.

Peter Pilgrim
Peter.Pilgrim@xenonsoft.demon.co.uk

[1] "Debuggable New and Delete Preamble", Overload 23, Peter A. Pilgrim

[2] "Dynamic Memory Integrity", ACCU/ CVu 8.5, Peter A. Pilgrim and "Dynamic Memory Tracking", ACCU/ CVu 8.6, Peter A. Pilgrim

[3] "Advanced C++ Programming Styles and Idioms": Ch 3.6, James O. Coplien, and Pub. Addison Wesley

[4] "C++ Primer" : 2nd Ed : The `new' operator pg. 144 -150, Stanley B. Lippman, Pub. Addison Wesley

[5] Section 8: Adhere to convention when writing `new`, "Effective C++" : Pub. by Addison Wesley 1992, Scott Meyers

[6] "The C Programming Language", Brian Kernighan & Denis Ritchie, published by Prentice-Hall. Borrowed 2nd Edition (section 8.7 describes a useful implementation of `malloc()`).

[7] Item 27: Requiring or prohibiting heap-based objects, "More Effective C++" Pub. by Addison Wesley 1997, Scott Meyers. (Have a look at Scott’s `HeapTracked` class on page 154 and item 28 too).

Beyond ACCU... C++ on the 'net

Pure Software Engineering

The Experimental Software Engineering Group at the University of Maryland is investigating new engineering methodologies and paradigms.

http://cs.umd.edu/projects/SoftEng/tame

The Software Engineering Institute at Carnegie Mellon has a mission of developing and deploying new approaches to software developments. Their most notable contribution in the past thirteen years has been the Capability Maturity Model.

http://www.sei.cmu.edu/

With all this web browsing aside, try "Journey of the Software Professional" (ISBN 0-13-236613-4). It's not easy going but it's very, very relevant. It's part of the excellent Prentice Hall Software Engineering series:

http://www.prenhall.com

Other high quality publishers include Addison Weseley, and Morgan Kaufman.

http://www.mkp.com/ and
http://www2.awl.com/cseng/

As always, a great starting point for all research is Yahoo's category lists:

http://www.yahoo.co.uk/Computers_and_Internet/Software/Institutes/

Applied Software Engineering

Lead by software project management guru Tom DeMarco, The Atlantic Systems Guild provides various consultancy services. The seven principles are regular contributers to IEEE Software and JOOP. Their site contains an excellent selection of articles and links to related sites.

http://www.atlsysguild.com/

Key software engineering checklists, published articles and extracts from Steve McDonnell's books "*Code Complete*" (ISBN 1-55615-484-4), "*Rapid Development*" (ISBN 1-55615-900-5) are available from the author's web site.

http://www.construx.com/stevemcc

Implementation resources

The Microsoft Development Network is an excellent source of articles and references for Windows developers. It can be accessed on-line freely, in return for filling in online information forms.

http://www.microsoft.com/msdn

The Linux Documentation Project continues its Herculean effort at.

http://sunsite.unc.edu/mdw/linux.html

Novell provide developer support services for their Network Operating System platform and various services.

http://developer.novell.com/cgi-bin/devnet

And, of course purely in the interests of balance, Netscape provide extensive SDK documentation for their server platforms on the DevEdge Online web site.

http://developer.netscape.com/index_home.html

The Mozilla Organisation is acting as a focal point for the source code release of the now public domain free Netscape browser.

http://www.mozilla.org/

UseNet

There are a number of relevant and useful UseNet news groups. The unmoderated tend to receive way too many messages, most of which tend to be noise. But, this is the heritage of the net…

Testing.	comp.software.testing.
Configuration management	comp.software.config-mgt
Software Engineering.	comp.lang.software-eng

Group FAQ's available from:

http://www.lib.ox.ac.uk/internet/news/faq/by_group.index.html.

As ever, topic and site suggestions to Ian Bruntlett, ibruntlett@libris.co.uk

Credits

Editor

John Merrells
merrells@netscape.com

Einar Nilsen-Nygaard
65 Beechlands Drive
Clarkston, GLASGOW, G76 7UX.
UK

P.O. Box 2336,
Sunnyvale, CA 94087-0336,
U.S.A.

Readers

Ray Hall
Ray@ashworth.demon.co.uk

Ian Bruntlett
ibruntlett@libris.co.uk

Einar Nilsen-Nygaard
EinarNN@atl.co.uk
einar@rhuagh.demon.co.uk

Production Editor

Alan Lenton
alan@ibgames.com

Advertising

John Washington
accuads@wash.demon.co.uk
Cartchers Farm, Carthouse Lane
Woking, Surrey, GU21 4XS

Membership and Subscription Enquiries

David Hodge
davidhodge@compuserve.com
31 Egerton Road
Bexhill-on-Sea, East Sussex. TN39 3HJ

Copyrights and Trademarks

Copy deadline

All articles intended for inclusion in *Overload 26* should be submitted to the editor, John Merrells < merrells@netscape.com>, by May 5th.