What happens when resources change hands in C++? **Kevlin Henney** explains how changing ownership of memory and file handles can be automated efficiently and easily

One careful owner

Kevlin Henney is an independent software development consultant and trainer. He can be reached at **www.curbralan.com**

THE LAST COLUMN[1] EXPLORED C++'S IDIOMATIC approach to exception-safe acquisition and release of resources, such as memory, file handles, mutex locks and wait cursors. Rather than extend the language – along with the length of the source code required to make code safe – C++'s exception safety relies on a more object-oriented solution: the resource is wrapped up in an object that automates its release, rather than requiring the user to interlace code with finally blocks. This refactoring has the additional, pleasant effect of reducing the amount of code needed to work with the resource in the first place.

Scoping with exceptions

You should never see the following pattern (in the sense of textual pattern) in your code:

```
{
  resource_type resource = acquire();
  try
  {
    .... // use resource, exceptions possible
  }
  catch(...)
  {
    release(resource); // clean up on exception
    throw; // rethrow so caller is aware of failure
  }
  release(resource); // clean up on normal exit
}
```

It can be resolved by the following pattern (in the sense of idiomatic design pattern):

```
{
  scoped<resource_type, resource_releaser> resource;
  .... // use resource, exceptions possible
}
```

Where scoped (see Listings 1, 2 and 3) was the class template explored in the last column, and resource_releaser is a simple function-object type used as the clean-up policy:

```
struct resource_releaser
{
  void operator()(resource_type resource) const
  {
    release(resource);
  }
};
```

A default policy is supplied for the common case of cleaning up memory, so the user doesn't need to define or explicitly supply a deleter type. However, the use of a function-object type makes scoped more generally applicable than the standard auto_ptr class template, which can only work in terms of pointers to single objects and delete. This is an over-simplifying assumption that's inappropriate for many mature object designs. For instance, an object may be allocated by a factory rather than using new directly in the user code. To dispose of the object, it should be returned to the factory rather than having delete called on it directly:

```
class product {...};
class factory
{
public:
  product *create();
  void dispose(product *);
  ...
};
```

std::auto_ptr has nothing to offer us in this situation, but scoped can be parameterised easily if supplied with an appropriate function-object type:

```
class disposer
{
public:
  disposer(factory *creator)
: creator(creator)
  {
  }
  void operator()(product *disposee)
  {
    creator->dispose(disposee);
  }
private:
  factory *creator;
};
```

The benefit of small classes again becomes apparent when you see how much code you don't need to write:

```
void example(factory *creator)
{
  scoped<product *, disposer> ptr(
```

```
        creator->create(), disposer(creator));
    .... // use ptr, exceptions possible
}
```

When a pointer is used as the target, scoped looks and feels like a smart pointer, supporting the appropriate operators and conversions.

Exchanging goods

The scoped class template is simple, but extensible and general-purpose. In its current form, however, it is rigidly tied to always releasing the resource it acquired at the beginning of its life. Suppose you intend to use scoped to simplify the implementation of a handle–body class arrangement[2,3]:

```
class handle
{
public:
    handle()
: self(new body)
    {
    }
    handle(const handle &other)
: self(new body(*other.self))
    {
    }
    ~handle()
    {
        delete body;
    }
    ....
private:
    class body {....};
    body *self;
    ....
};
```

On the surface, this appears to present no problems:

```
class handle
{
public:
    handle(); // as before
    handle(const handle &); // as before
    ~handle()
    {
    }
    ....
private:
    class body {....};
    scoped<body *> self;
    ....
};
```

If handle supports assignment, there's a simple, idiomatic implementation that's both exception and self-assignment safe for the raw pointer version[4]:

```
handle &operator=(const handle &rhs)
{
    body *old_self = self;
    self = new body(*rhs.self);
    delete old_self;
    return *this;
}
```

It's not clear, however, how the revised version should be implemented, as scoped disables assignment and has no other way of exchanging its goods.

It's tempting to try to figure out how to make scoped assignment do the right thing, so that the above idiom still works. However, as with many tempting solutions, this is the wrong one to pursue: there are no implementations of operator= that can both satisfy user expectations of assignment and have exclusive resource management semantics. In a classic misuse of operator overloading, std::auto_ptr offers such a facility:

```
std::auto_ptr<type> ptr(new type); // ptr is non-null
std::auto_ptr<type> qtr; // qtr is null
qtr = ptr; // qtr is non-null and ptr is null
```

The path towards the solution – and away from this basic violation of the principle of least astonishment ("when in doubt, do as the ints do"[5]) – comes from appreciating an alternative idiom for writing assignment operators based on swap functions[6]:

```
template<
    typename resource_type, typename destructor = deleter>
class scoped
{
    ....
    void swap(scoped &other)
    {
        std::swap(resource, other.resource);
        std::swap(destruct, other.destruct);
    }
    ....
};
```

Adding this feature to scoped now allows a simple and safe implementation of handle's assignment operator:

```
handle &operator=(const handle &rhs)
{
    handle copy(rhs); // take a copy of the RHS
    self.swap(rhs.self); // exchange representations
    return *this;
} // copy's destructor cleans up old representation
```

Premature release

The swapping technique liberalises scoped's capabilities at no cost to its representation. Once a scoped object has acquired a resource, it will be released by that scoped object or one of its kin. This is fine in most cases, but there are times when we want to use scoped in a transactional style. For example, in the event of failure, we want to roll back cleanly – that is, clean up resources to avoid leaks – otherwise we want to commit to a given state and disable any clean-up.

Consider the following exception-unsafe code:

```
void example(vector<type *> &ptrs)
{
    ptrs.push_back(new type);
}
```

This is unsafe, because if the call to push_back throws an exception from lack of memory, the newly created object will be forgotten and lost. There are a number of workarounds, including reserving space in advance or making the code exception-aware (and verbose) with try and catch. Perhaps the most elegant is the transactional style, but in its current form, scoped can't help us all the way:

```
void example(vector<type *> &ptrs)
{
   scoped<type *> ptr(new type);
   ptrs.push_back(ptr);
} // object pointed to by ptr deleted... bad news for ptrs!
```

What is needed is a way of relieving ptr of its custodial duties when no exceptions are thrown – something that can play the role of a commit, like the following:

```
void example(vector<type *> &ptrs)
{
   scoped<type *> ptr(new type);
   ptrs.push_back(ptr);
   ptr.release(); // commit the existence of the new object
}
```

The following revisions take into account this requirement, the need for acquisition after creation, and the capability for default construction:

```
template<
   typename resource_type, typename destructor = deleter>
class scoped
{
public: // structors
   scoped()
: to_destruct(false)
   {
   }
   explicit scoped(
      resource_type resourced,
      destructor on_destruct = destructor())
: to_destruct(true), ....
   {
   }
   ~scoped()
   {
      if(to_destruct)
         destruct(resource);
   }
public: // acquisition and release
   void swap(scoped &other)
   {
      std::swap(to_destruct, other.to_destruct);
      ....
   }
   void acquire(resource_type new_resource)
   {
      resource_type old_resource = resource;
      resource = new_resource;
      if(to_destruct)
         destruct(old_resource);
      to_destruct = true;
   }
   resource_type release()
   {
      to_destruct = false;
      return resource;
   }
   ....
private: // representation
   bool to_destruct;
   ....
};
```

Safe transfer of goods

With only a small addition to the interface, users appear to have as much control over the behaviour of scoped as they might need. However, there's still one use that could be made safer – returning a result from a function. In its current form, if you wish to return a resource currently guarded by a scoped object to the caller of a function, you have to release it and return the raw resource:

```
type *example()
{
   scoped<type *> result(new type);
   ....
   return result.release();
}
```

The caller can pick it up as follows:

```
type *result = example();
```

This appears to be fine, until you consider one constraint that C++ does not enforce – function return values can be ignored:

```
example(); // memory leak
```

Again, the temptation is to solve the wrong problem: how can we make scoped objects copyable so they can be used as return values? This approach led auto_ptr into deep, hot water, endowing it with unnecessarily subtle, surprising and suspicious copy construction semantics. For more details on the interface and implementation of auto_ptr, see Nicolai Josuttis's *C++ Standard Library* book[7].

Transfer of ownership isn't something we should aim to hide. Instead, it should be explicit in a function's interface as well as in its body. We can introduce an intermediate, proxy-like object to represent the token of ownership. A revised version makes the ownership transfer more explicit:

```
scoped<type *>::result example()
{
   scoped<type *> result(new type);
   ....
   return result.transfer();
}
```

The caller's code remains unaffected. In essence, if the temporary result object undergoes conversion, it assumes it has been picked up and a new owner has been found for the resource. If it's ignored and forgotten, it will clean up the resource (see Listing 4).

As a scoped<...>::result object is passed around, the token of ownership is passed with it. This necessitates the use of mutable to negotiate easy passage, but this const comprise is contained within its intended usage. It's designed for transfer purposes only, rather than general resource management, which is the responsibility of scoped.

Simple and general

In spite of the increased complexity, arising mostly from the safe transfer semantics, the scoped template is still simpler to use and implement than std::auto_ptr, and is more generally applicable.

Exception safety can be mastered with relatively few idioms, but it's understanding interface usage and design that poses the greatest challenges, not the mechanisms involved. It's the challenge of transferring ownership that turned the C++ standard library's auto_ptr class template into a hotbed of questionable design, turning a simple and obvious concept into something complex and subtle. The challenge was left as an exercise for the reader at the end of the last column. A clear separation of roles and responsibilities, making the explicit truly explicit rather than hidden and subtle, has led to a clearer and more responsible abstraction. ■

References

1. Kevlin Henney, Making an Exception, *Application Development Advisor*, May 2001
2. James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992
3. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
4. Kevlin Henney, Creating Stable Assignments, *C++ Report* 10(6), June 1998, is also available at www.curbralan.com
5. Scott Meyers, *More Effective C++*, Addison-Wesley, 1996
6. Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000
7. Nicolai Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 1999

Listing 1: The scoped class template for exception safety

```
template<
   typename resource_type, typename destructor = deleter>
class scoped
{
public: // structors
   explicit scoped(
      resource_type resourced,
      destructor on_destruct = destructor())
: resource(resourced), destruct(on_destruct)
   {
   }
   ~scoped()
   {
      destruct(resource);
   }
public: // conversions
   operator resource_type() const
   {
      return resource;
   }
   resource_type operator->() const
   {
      return resource;
   }
   dereferenced<resource_type>::type operator*() const
   {
      return *resource;
   }
private: // prevention
   scoped(const scoped &);
   scoped &operator=(const scoped &);
private: // representation
   resource_type resource;
   destructor destruct;
};
```

Listing 2: The deleter function-object type for single object deletion

```
struct deleter
{
   template<typename deleted_type>
   void operator()(deleted_type to_delete) const
   {
      delete to_delete;
   }
};
```

Listing 3: The dereferenced traits type for determining the return type of operator*

```
template<typename value_type>
struct dereferenced
```

```
{
   typedef void type;
};

template<typename element_type>
struct dereferenced<element_type *>
{
   typedef element_type &type;
};

template<>
struct dereferenced<void *>
{
   typedef void type;
};
```

Listing 4: Housekeeping after a change of ownership

```
template<
   typename resource_type, typename destructor = deleter>
class scoped
{
   ....
   class result
   {
   public: // structors and access
      result(
         resource_type resourced,
         destructor_type on_destruct)
: to_destruct(true),
         resource(resourced),
         destruct(on_destruct)
      {
      }
      result(const result &other)
: to_destruct(other.to_destruct),
         resource(other.resource),
         destruct(other.destruct)
      {
         other.to_destruct = false;
      }
      ~transfer()
      {
         if(to_destruct)
            destruct(resource);
      }
      operator resource_type()
      {
         to_destruct = false;
         return resource;
      }
   private: // representation
      mutable bool to_destruct;
      resource_type resource;
      destructor_type destruct;
   };
   result transfer()
   {
      to_destruct = false;
      return result(resource, destruct);
   }
   ....
};
```