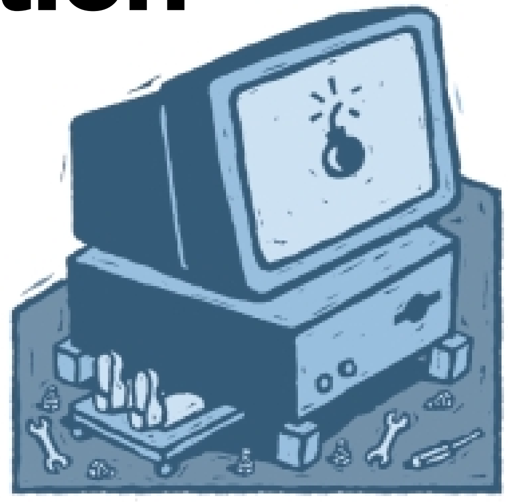
Exception handling in C++ can save your program from digital death, but it must be treated with care. **Kevlin Henney** explains how to make your programs exception-safe

Making an exception



NO MATTER HOW GOOD OUR INTENTIONS are, bad things happen. This is as true of programs at runtime as it is in everyday life. The reality differentiates industrial-strength code from simple, toy code examples. But handling the bad along with the good is where things can get ugly. There's always more that can go wrong than can go right, so traditional error handling often chops up and obscures normal logic, making it hard to trace the original intent.

The intricacy of error handling has sometimes driven programmers blindly into the arms of the goto, which has an effect on code similar to that of the mile-a-minute plant on gardens. To those who are not forewarned, it initially looks harmless enough, but before long it's completely taken over. And it takes more than a pair of garden shears to disentangle either your code or your garden.

This is where exceptions fit in. C++'s exception handling mechanism is a marriage between its object model and its control-flow features. It supports a separation between the normal flow of control (the try block) and the handling of the abnormal and abominable (the catch handlers). It also simplifies and broadens the communication of the bad news (the throw), just as a goto does not. It's true that exceptions inflict a certain amount of violence on the normal, gentle flow of control, but they do so sympathetically. Exceptions support rather than obstruct the block structure and modularity of your code, whereas "a goto completely invalidates the high-level structure of the code"[1], making it difficult to read, refactor or rescue.

Exceptions also have a dark side. If code isn't written to be exception-safe, an exception will lead to graceless rather than graceful failure – the act of delivering bad news creates worse news. Code that releases a previously acquired resource, for example, should be executed regardless of the route out of a function. Code not written with exceptions in mind normally fails to do the right thing when they happen.

An object finale

The common focus of exception handling is on separating the normal and exceptional paths. In contrast, exception safety focuses on identifying code that's common to both paths. In the case of paired actions, such as *acquire–release* or *open–close*, an exception-safe approach ensures the bracketing action is always executed. Consider the following code:

```
{
    resource_type resource = acquire();
    .... // use resource, exceptions possible
    release(resource);
}
```

There are two things to notice about this. It's brief and clear, and it's unsafe, with the release action bypassed if an exception is thrown in the body of the block. The first instinct of many programmers is to make the code exception-aware in order to make it safe:

```
{
    resource_type resource = acquire();
    try
    {
        .... // use resource, exceptions possible
    }
    catch(...)
    {
        release(resource);
```

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

FACTS AT A GLANCE

- **C++'s exception handling mechanism respects a program's modularity when it transfers control**
- **Exceptions can cause problems if code is not exception safe**
- **Some languages use a finally block to allow clean up of resources on block exit**
- **The C++ idiom is to use objects to encapsulate actions rather than extend the language itself**
- **A simple, extensible class template can be written to handle many common clean-up tasks**

```
        throw; // rethrow so caller is aware of failure
    }
    release(resource);
}
```

There are three things to notice about this code: it's verbose, exception-safe and, because of its verbosity, it's not immediately obvious that it's exception-safe. It's not enough that code should be exception-safe, either – it should also be clear that it's exception-safe. The code above demonstrates a simple scenario with a single resource and single release action. Include more of either and it looks like the mile-a-minute has returned to take over your code.

And finally...

Some languages that adopt an exception handling mechanism like C++'s also introduce a finally block. For example, Java and C# work this way. A finally block groups code common to both execution paths into a single block. C++ doesn't have finally, but if it did, it would look something like the following:

```
{
    resource_type resource = acquire();
    try
    {
        .... // use resource, exceptions possible
    }
    finally // not legal C++
    {
        release(resource);
    }
}
```

This is clearly briefer and briefly clearer than the previous, tortured code fragment. Is it an oversight that C++ doesn't support such a construct? On the contrary, it's by design[2]. Although finally reduces the amount of code duplication within a single function, it still bloats the source of the function and often leads to code duplication when the resource type is used in many places. The same finally and release code are repeated everywhere the resource is used.

C++ in addition has a feature that neither Java nor C# supports – deterministic object lifetime and destruction. An ordinary local variable will have its destructor called when it goes out of scope, both for normal exit and as the stack unwinds from an exception. Destructors are not just for cleaning up the resources used by the destructing object, they can be used to clean up other objects' resources as well.

Thanks for the memory

The most common example of this idiom is in managing memory that is allocated only for a block, although it can also be used to automate lifetime management for nested objects. This is one of the roles that the standard auto_ptr class template is designed to cover:

```
{
    std::auto_ptr<type> ptr(new type);
    .... // use ptr, exceptions possible
}

As opposed to:

{
    type *ptr = new type;
    try
    {
        .... // use ptr, exceptions possible
    }
    catch(...)
    {
        delete ptr;
        throw;
    }
    delete ptr;
}
```

auto_ptr is clearly a winner in this department. However, beyond this simple scenario, it's let down by some rather messy semantics. auto_ptr's specification passes all understanding and defies implementation. It's clear from the scenario already outlined that such acquisition objects should not be copyable, as this would lead to aliasing and attempted multiple deletion of a single object. However, after a few rounds with standardisation politics, what was once a simple class that could be implemented by a C++ novice grew into a monster that supported *move* rather than *copy* semantics for its copy operations – a handful of C++ experts, and not everyone involved with the standardisation, can implement it. If you want surprising semantics, consider the following:

```
{
    std::auto_ptr<type> ptr(new type), qtr;
    qtr = ptr;
    ....
}
```

There is an expectation that assignment leaves the right-hand side of the assignment unaffected. Alas, auto_ptr fails to live up to expectations: the right-hand side is set to null. While such transfer functionality is sometimes useful, it isn't copy assignment. Sadly, this is the tip of the iceberg with this class template's problems.

Let's go back to first principles, see what's involved in designing such a class for its original use in simplifying block-scoped memory management, and then generalise it to other resources while retaining its essential simplicity. To bind object deletion to block scope, we need a constructor to be told about the object, a destructor to destroy it and no copying operations (because that way lies madness, darkness and the shadow of std::auto_ptr):

```
template<typename resource_type>
class scoped
{
public: // structors
    explicit scoped(resource_type resourced)
      : resource(resourced)
    {
    }
    ~scoped()
    {
        delete resource;
    }
```

The solution to the exception-safety problem is to give custody of the resource to a locally scoped object which then performs the clean-up on destruction. This solution is a more object-oriented one: the control flow is abstracted into an object whose responsibility is clear and whose action is reusable. The resulting code is exception-neutral, meaning that it's safe, but not cluttered with try blocks of any kind, and requires no extension to the language. The technique is well known and well documented in a number of forms. It is commonly known as the RESOURCE ACQUISITION IS INITIALIZATION idiom[3], although this name is something of a misnomer: the idiom has nothing to do with initialisation and everything to do with finalisation.

```
private: // prevention
    scoped(const scoped &);
    scoped &operator=(const scoped &);
private: // representation
    resource_type resource;
};
```

scoped is simple in both implementation and use:

```
{
    scoped<type *> ptr(new type);
    ....
}
```

A change of policy

The precondition to using scoped as it currently stands is that the pointer must either be null or have been allocated using new. If you're using a different allocator, such as malloc or new[], then delete is the wrong deallocation. Instead, you need free and delete[] respectively. Rather than write a separate class template for each possibility, it would be simpler to allow scoped to be parameterised by a destruction policy. A policy class describes a STRATEGY[4] that can be used as a template parameter to control some aspect of the template's behaviour. By default, we tend to work with single objects rather than arrays, which suggests the following usage:

```
{
    scoped<type *> ptr(new type);
    scoped<type *, array_deleter> array(new type[size]);
    ....
}
```

Treating the policy as a function object type means we get the desired flexibility, a simple syntax to use inside scoped, and the possibility of passing in a policy object that has its own state. Here is the revised scoped template:

```
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);
    }
private: // prevention
    scoped(const scoped &);
    scoped &operator=(const scoped &);
private: // representation
    resource_type resource;
    destructor destruct;
};
```

The default object deletion policy and function object type are simple:

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

A member template is used so that this simple class will work with any pointer type. The alternative would be to make the whole class a template, which would make its use verbose: to use it, you would have to spell out the class template name and its parameter types, rather than just use the class name and let the compiler deduce the type from the argument of the function call operator – deleter<type *> as opposed to deleter.

As a function-object type, deleter can be reused in contexts other than scoped. For instance, consider the common need to delete each object pointed to by a container:

```
std::list<type *> objects;
objects.push_back(new type);
....
```

Rather than the following, slightly long-winded loop:

```
for(std::list<type *>::iterator at = objects.begin();
    at != objects.end();
    ++at)
{
    delete *at;
}
```

You can write the following:

```
std::for_each(objects.begin(), objects.end(), deleter());
```

Given deleter, array_deleter is quite straightforward:

```
struct array_deleter
{
    template<typename deleted_type>
    void operator()(deleted_type to_delete)
    {
        delete[] to_delete;
    }
};
```

A sense of closure

Not all pointers want deleting. Some have more complex finalisation. If, for compatibility reasons, you end up using C-style I/O, you may be using FILE * with fopen and fclose from the C standard library. In this case, the pointer is 'allocated' using fopen and 'deallocated' using fclose. We can provide a simple closer function object type to handle this:

```
struct closer
{
    void operator()(FILE *to_close)
    {
        fclose(to_close);
    }
};
```

And to use it, scoped needs no change:

```
{
    scoped<FILE *, closer> file(fopen(name, mode));
    ....
}
```

Now, the way that I have written the scoped template allows any type to be used as a resource, not just pointers. For instance, you can use scoped with an int. An int? Why would an int need tidying up in the event of an exception? In the POSIX API, int is the type used for file handles. So, although the int itself does not need any extra management, an open needs to be matched with a close. Given that 'close' is the most common name for any close action, we can generalise the closer function object type beyond just FILE * and int:

```
struct closer
{
   template<typename closed_type>
   void operator()(closed_type to_close)
   {
      close(to_close);
   }
   void operator()(FILE *to_close)
   {
      fclose(to_close);
   }
};
```

This now takes the appropriate action depending on the type of the resource:

```
{
   scoped<FILE *, closer> file(fopen(name, mode));
   ....
} // fclose called
{
   scoped<int, closer> file(open(name, flags));
   ....
} // close called
```

Getting a result

As it stands, the scoped template is simple, flexible – and almost useless. The only public interface it currently sports is a constructor and a destructor. This is something of a write-only interface: there's no way to get your hands on the actual resource to use it.

We can provide a simple outward conversion with a user-defined conversion operator:

```
template<
   typename resource_type, typename destructor = deleter>
class scoped
{
   ....
   operator resource_type() const
   {
      return resource;
   }
   ....
};
```

For the common case of just getting your hands on the value of the resource, the implicit conversion makes the use of scoped practically transparent:

```
{
   scoped<FILE *, closer> file(fopen(name, mode));
   ....
   fflush(file); // implicit conversion to FILE *
   ....
}
```

Except for pointers. For instance, auto_ptr, like most other smart pointers, supports operator* and operator-> for dereferencing. The arrow operator is easy enough:

```
template<
   typename resource_type, typename destructor = deleter>
class scoped
{
   ....
   resource_type operator->() const
   {
      return resource;
   }
   ....
};
```

If resource_type is a pointer, or another class that supports its own operator->, the operator-> just defined can be used. If not, any attempt to use it will result in a compile time error. So the code is valid when it's supposed to be, and isn't when it isn't, which is just the level of type checking we want.

operator* presents us with a bit more of a challenge: what's the return type in the case of a pointer-based resource, and how do we prevent it being used with non-pointer resources? How do we get from resource_type to the type of *resource? C++ doesn't support a typeof operator, which would make this trivial, so we resolve the situation using the TRAITS idiom. A trait class allows compile-time deduction of features of another type based on template specialisation. We are going to define a simple, single-feature trait that allows us to deduce a legal result type for operator*:

```
template<
   typename resource_type, typename destructor = deleter>
class scoped
{
   ....
   dereferenced<resource_type>::type operator*() const
   {
      return *resource;
   }
   ....
};
```

The dereferenced trait class has a nested typedef that resolves to the right type. For non-pointer types, operator* is unusable because the *resource expression isn't legal. Template functions aren't fully checked and compiled unless they are used, but they still need to have legal signatures. In this case, a return type of void will do the trick. So, the default implementation for dereferenced does just that:

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

For pointer types, we can use partial specialisation to offer the dereferenced result type we are expecting:

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

This specialisation effectively overrides the primary template for all types that match form T *, and which dereferenced would therefore be T &. The only other case we need to specialise for is that of void *. There's no such thing, however, as a reference to void. In the event that we are working at a low level, we may find ourselves working with raw memory, but we still want the safety and convenience of scoped at our disposal. At the moment, the compiler will reject any attempt to use scoped<void *>, as indeed it does for auto_ptr<void>, because the type void & isn't well formed. We can add a full specialisation for this case, silencing the compiler and disabling use of operator* while still offering the core scoped features:

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

There are many benefits to the traits-based approach, not least of which is that it makes the scoped template's operator* work! Another is that traits are open to further specialisation by other users. For instance, if you have your own smart pointer class that you would like to use with scoped, you could specialise dereferenced accordingly.

Exceptions are a simple mechanism that can be used to send bad news from one part of a program to another. They can also be used to make bad news if the programmer isn't aware of a handful of simple exception-safe practices. Rather than introducing a finally block to guarantee resource release, C++ encourages a more object-oriented approach that abstracts control flow and reduces code duplication. Exception-neutral code that uses this technique is safer and more concise than exception-aware code that laboriously addresses each outcome in turn. The simple principles and techniques described in this article can be taken to quite an advanced level[5,6].

auto_ptr is useful for the simple task of scope-managed memory, but it doesn't generalise beyond delete of new-allocated objects. It is further complicated for both the user and the standard library implementor by its move semantics for copying. So, if nothing else is available, use auto_ptr. However, you might want to experiment with other libraries, such as Boost[7], or the scoped template presented here. The only feature currently missing from scoped is the ability to transfer ownership of a resource: this is left as an exercise for the reader. ■

References

1. *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994 http://pcroot.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_1.html.
2. Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994
3. Bjarne Stroustrup, *The C++ Programming Language*, Third edition, Addison-Wesley, 1997
4. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
5. Kevlin Henney, "C++ Patterns: Executing Around Sequences", *EuroPLoP 2000*, July 2000, www.curbralan.com.
6. Andrei Alexandrescu and Petru Marginean, "Generic<Programming>: Simplifying Your Exception-Safe Code", C/C++ Users Journal web site, December 2000, www.cuj.com/experts/1812/alexandr.htm.
7. Boost library web site, www.boost.org.