6 Memory and Resource Management

C++ offers tremendous flexibility in managing memory, but few C++ programmers fully understand the available mechanisms. In this area of the language, overloading, name hiding, constructors and destructors, exceptions, static and virtual functions, operator and non-operator functions all come together to provide great flexibility and customizability of memory management. Unfortunately, and perhaps unavoidably, things can also get a bit complex.

In this chapter, we'll look at how the various features of C++ are used together in memory management, how they sometimes interact in surprising ways, and how to simplify their interactions.

Inasmuch as memory is just one of many resources a program manages, we'll also look at how to bind other resources to memory so we can use C++'s sophisticated memory management facilities to manage other resources as well.

Gotcha #60: Failure to Distinguish Scalar and Array Allocation

Is a `Widget` the same thing as an array of `Widgets`? Of course not. Then why are so many C++ programmers surprised to find that different operators are used to allocate and free arrays and scalars?

We know how to allocate and free a single `Widget`. We use the `new` and `delete` operators:

```
Widget *w = new Widget( arg );
// . . .
delete w;
```

Unlike most operators in C++, the behavior of the `new` operator can't be modified by overloading. The `new` operator always calls a function named `operator` new to (presumably) obtain some storage, then may initialize that storage. In the case of Widget, above, use of the new operator will cause a call to an operator new function that takes a single argument of type size_t, then will invoke a Widget constructor on the uninitialized storage returned by operator new to produce a Widget object.

The delete operator invokes a destructor on the Widget and then calls a function named operator delete to (presumably) deallocate the storage formerly occupied by the now deceased Widget object.

Variation in behavior of memory allocation and deallocation is obtained by overloading, replacing, or hiding the functions operator new and operator delete, not by modifying the behavior of the new and delete operators.

We also know how to allocate and free arrays of Widgets. But we don't use the new and delete operators:

```
w = new Widget[n];
// . . .
delete [] w;
```

We instead use the new [] and delete [] operators (pronounced "array new" and "array delete"). Like new and delete, the behavior of the array new and array delete operators cannot be modified. Array new first invokes a function called operator new[] to obtain some storage, then (if necessary) performs a default initialization of each allocated array element from the first element to the last. Array delete destroys each element of the array in the reverse order of its initialization, then invokes a function called operator delete[] to reclaim the storage.

As an aside, note that it's often better design to use a standard library vector rather than an array. A vector is nearly as efficient as an array and is typically safer and more flexible. A vector can generally be considered a "smart" array, with similar semantics. However, when a vector is destroyed, its elements are destroyed from first to last: the opposite order in which they would be destroyed in an array.

Memory management functions must be properly paired. If new is used to obtain storage, delete should be used to free it. If malloc is used to obtain storage, free should be used to free it. Sometimes, using free with new or malloc with delete will "work" for a limited set of types on a particular platform, but there is no guarantee the code will continue to work:

```
int *ip = new int(12);
// . . .
```

```
free( ip ); // wrong!
ip = static_cast<int *>(malloc( sizeof(int) ));
*ip = 12;
// . . .
delete ip; // wrong!
```

The same requirement holds for array allocation and deletion. A common error is to allocate an array with array new and free it with scalar delete. As with mismatched `new` and `free`, this code may work by chance in a particular situation but is nevertheless incorrect and is likely to fail in the future:

```
double *dp = new double[1];
// . . .
delete dp; // wrong!
```

Note that the compiler can't warn of an incorrect scalar deletion of an array, since it can't distinguish between a pointer to an array and a pointer to a single element. Typically, array new will insert information adjacent to the memory allocated for an array that indicates not only the size of the block of storage but also the number of elements in the allocated array. This information is examined and acted upon by array delete when the array is deleted.

The format of this information is probably different from that of the information stored with a block of storage obtained through scalar new. If scalar delete is invoked upon storage allocated by array new, the information about size and element count—which are intended to be interpreted by an array delete—will probably be misinterpreted by the scalar delete, with undefined results. It's also possible that scalar and array allocation employ different memory pools. Use of a scalar deletion to return array storage allocated from the array pool to the scalar pool is likely to end in disaster.

```
delete [] dp; // correct
```

This imprecision regarding the concepts of array and scalar allocation also show up in the design of member memory-management functions:

```
class Widget {
 public:
   void *operator new( size_t );
   void operator delete( void *, size_t );
   // . . .
};
```

The author of the `Widget` class has decided to customize memory management of `Widgets` but has failed to take into account that array operator new and delete functions have different names from their scalar counterparts and are therefore not hidden by the member versions:

```
Widget *w = new Widget( arg ); // OK
// . . .
delete w; // OK
w = new Widget[n];  // oops!
// . . .
delete [] w; // oops!
```

Because the `Widget` class declares no `operator new[]` or `operator delete[]` functions, memory management of arrays of `Widgets` will use the global versions of these functions. This is probably incorrect behavior, and the author of the `Widget` class should provide member versions of the array new and delete functions.

If, to the contrary, this is correct behavior, the author of the class should clearly indicate that fact to future maintainers of the `Widget` class, since otherwise they're likely to "fix" the problem by providing the "missing" functions. The best way to document this design decision is not with a comment but with code:

```
class Widget {
 public:
   void *operator new( size_t );
   void operator delete( void *, size_t );
   void *operator new[]( size_t n )
       { return ::operator new[](n); }
   void operator delete[]( void *p, size_t )
       { ::operator delete[](p); }
   // . . .
};
```

The inline member versions of these functions cost nothing at runtime and should convince even the most inattentive maintainer not to second-guess the author's decision to invoke the global versions of array new and delete functions for `Widgets`.

Gotcha #61: Checking for Allocation Failure

Some questions should just not be asked, and whether a particular memory allocation has succeeded is one of them.

Let's look at how life used to be in C++ when allocating memory. Here's some code that's careful to check that every memory allocation succeeds:

```
bool error = false;
String **array = new String *[n];
if( array ) {
   for( String **p = array; p < array+n; ++p ) {
       String *tmp = new String;
       if( tmp )
           *p = tmp;
       else {
           error = true;
           break;
       }
   }
}
else
   error = true;
if( error )
   handleError();
```

This style of coding is a lot of trouble, but it might be worth the effort if it were able to detect all possible memory allocation failures. It won't. Unfortunately, the `String` constructor itself may encounter a memory allocation error, and there is no easy way to propagate that error out of the constructor. It's possible, but not a pleasant prospect, to have the `String` constructor put the `String` object in some sort of acceptable error state and set a flag that can be checked by users of the class. Even assuming we have access to the implementation of `String` to implement this behavior, this approach gives both the original author of the code and all future maintainers yet another condition to test.

Or neglect to test. Error-checking code that's this involved is rarely entirely correct initially and is almost never correct after a period of maintenance. A better approach is not to check at all:

```
String **array = new String *[n];
for( String **p = array; p < array+n; ++p )
   *p = new String;
```

This code is shorter, clearer, faster, and correct. The standard behavior of `new` is to throw a `bad_alloc` exception in the event of allocation failure. This allows us to encapsulate error-handling code for allocation failure from the rest of the program, resulting in a cleaner, clearer, and generally more efficient design.

In any case, an attempt to check the result of a standard use of `new` will never indicate a failure, since the use of `new` will either succeed or throw an exception:

```
int *ip = new int;
if( ip ) { // condition always true
   // . . .
}
else {
   // will never execute
}
```

It's possible to employ the standard "nothrow" version of `operator new` that will return a null pointer on failure:

```
int *ip = new (nothrow) int;
if( ip ) { // condition almost always true
   // . . .
}
else {
   // will almost never execute
}
```

However, this simply brings back the problems associated with the old semantics of `new`, with the added detriment of hideous syntax. It's better to avoid this clumsy backward compatibility hack and simply design and code for the exception-throwing new.

The runtime system will also handle automatically a particularly nasty problem in allocation failure. Recall that the `new` operator actually specifies two function calls: a call to an `operator new` function to allocate storage, followed by an invocation of a constructor to initialize the storage:

```
Thing *tp = new Thing( arg );
```

If we catch a `bad_alloc` exception, we know there was a memory allocation error, but where? The error could have occurred in the original allocation of the storage for `Thing`, or it could have occurred within the constructor for `Thing`. In the first case we have no memory to deallocate, since `tp` was never set to anything. In the second case, we should return the (uninitialized) memory to which `tp` refers to the heap. However, it can be difficult or impossible to determine which is the case.

Fortunately, the runtime system handles this situation for us. If the original allocation of storage for the `Thing` object succeeds but the `Thing` constructor fails and throws any exception, the runtime system will call an appropriate `operator delete` (see Gotcha #62) to reclaim the storage.

Gotcha #62: Replacing Global New and Delete

It's almost never a good idea to replace the standard, global versions of `operator new`, `operator delete`, array new, or array delete, even though the standard permits it. The standard versions are typically highly optimized for general-purpose storage management, and user-defined replacements are unlikely to do better. (However, it's often reasonable to employ member memory-management operations to customize memory management for a specific class or hierarchy.)

Special-purpose versions of `operator new` and `operator delete` that implement different behavior from the standard versions will probably introduce bugs, since the correctness of much of the standard library and many third-party libraries depends on the default standard implementations of these functions.

A safer approach is to overload the global `operator new` rather than replace it. Suppose we'd like to fill newly allocated storage with a particular character pattern:

```
void *operator new( size_t n, const string &pat ) {
   char *p = static_cast<char *>(::operator new( n ));
   const char *pattern = pat.c_str();
   if( !pattern || !pattern[0] )
       pattern = "\0"; // note: two null chars
   const char *f = pattern;
   for( int i = 0; i < n; ++i ) {
       if( !*f )
           f = pattern;
       p[i] = *f++;
   }
  return p;
}
```

This version of `operator new` accepts a `string` pattern argument that is copied into the newly allocated storage. The compiler distinguishes between the standard `operator new` and our two-argument version through overload resolution.

```
string fill( "<garbage>" );
string *string1 = new string( "Hello" ); // standard version
string *string2 =
   new (fill) string( "World!" ); // overloaded version
```

The standard also defines an overloaded `operator new` that takes, in addition to the required `size_t` first argument, a second argument of type `void *`. The implementation simply returns the second argument. (The `throw()` syntax is an exception-specification indicating that this function will not propagate any exceptions. It may be safely ignored in the following discussion, and in general.)

```
void *operator new( size_t, void *p ) throw()
   { return p; }
```

This is the standard "placement new," used to construct an object at a specific location. (Unlike with the standard, single-argument `operator new`, however, attempting to replace placement new is illegal.) Essentially, we use it to trick the compiler into calling a constructor for us. For example, for an embedded application, we may want to construct a "status register" object at a particular hardware address:

```
class StatusRegister {
   // . . .
};
void *regAddr = reinterpret_cast<void *>(0XFE0000);
// . . .
// place register object at regAddr
StatusRegister *sr = new (regAddr) StatusRegister;
```

Naturally, objects created with placement new must be destroyed at some point. However, since no memory is actually allocated by placement new, it's important to ensure that no memory is deleted. Recall that the behavior of the `delete` operator is to first activate the destructor of the object being deleted before calling an `operator delete` function to reclaim the storage. In the case of an object "allocated" with placement new, we must resort to an explicit destructor call to avoid any attempt to reclaim memory:

```
sr->~StatusRegister(); // explicit dtor call, no operator delete
```

Placement new and explicit destruction are clearly useful features, but they're just as clearly dangerous if not used sparingly and with caution. (See Gotcha #47 for one example from the standard library.)

Note that while we can overload `operator delete`, these overloaded versions will never be invoked by a standard delete-expression:

```
void *operator new( size_t n, Buffer &buffer ); // overloaded new
void operator delete( void *p,
   Buffer &buffer ); // corresponding delete
// . . .
Thing *thing1 = new Thing; // use standard operator new
Buffer buf;
Thing *thing2 = new (buf) Thing; // use overloaded operator new
delete thing2; // incorrect, should have used overloaded delete
delete thing1; // correct, uses standard operator delete
```

Instead, as with an object created with placement new, we're forced to call the object's destructor explicitly, then explicitly deallocate the former object's storage with a direct call to the appropriate `operator delete` function:

```
thing2->~Thing(); // correct, destroy Thing
operator delete( thing2, buf ); // correct, use overloaded delete
```

In practice, storage allocated by an overloaded global `operator new` is often erroneously deallocated by the standard global `operator delete`. One way to avoid this error is to ensure that any storage allocated by an overloaded global `operator new` obtains that storage from the standard global `operator new`. This is what we've done with the first overloaded implementation above, and our first version works correctly with standard global `operator delete`:

```
string fill( "<garbage>" );
string *string2 = new (fill) string( "World!" );
// . . .
delete string2; // works
```

Overloaded versions of global `operator new` should, in general, either not allocate any storage or should be simple wrappers around the standard global `operator new`.

Often, the best approach is to avoid doing anything at all with global scope memory-management operator functions, but instead customize memory management on a class or hierarchy basis through the use of member operators new, delete, array new, and array delete.

We noted at the end of Gotcha #61 that an "appropriate" operator delete would be invoked by the runtime system in the event of an exception propagating out of an initialization in a new-expression:

```
Thing *tp = new Thing( arg );
```

If the allocation of `Thing` succeeds but the constructor for `Thing` throws an exception, the runtime system will invoke an appropriate `operator delete` to reclaim the uninitialized memory referred to by `tp`. In the case above, the appropriate `operator delete` would be either the global `operator delete(void *)` or a member `operator delete` with the same signature. However, a different `operator new` would imply a different `operator delete`:

```
Thing *tp = new (buf) Thing( arg );
```

In this case, the appropriate `operator delete` is the two-argument version corresponding to the overloaded `operator new` used for the allocation of `Thing`; `operator delete( void *, Buffer &)`, and this is the version the runtime system will invoke.

C++ permits much flexibility in defining the behavior of memory management, but this flexibility comes at the cost of complexity. The standard, global versions of `operator new` and `operator delete` are sufficient for most needs. Employ more complex approaches only if they are clearly necessary.

Gotcha #63: Confusing Scope and Activation of Member new and delete

Member `operator new` and `operator delete` are invoked when objects of the class declaring them are created and destroyed. The actual scope in which the allocation expression occurs is immaterial:

```
class String {
 public:
   void *operator new( size_t ); // member operator new
   void operator delete( void * ); // member operator delete
   void *operator new[]( size_t ); // member operator new[]
   void operator delete [] ( void * ); // member operator delete[]
   String( const char * = "" );
   // . . .
};
```

```
void f() {
   String *sp = new String( "Heap" ); // uses String::operator new
   int *ip = new int( 12 ); // uses ::operator new
   delete ip; // uses :: operator delete
   delete sp; // uses String::delete
}
```

Again: the scope of the allocation doesn't matter; it's the type being allocated that determines the function called:

```
String::String( const char *s )
   : s_( strcpy( new char[strlen(s)+1], s ) )
   {}
```

The array of characters is allocated in the scope of class `String`, but the allocation uses the global array new, not `String`'s array new; a `char` is not a `String`. Explicit qualification can help:

```
String::String( const char *s )
   : s_( strcpy( reinterpret_cast<char *>
       (String::operator new[](strlen(s)+1 )),s ) )
   {}
```

It would be nice if we could say something like `String::new char[strlen(s)+1]` to access `String`'s `operator new[]` through the `new` operator (parse that!), but that's illegal syntax. (Although we can use `::new` to access a global `operator new` and `operator new[]` and `::delete` to access a global `operator delete` or `operator delete[]`.)

Gotcha #64: Throwing String Literals

Many authors of C++ programming texts demonstrate exceptions by throwing character string literals:

```
throw "Stack underflow!";
```

They know this is a reprehensible practice, but they do it anyway, because it's a "pedagogic example." Unfortunately, these authors often neglect to mention to their readers that actually following the implicit advice to imitate the example will spell mayhem and doom.

Never throw exception objects that are string literals. The principle reason is that these exception objects should eventually be caught, and they're caught based on their type, not on their value:

```
try {
   // . . .
}
catch( const char *msg ) {
   string m( msg );
   if( m == "stack underflow" ) // . . .
   else if( m == "connection timeout" ) // . . .
   else if( m == "security violation" ) // . . .
   else throw;
}
```

The practical effect of throwing and catching string literals is that almost no information about the exception is encoded in the type of the exception object. This imprecision requires that a catch clause intercept every such exception and examine its value to see if it applies. Worse, the value comparison is also highly subject to imprecision, and it often breaks under maintenance when the capitalization or formatting of an "error message" is modified. In our example above, we'll never recognize that a stack underflow has occurred.

These comments also apply to exceptions of other predefined and standard types. Throwing integers, floating point numbers, `string`s, or (on a really bad day) `set`s of `vector`s of `float`s will give rise to similar problems. Simply stated, the problem with throwing exception objects of predefined types is that once we've caught one, we don't know what it represents, and therefore how to respond to it. The thrower of the exception is taunting us: "Something really, really bad happened. Guess what!" And we have no choice but to submit to a contrived guessing game at which we're likely to lose.

An exception type is an abstract data type that represents an exception. The guidelines for its design are no different from those for the design of any abstract data type: identify and name a concept, decide on an abstract set of operations for the concept, and implement it. During implementation, consider initialization, copying, and conversions. Simple. Use of a string literal to represent an exception makes about as much sense as using a string literal as a complex number. Theoretically it might work, but practically it's going to be tedious and buggy.

What abstract concept are we trying to represent when we throw an exception that represents a stack underflow? Oh. Right.

```
class StackUnderflow {};
```

Often, the type of an exception object communicates all the required information about an exception, and it's not uncommon for exception types to dispense with explicitly declared member functions. However, the ability to provide some descriptive text is often handy. Less commonly, other information about the exception may also be recorded in the exception object:

```
class StackUnderflow {
 public:
   StackUnderflow( const char *msg = "stack underflow" );
   virtual ~StackUnderflow();
   virtual const char *what() const;
   // . . .
};
```

If provided, the function that returns the descriptive text should be a virtual member function named `what`, with the above signature. This is for orthogonality with the standard exception types, all of which provide such a function. In fact, it's often a good idea to derive an exception type from one of the standard exception types:

```
class StackUnderflow : public std::runtime_error {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
       : std::runtime_error( msg ) {}
};
```

This allows the exception to be caught either as a `StackUnderflow`, as a more general `runtime_error`, or as a very general standard `exception` (`runtime_error`'s public base class). It's also often a good idea to provide a more general, but nonstandard, exception type. Typically, such a type would serve as a base class for all exception types that may be thrown from a particular module or library:

```
class ContainerFault {
 public:
   virtual ~ContainerFault();
   virtual const char *what() const = 0;
   // . . .
};
```

```
class StackUnderflow
   : public std::runtime_error, public ContainerFault {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
       : std::runtime_error( msg ) {}
   const char *what() const
       { return std::runtime_error::what(); }
};
```

Finally, it's also necessary to provide proper copy and destruction semantics for exception types. In particular, the throwing of an exception implies that it must be legal to copy construct objects of the exception type, since this is what the run-time exception mechanism does when an exception is thrown (see Gotcha #65), and the copied exception must be destroyed after it has been handled. Often, we can allow the compiler to write these operations for us (see Gotcha #49):

```
class StackUnderflow
   : public std::runtime_error, public ContainerFault {
 public:
   explicit StackUnderflow( const char *msg = "stack underflow" )
       : std::runtime_error( msg ) {}
   // StackUnderflow( const StackUnderflow & );
   // StackUnderflow &operator =( const StackUnderflow & );
   const char *what() const
       { return std::runtime_error::what(); }
};
```

Now, users of our stack type can choose to detect a stack underflow as a `StackUnderflow` (they know they're using our stack type and are keeping close watch), as a more general `ContainerFault` (they know they're using our container library and are on the qui vive for any container error), as a `runtime_error` (they know nothing about our container library but want to handle any sort of standard runtime error), or as an `exception` (they're prepared to handle any standard exception).

Gotcha #65: Improper Exception Mechanics

Issues of general exception-handling policy and architecture are still subject to debate. However, lower-level guidelines concerning how exceptions should be thrown and caught are both well understood and commonly violated.

When a throw-expression is executed, the runtime exception-handling mechanism copies the exception object to a temporary in a "safe" location. The location of the temporary is highly platform dependent, but the temporary is guaranteed to persist until the exception has been handled. This means that the temporary will be usable until the last catch clause that uses the temporary has completed, even if several different catch clauses are executed for that temporary exception object. This is an important property because, to put it bluntly, when you throw an exception, all hell breaks loose. That temporary is the calm in the eye of the exception-handling maelstrom.

This is why it's not a good idea to throw a pointer.

```
throw new StackUnderflow( "operator stack" );
```

The address of the `StackUnderflow` object on the heap is copied to a safe location, but the heap memory to which it refers is unprotected. This approach also leaves open the possibility that the pointer may refer to a location that's on the runtime stack:

```
StackUnderflow e( "arg stack" );
throw &e;
```

Here, the storage to which the pointer exception object (remember, the pointer is what's being thrown, not what it points to) is referring to storage that may not exist when the exception is caught. (By the way, when a string literal is thrown, the entire array of characters is copied to the temporary, not just the address of the first character. This information is of little practical use, because we should never throw string literals. See Gotcha #64.) Additionally, a pointer may be null. Who needs this additional complexity? Don't throw pointers, throw objects:

```
StackUnderflow e( "arg stack" );
throw e;
```

The exception object is immediately copied to a temporary by the exception-handling mechanism, so the declaration of `e` is really not necessary. Conventionally, we throw anonymous temporaries:

```
throw StackUnderflow( "arg stack" );
```

Use of an anonymous temporary clearly states that the `StackUnderflow` object is for use only as an exception object, since its lifetime is restricted to the throw-expression. While the explicitly declared variable `e` will also be destroyed when the throw-expression executes, it is in scope, and accessible, until the end of the block containing its declaration. Use of an anonymous temporary also helps to stem some of the more "creative" attempts to handle exceptions:

```
static StackUnderflow e( "arg stack" );
extern StackUnderflow *argstackerr;
argstackerr = &e;
throw e;
```

Here, our clever coder has decided to stash the address of the exception object for use later, probably in some upstream catch clause. Unfortunately, the `argstackerr` pointer doesn't refer to the exception object (which is a temporary in an undisclosed location) but to the now destroyed object used to initialize it. Exception-handling code is not the best location for the introduction of obscure bugs. Keep it simple.

What's the best way to catch an exception object? Not by value:

```
try {
   // . . .
}
catch( ContainerFault fault ) {
   // . . .
}
```

Consider what would happen if this catch clause successfully caught a thrown `StackUnderflow` object. Slice. Since a `StackUnderflow` is-a `ContainerFault`, we could initialize `fault` with the thrown exception object, but we'd slice off all the derived class's data and behavior. (See Gotcha #30.)

In this particular case, however, we won't have a slicing problem, because `ContainerFault` is, as is proper in a base class, abstract (see Gotcha #93). The catch clause is therefore illegal. It's not possible to catch an exception object, by value, as a `ContainerFault`.

Catching by value allows us to expose ourselves to even more obscure problems:

```
catch( StackUnderflow fault ) {
   // do partial recovery . . .
   fault.modifyState(); // my fault
   throw; // re-throw current exception
}
```

It's not uncommon for a catch clause to perform a partial recovery, record the state of the recovery in the exception object, and re-throw the exception object for additional processing. Unfortunately, that's not what's happening here. This catch clause has performed a partial recovery, recorded the state of the recovery in a local copy of the exception object, and re-thrown the (unchanged) exception object.

For simplicity, and to avoid all these difficulties, we always throw anonymous temporary objects, and we catch them by reference.

Be careful not to reintroduce value copy problems into a handler. This occurs most commonly when a new exception is thrown from a handler rather than a re-throw of the existing exception:

```
catch( ContainerFault &fault ) {
   // do partial recovery . . .
   if( condition )
       throw; // re-throw
   else {
       ContainerFault myFault( fault );
       myFault.modifyState(); // still my fault
       throw myFault; // new exception object
   }
}
```

In this case, the recorded changes will not be lost, but the original type of the exception will be. Suppose the original thrown exception was of type `Stack-Underflow`. When it's caught as a reference to `ContainerFault`, the dynamic type of the exception object is still `StackUnderflow`, so a re-thrown object has the opportunity to be caught subsequently by a `StackUnderflow` catch clause as well as a `ContainerFault` clause. However, the new exception object `myFault` is of type `ContainerFault` and cannot be caught by a `StackUnderflow` clause. It's generally better to re-throw an existing exception object rather than handle the original exception and throw a new one:

```
catch( ContainerFault &fault ) {
   // do partial recovery . . .
   if( !condition )
       fault.modifyState();
   throw;
}
```

Fortunately, the ContainerFault base class is abstract, so this particular manifestation of the error is not possible; in general, base classes should be abstract. Obviously, this advice doesn't apply if you must throw an entirely different type of exception:

```
catch( ContainerFault &fault ) {
   // do partial recovery . . .
   if( out_of_memory )
       throw bad_alloc(); // throw new exception
   fault.modifyState();
   throw; // re-throw
}
```

Another common problem concerns the ordering of the catch clauses. Because the catch clauses are tested in sequence (like the conditions of an if-elseif, rather than a switch-statement) the types should, in general, be ordered from most specific to most general. For exception types that admit to no ordering, decide on a logical ordering:

```
catch( ContainerFault &fault ) {
   // do partial recovery . . .
   fault.modifyState(); // not my fault
   throw;
}
catch( StackUnderflow &fault ) {
   // . . .
}
catch( exception & ) {
   // . . .
}
```

The handler-sequence above will never catch a StackUnderflow exception, because the more general ContainerFault exception occurs first in the sequence.

The mechanics of exception handling offer much opportunity for complexity, but it's not necessary to accept the offer. When throwing and catching exceptions, keep things simple.

Gotcha #66: Abusing Local Addresses

Don’t return a pointer or reference to a local variable. Most compilers will warn about this situation; take the warning seriously.

Disappearing Stack Frames

If the variable is an automatic, the storage to which it refers will disappear on return:

```
char *newLabel1() {
   static int labNo = 0;
   char buffer[16]; // see Gotcha #2
   sprintf( buffer, "label%d", labNo++ );
   return buffer;
}
```

This function has the annoying property of working on occasion. After return, the stack frame for the `newLabel1` function is popped off the execution stack, releasing its storage (including the storage for `buffer`) for use by a subsequent function call. However, if the value is copied before another function is called, the returned pointer, though invalid, may still be usable:

```
char *uniqueLab = newLabel1();
char mybuf[16], *pmybuf = mybuf;
while( *pmybuf++ = *uniqueLab++ );
```

This is not the kind of code a maintainer will put up with for very long. The maintainer might decide to allocate the buffer off the heap:

```
char *pmybuf = new char[16];
```

The maintainer might decide not to hand-code the buffer copy:

```
strcpy( pmybuf, uniqueLab );
```

The maintainer might decide to use a more abstract type than a character buffer:

```
std::string mybuf( uniqueLab );
```

Any of these modifications may cause the local storage referred to by `uniqueLab` to be modified.

Static Interference

If the variable is static, a later call to the same function will affect the results of earlier calls:

```
char *newLabel2() {
   static int labNo = 0;
   static char buffer[16];
   sprintf( buffer, "label%d", labNo++ );
   return buffer;
}
```

The storage for the buffer is available after the function returns, but any other use of the function can affect the result:

```
//case 1
cout << "first: " << newLabel2() << ' ';
cout << "second: " << newLabel2() << endl;

// case 2
cout << "first: " << newLabel2() << ' '
    << "second: " << newLabel2() << endl;
```

In the first case, we'll print different labels. In the second case, we'll probably (but not necessarily) print the same label twice. Presumably, someone who was intimately aware of the unusual implementation of the `newLabel2` function wrote case 1 to break up the label output into separate statements, to take that flawed implementation into account. A later maintainer is unlikely to be as familiar with the implementation vagaries of `newLabel2` and is likely to merge the separate output statements into one, causing a bug. Worse, the merged output statement could continue to exhibit the same behavior as the separate statements and change unpredictably in the future. (See Gotcha #14.)

Idiomatic Difficulties

Another danger is lurking as well. Keep in mind that users of a function generally do not have access to its implementation and therefore have to determine how to handle a function's return value from a reading of the function declaration. While a comment may provide this information (see Gotcha #1), it's also important that the function be designed to encourage proper use.

Avoid returning a reference that refers to memory allocated within the function. Users of the function will invariably neglect to delete the storage, causing memory leaks:

```
int &f()
   { return *new int( 5 ); }
// . . .
int i = f(); // memory leak!
```

The correct code has to convert the reference to an address or copy the result and free the memory. Not on my shift, buddy:

```
int *ip = &f(); // one horrible way
int &tmp = f(); // another
int i = tmp;
delete &tmp;
```

This is a particularly bad idea for overloaded operator functions:

```
Complex &operator +( const Complex &a, const Complex &b )
   { return *new Complex( a.re+b.re, a.im+b.im ); }
// . . .
Complex a, b, c;
a = b + c + a + b; // lots of leaks!
```

Return a pointer to the storage instead, or don't allocate storage and return by value:

```
int *f() { return new int(5); }
Complex operator +( Complex a, Complex b )
   { return Complex( a.re+b.re, a.im+b.im ); }
```

Idiomatically, users of a function that returns a pointer expect that they might be responsible for the eventual deletion of the storage referred to by the pointer and will make some effort to determine whether this is actually the case (say, by reading a comment). Users of a function that returns a reference rarely do.

Local Scope Problems

The problems we encounter with lifetimes of local variables can occur not only on the boundaries between functions but also within the nested scopes of an individual function:

```
void localScope( int x ) {
   char *cp = 0;
   if( x ) {
       char buf1[] = "asdf";
       cp = buf1; // bad idea!
       char buf2[] = "qwerty";
       char *cp1 = buf2;
       // . . .
   }
   if( x-1 ) {
       char *cp2 = 0; // overlays buf1?
       // . . .
   }
   if( cp )
       printf( cp ); // error, maybe . . .
}
```

Compilers have a lot of flexibility in how they lay out the storage for local variables. Depending on the platform and compiler options, the compiler may overlay the storage for `buf1` and `cp2`. This is legal, because `buf1` and `cp2` have disjoint scope and lifetime. If the overlay does occur, `buf1` will be corrupted, and the behavior of the `printf` may be affected (it probably just won't print anything). For the sake of portability, it's best not to depend on a particular stack frame layout.

The Static Fix

When faced with a difficult bug, sometimes the problem "goes away" with an application of the `static` storage class specifier:

```
// . . .
char buf[MAX];
long count = 0;
// . . .
```

```
int i = 0;
while( i++ <= MAX )
   if( buf[i] == '\0' ) {
       buf[i] = '*';
       ++count;
   }
assert( count <= i );
// . . .
```

This code has a poorly written loop that will sometimes write past the end of the `buf` array into `count`, causing the assertion to fail. In the wild thrashing that sometimes accompanies attempts to bug fix, the programmer may declare `count` to be a local static, and the code will then work:

```
char buf[MAX];
static long count;
// . . .
count = 0;
int i = 0;
while( i++ <= MAX )
   if( buf[i] == '\0' ) {
       buf[i] = '*';
       ++count;
   }
assert( count <= i );
```

Many programmers, not willing to question their good luck in fixing the problem so easily, will leave it at that. Unfortunately, the problem has not gone away; it has just been moved somewhere else. It's lying in wait, ready to strike at a future time.

Making the local variable `count` static has the effect of moving its storage out of the stack frame of the function and into an entirely different region of memory, where static objects are located. Because it has moved, it will no longer be overwritten. However, not only is `count` now subject to the problems mentioned under "Static Interference" above; it's also likely that another local variable—or a future local variable—is being overwritten. The proper solution is, as usual, to fix the bug rather than hide it:

```
char buf[MAX];
long count = 0;
// . . .
```

```
for( int i = 1; i < MAX; ++i )
   if( buf[i] == '\0' ) {
       buf[i] = '*';
       ++count;
   }
// . . .
```

Gotcha #67: Failure to Employ Resource Acquisition Is Initialization

It's a shame that many newer C++ programmers don't appreciate the wonderful symmetry of constructors and destructors. For the most part, these are programmers who were reared on languages that tried to keep them safe from the vagaries of pointers and memory management. Safe and controlled. Ignorant and happy. Programming precisely the way the designer of the language has decreed that one should program. The one, true way. Their way.

Happily, C++ has more respect for its practitioners and provides much flexibility as to how the language may be applied. This is not to say we don't have general principles and guiding idioms (see Gotcha #10). One of the most important of these idioms is the "resource acquisition is initialization" idiom. That's quite a mouthful, but it's a simple and extensible technique for binding resources to memory and managing both efficiently and predictably.

The order of execution of construction and destruction are mirror images of each other. When a class is constructed, the order of initialization is always the same: the virtual base class subobjects first ("in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes," according to the standard), followed by the immediate base classes in the order of their appearance on the base-list in the class's definition, followed by the non-static data members of the class, in the order of their declaration, followed by the body of the constructor. The destructor implements the reverse order: destructor body, members in the reverse order of their declarations, immediate base classes in the inverse order of their appearance, and virtual base classes. It's helpful to think of construction as pushing a sequence onto a stack and destruction as popping the stack to implement the reverse sequence. The symmetry of construction and destruction is considered so important that all of a class's constructors perform their initializations in the same sequence, even if their member initialization lists are written in different orders (see Gotcha #52).

As a side effect or result of initialization, a constructor gathers resources for the object's use as the object is constructed. Often, the order in which these resources are seized is essential (for example, you have to lock the database before you write it; you have to get a file handle before you write to the file), and typically, the destructor has the job of releasing these resources in the inverse order in which they were seized. That there may be many constructors but only a single destructor implies that all constructors must execute their component initializations in the same sequence.

(This wasn't always the case, by the way. In the very early days of the language, the order of initializations in constructors was not fixed, which caused much difficulty for projects of any level of complexity. Like most language rules in C++, this one is the result of thoughtful design coupled with production experience.)

This symmetry of construction and destruction persists even as we move from the object structure itself to the uses of multiple objects. Consider a simple trace class:

➤➤ gotcha67/trace.h

```
class Trace {
 public:
   Trace( const char *msg )
       : m_( msg ) { cout << "Entering " << m_ << endl; }
   ~Trace()
       { cout << "Exiting " << m_ << endl; }
 private:
   const char *m_;
};
```

This trace class is perhaps a little too simple, in that it makes the assumption that its initializer is valid and will have a lifetime at least as long as the `Trace` object, but it's adequate for our purposes. A `Trace` object prints out a message when it's created and again when it's destroyed, so it can be used to trace flow of execution:

➤➤ gotcha67/trace.cpp

```
Trace a( "global" );
void loopy( int cond1, int cond2 ) {
   Trace b( "function body" );
it: Trace c( "later in body" );
   if( cond1 == cond2 )
       return;
```

```
        if( cond1-1 ) {
            Trace d( "if" );
            static Trace stat( "local static" );
            while( --cond1 ) {
                Trace e( "loop" );
                if( cond1 == cond2 )
                    goto it;
            }
            Trace f( "after loop" );
        }
        Trace g( "after if" );
    }
```

Calling the function `loopy` with the arguments `4` and `2` produces the following:

```
Entering global
Entering function body
Entering later in body
Entering if
Entering local static
Entering loop
Exiting loop
Entering loop
Exiting loop
Exiting if
Exiting later in body
Entering later in body
Exiting later in body
Exiting function body
Exiting local static
Exiting global
```

The messages show clearly how the lifetime of a `Trace` object is associated with the current scope of execution. In particular, note the effect the `goto` and `return` have on the lifetimes of the active `Trace` objects. Neither of these branches is exemplary coding practice, but they're the kinds of constructs that tend to appear as code is maintained.

```
void doDB() {
   lockDB();
   // do stuff with database . . .
   unlockDB();
}
```

In the code above, we've been careful to lock the database before access and unlock it when we've finished accessing it. Unfortunately, this is the kind of careful code that breaks under maintenance, particularly if the section of code between the lock and unlock is lengthy:

```
void doDB() {
   lockDB();
   // . . .
   if( i_feel_like_it )
       return;
   // . . .
   unlockDB();
}
```

Now we have a bug whenever the `doDB` function feels like it; the database will remain locked, and this will no doubt cause much difficulty elsewhere. Actually, even the original code was not properly written, because an exception might have been thrown after the database was locked but before it was unlocked. This would have the same effect as any branch past the call to `unlockDB`: the database would remain locked.

We could try to fix the problem by taking exceptions explicitly into account and by giving stern lectures to maintainers:

```
void doDB() {
   lockDB();
   try {
       // do stuff with database . . .
   }
   catch(  . . .  ) {
       unlockDB();
       throw;
   }
   unlockDB();
}
```

This approach is wordy, low-tech, slow, hard to maintain, and will cause you to be mistaken for a member of the Department of Redundancy Department. Properly written, exception-safe code usually employs few try blocks. Instead, it uses resource acquisition is initialization:

```
class DBLock {
 public:
   DBLock() { lockDB(); }
   ~DBLock() { unlockDB(); }
};

void doDB() {
   DBLock lock;
   // do stuff with database . . .
}
```

The creation of a `DBLock` object causes the database lock resource to be seized. When the `DBLock` object goes out of scope for whatever reason, the destructor will reclaim the resource and unlock the database. This idiom is so commonly used in C++, it often passes unnoticed. But any time you use a standard `string`, `vector`, `list`, or a host of other types, you're employing resource acquisition is initialization.

By the way, be wary of two common problems often associated with the use of resource handle classes like `DBLock`:

```
void doDB() {
   DBLock lock1; // correct
   DBLock lock2(); // oops!
   DBLock(); // oops!
   // do stuff with database . . .
}
```

The declaration of `lock1` is correct; it's a `DBLock` object that comes into scope just before the terminating semicolon of the declaration and goes out of scope at the end of the block that contains its declaration (in this case, at the end of the function). The declaration of `lock2` declares it to be a function that takes no argument and returns a `DBLock` (see Gotcha #19). It's not an error, but it's probably not what was intended, since no locking or unlocking will be performed.

The following line is an expression-statement that creates an anonymous temporary `DBLock` object. This will indeed lock the database, but because the anonymous temporary goes out of scope at the end of the expression (just before the semicolon), the database will be immediately unlocked. Probably not what you want.

The standard `auto_ptr` template is a useful general-purpose resource handle for objects allocated on the heap. See Gotchas #10 and #68.

Gotcha #68: Improper Use of `auto_ptr`

The standard `auto_ptr` template is a simple and useful resource handle with unusual copy semantics (see Gotcha #10). Most uses of `auto_ptr` are straightforward:

```
template <typename T>
void print( Container<T> &c ) {
   auto_ptr< Iter<T> > i( c.genIter() );
   for( i->reset(); !i->done(); i->next() ) {
       cout << i->get() << endl;
       examine( c );
   }
   // implicit cleanup . . .
}
```

This is a common use of `auto_ptr` to ensure that the storage and resources of a heap-allocated object are freed when the pointer that refers to it goes out of scope. (See Gotcha #90 for a more complete rendering of the `Container` hierarchy.) The assumption above is that the memory for the `Iter<T>` returned by `genIter` has been allocated from the heap. The `auto_ptr< Iter<T> >` will therefore invoke the `delete` operator to reclaim the object when the `auto_ptr` goes out of scope.

However, there are two common errors in the use of `auto_ptr`. The first is the assumption that an `auto_ptr` can refer to an array.

```
void calc( double src[], int len ) {
   double *tmp = new double[len];
   // . . .
   delete [] tmp;
}
```

The calc function is fragile, in that the allocated tmp array will not be recovered in the event that an exception occurs during execution of the function or if improper maintenance causes an early exit from the function. A resource handle is what's required, and auto_ptr is our standard resource handle:

```
void calc( double src[], int len ) {
   auto_ptr<double> tmp( new double[len] );
   // . . .
}
```

However, an auto_ptr is a standard resource handle to a single object, not to an array of objects. When tmp goes out of scope and its destructor is activated, a scalar deletion will be performed on the array of doubles that was allocated with an array new (see Gotcha #60), because, unfortunately, the compiler can't tell the difference between a pointer to an array and a pointer to a single object. Even more unfortunately, this code may occasionally work on some platforms, and the problem may be detected only when porting to a new platform or when upgrading to a new version of an existing platform.

A better solution is to use a standard vector to hold the array of doubles. A standard vector is essentially a resource handle for an array, a kind of "auto_array," but with many additional facilities. At the same time, it's probably a good idea to get rid of the primitive and dangerous use of a pointer formal argument masquerading as an array:

```
void calc( vector<double> &src ) {
   vector<double> tmp( src.size() );
   // . . .
}
```

The other common error is to use an auto_ptr as the element type of an STL container. STL containers don't make many demands on their elements, but they do require conventional copy semantics.

In fact, the standard defines auto_ptr in such a way that it's illegal to instantiate an STL container with an auto_ptr element type; such usage should produce a compile-time error (and probably a cryptic one, at that). However, many current implementations lag behind the standard.

In one common outdated implementation of auto_ptr, its copy semantics are actually suitable for use as the element type of a container, and they can be used successfully. That is, until you get a different or newer version of the standard library, at which time your code will fail to compile. Very annoying, but usually a straightforward fix.

A worse situation occurs when the implementation of `auto_ptr` is not fully standard, so that it's possible to use it to instantiate an STL container, but the copy semantics are not what is required by the STL. As described in Gotcha #10, copying an `auto_ptr` transfers control of the pointed-to object and sets the source of the copy to null:

```
auto_ptr<Employee> e1( new Hourly );
auto_ptr<Employee> e2( e1 );  // e1 is null
e1 = e2; // e2 is null
```

This property is quite useful in many contexts but isn't what is required of an STL container element:

```
vector< auto_ptr<Employee> > payroll;
// . . .
list< auto_ptr<Employee> > temp;
copy( payroll.begin(), payroll.end(), back_inserter(temp) );
```

On some platforms this code may compile and run, but it probably won't do what it should. The `vector` of `Employee` pointers will be copied into the `list`, but after the copy is complete, the `vector` will contain all null pointers!

Avoid the use of `auto_ptr` as an STL container element, even if your current platform allows you to get away with it.