## Item 1:  Prefer const and inline to #define

1. Better be called "prefer the compiler to the preprocessor,"

When you do something like this,

#define ASPECT\_RATIO 1.653

The symbolic name ASPECT\_RATIO may never be seen by compilers; it may be removed by the preprocessor before the source code ever gets to a compiler

As a result, the name ASPECT\_RATIO may not get entered into the symbol table

This problem can also crop up in a symbolic debugger, because, again, the name you're programming with may not be in the symbol table.

1. The solution to this sorry scenario is simple and succinct. Instead of using a preprocessor macro, define a constant:

const double ASPECT\_RATIO = 1.653;

1. Notation about above:
   1. First, things can get a bit tricky when defining constant pointers

It's important that the *pointer* be declared const

const char \* const authorName = "Scott Meyers";

* 1. Second, it's often convenient to define class-specific constants

1. Another common (mis)use of the #define directive is using it to implement macros that look like functions but that don't incur the overhead of a function call. The canonical example is computing the maximum of two values:

#define max(a,b) ((a) > (b) ? (a) : (b))

* 1. remember to parenthesize all the arguments
  2. But even if you get that right, look at the weird things that can happen:

int a = 5, b = 0;

max(++a, b); // a is incremented twice

max(++a, b+10); // a is incremented once

1. It can be done with a template, inline function

template<class T>

inline const T& max(const T& a, const T& b)

{ return a > b ? a : b; }

## Item 2:  Prefer <iostream> to <stdio.h>

pro/cons for stdio.h functions

1. Portable.
2. Efficient.
3. You already know how to use them
4. Not type-safe and they're not extensible. Because type safety and extensibility are cornerstones of the C++ way of life, you might just as well resign yourself to them right now.
5. The printf/scanf family of functions separate the variables to be read or written from the formatting information that controls the reads and writes, just like FORTRAN does.

These weaknesses of printf/scanf are the strengths of operator>> and <<.

int i;

Rational r; // r is a rational number

...

cin >> i >> r;

cout << i << r;

Compilers take care of figuring out which versions of the operators to call for different variables, so you needn't worry about specifying that the first object to be read or written is an int and the second is a Rational.

In addition, objects to be read are passed using the same syntactic form as are those to be written, so you don't have to remember silly rules like you do for scanf, where if you don't already have a pointer, you have to be sure to take an address, but if you've already got a pointer, you have to be sure *not* to take an address. Let C++ compilers take care of those details. They have nothing better to do, and you *do* have better things to do.

Finally, note that built-in types like int are read and written in the same manner as user-defined types like Rational. Try *that* using scanf and printf!

Here's how you might write an output routine for a class representing rational numbers:

class Rational {

public:

Rational(int numerator = 0, int denominator = 1);

private:

int n, d; // numerator and denominator

friend ostream& operator<<(ostream& s, const Rational& r);

};

ostream& operator<<(ostream& s, const Rational& r){

s << r.n << '/' << r.d;

return s;

}

This version of operator<< demonstrates some subtle (but important) points that are discussed elsewhere in this book.

1. operator<< is not a member function ([Item 19](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI19_FR.HTM#5887) explains why)
2. The Rational object to be output is passed into operator<< as a reference-to-const rather than as an object (see [Item 22](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI22_FR.HTM#6133)).
3. The corresponding input function, operator>>, would be declared and implemented in a similar manner.

Some situations in which it may make sense to fall back on the tried and true.

1. Some implementations of iostream operations are less efficient than the corresponding C stream operations, so it's possible (though unlikely — see [Item M16](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI16_FR.HTM#40995)) that you have an application in which this makes a significant difference. Bear in mind, though, that this says nothing about iostreams *in general*, only about particular implementations; see [Item M23](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI23_FR.HTM#41253).
2. The iostream library was modified in some rather fundamental ways during the course of its standardization (see [Item 49](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI49_FR.HTM#8392)), so applications that must be maximally portable may discover that different vendors support different approximations to the standard.
3. Finally, because the classes of the iostream library have constructors and the functions in <stdio.h> do not, there are rare occasions involving the initialization order of static objects (see [Item 47](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI47_FR.HTM#8299)) when the standard C library may be more useful simply because you know that you can always call it with impunity.[Item E2, P8](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI2_DIR.HTM#dingp8)

The type safety and extensibility offered by the classes and functions in the iostream library are more useful than you might initially imagine, so don't throw them away just because you're used to <stdio.h>. After all, even after the transition, you'll still have your memories.[Item E2, P9](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI2_DIR.HTM#dingp9)

<iostream> vs <iostream.h>.

Technically speaking, there is no such thing as <iostream.h> — the **°**[standardization committee](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=committee) eliminated it in favor of <iostream> when they truncated the names of the other non-C standard header names. The reasons for their doing this are explained in [Item 49](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI49_FR.HTM#8392), but what you really need to understand is that if (as is likely) your compilers support both <iostream> and <iostream.h>, the headers are subtly different.

In particular, if you #include <iostream>, you get the elements of the iostream library ensconced within the namespace std (see [Item 28](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI28_FR.HTM#6429)), but if you #include <iostream.h>, you get those same elements at global scope. Getting them at global scope can lead to name conflicts, precisely the kinds of name conflicts the use of namespaces is designed to prevent. Besides, <iostream> is less to type than <iostream.h>. For many people, that's reason enough to prefer it. [¤](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI2_DIR.HTM#dingp10) [Item E2](file:///J:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI2_DIR.HTM#dingp10)

## Item 3:   Prefer new and delete to malloc and free. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp1) [Item E3, P1](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp1)

The problem with malloc and free (and their variants) is simple: they don't know about **constructors and destructors**. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp2) [Item E3, P2](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp2)

Consider the following two ways to get space for an array of 10 string objects, one using malloc, the other using new: [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp3) [Item E3, P3](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp3)

string \*stringArray1 =

static\_cast<string\*>(malloc(10 \* sizeof(string)));

string \*stringArray2 = new string[10];

Here stringArray1 points to enough memory for 10 string objects, but no objects have been constructed in that memory. Furthermore, without jumping through some rather obscure linguistic hoops (such as those described in Items [M4](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI4_FR.HTM#5218) and [M8](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI8_FR.HTM#33985)), you have no way to initialize the objects in the array. In other words, stringArray1 is pretty useless. In contrast, stringArray2 points to an array of 10 fully constructed string objects, each of which can safely be used in any operation taking a string. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp4) [Item E3, P4](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp4)

Nonetheless, let's suppose you magically managed to initialize the objects in the stringArray1 array. Later on in your program, then, you'd expect to do this: [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp5) [Item E3, P5](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp5)

free(stringArray1);

delete [] stringArray2; // see [Item 5](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI5_FR.HTM#1869) for why the

// "[]" is necessary

The call to free will release the memory pointed to by stringArray1, but no destructors will be called on the string objects in that memory. If the string objects themselves allocated memory, as string objects are wont to do, all the memory they allocated will be lost. On the other hand, when delete is called on stringArray2, a destructor is called for each object in the array before any memory is released. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp6) [Item E3, P6](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp6)

Because new and delete interact properly with constructors and destructors, they are clearly the superior choice. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp7) [Item E3, P7](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp7)

Mixing new and delete with malloc and free is usually a bad idea. When you try to call free on a pointer you got from new or call delete on a pointer you got from malloc, the results are undefined, and we all know what "undefined" means: it means it works during development, it works during testing, and it blows up in your most important customers' faces. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp8) [Item E3, P8](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp8)

The incompatibility of new/delete and malloc/free can lead to some interesting complications. For example, the strdup function commonly found in <string.h> takes a char\*-based string and returns a copy of it: [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp9) [Item E3, P9](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp9)

char \* strdup(const char \*ps); // return a copy of what

// ps points to

At some sites, both C and C++ use the same version of strdup, so the memory allocated inside the function comes from malloc. As a result, unwitting C++ programmers calling strdup might overlook the fact that they must use free on the pointer returned from strdup. But wait! To forestall such complications, some sites might decide to rewrite strdup for C++ and have this rewritten version call new inside the function, thereby mandating that callers later use delete. As you can imagine, this can lead to some pretty nightmarish portability problems as code is shuttled back and forth between sites with different forms of strdup. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp10) [Item E3, P10](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp10)

Still, C++ programmers are as interested in code reuse as C programmers, and it's a simple fact that there are lots of C libraries based on malloc and free containing code that is very much worth reusing. When taking advantage of such a library, it's likely you'll end up with the responsibility for freeing memory malloced by the library and/or mallocing memory the library itself will free. That's fine. There's nothing wrong with calling malloc and free inside a C++ program as long as you make sure the pointers you get from malloc always meet their maker in free and the pointers you get from new eventually find their way to delete. The problems start when you get sloppy and try to mix new with free or malloc with delete. That's just asking for trouble. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp11) [Item E3, P11](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp11)

Given that malloc and free are ignorant of constructors and destructors and that mixing malloc/free with new/delete can be more volatile than a fraternity rush party, you're best off sticking to an exclusive diet of news and deletes whenever you can. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp12) [Item E3, P](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI3_DIR.HTM#dingp12)

## Memory Management [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EMEM_DIR.HTM#dingp1) [EC++ Mem Mgmt, P1](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EMEM_DIR.HTM#dingp1)

1. Two meanings: Get things right – calling memory allocation and dellocation routine correctly; Make it efficiently – writing custom versions of the allocation and deallocation routines.
2. What is memory leak?

Memory management concerns in C++ fall into two general camps: getting it right and making it perform efficiently. Good programmers understand that these concerns should be addressed in that order, because a program that is dazzlingly fast and astoundingly small is of little use if it doesn't behave the way it's supposed to. For most programmers, getting things right means calling memory allocation and deallocation routines correctly. Making things perform efficiently, on the other hand, often means writing custom versions of the allocation and deallocation routines. Getting things right there is even more important. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EMEM_DIR.HTM#dingp2) [EC++ Mem Mgmt, P2](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EMEM_DIR.HTM#dingp2)

On the correctness front, C++ inherits from C one of its biggest headaches, that of potential memory leaks. Even virtual memory, wonderful invention though it is, is finite, and not everybody has virtual memory in the first place. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EMEM_DIR.HTM#dingp3) [EC++ Mem Mgmt, P3](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EMEM_DIR.HTM#dingp3)

In C, a **memory leak** arises whenever memory allocated through malloc is never returned through free. The names of the players in C++ are new and delete, but the story is much the same. However, the situation is improved somewhat by the presence of destructors, because they provide a convenient repository for calls to delete that all objects must make when they are destroyed. At the same time, there is more to worry about, because new implicitly calls constructors and delete implicitly calls destructors. Furthermore, there is the complication that you can define your own versions of operator new and operator delete, both inside and outside of classes. This gives rise to all kinds of opportunities to make mistakes. The following Items (as well as [Item M8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI8_FR.HTM#33985)) should help you avoid some of the most common ones. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EMEM_DIR.HTM#dingp4) [EC++ Mem Mgmt, P4](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EMEM_DIR.HTM#dingp4)

## Item 5:  Use the same form in corresponding uses of new and delete. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp1) [Item E5, P1](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp1)

1. What happens when calling new and delete?
2. Make sure to use the same form or *new* and *delete*.
3. Also pay attention to *typedef*

What's wrong with this picture? [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp2) [Item E5, P2](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp2)

string \*stringArray = new string[100];

...

delete stringArray;

Everything here appears to be in order — the use of new is matched with a use of delete — but something is still quite wrong: your program's behavior is undefined. At the very least, 99 of the 100 string objects pointed to by stringArray are unlikely to be properly destroyed, because their destructors will probably never be called. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp3) [Item E5, P3](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp3)

When you use new, two things happen. First, memory is allocated (via the function operator new, about which I'll have more to say in Items [7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_FR.HTM#1894)-[10](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI10_FR.HTM#1986) as well as [Item M8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI8_FR.HTM#33985)). Second, one or more constructors are called for that memory. When you use delete, two other things happen: one or more destructors are called for the memory, then the memory is deallocated (via the function operator delete — see Items [8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_FR.HTM#120851) and [M8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI8_FR.HTM#33985)). The big question for delete is this: *how many* objects reside in the memory being deleted? The answer to that determines how many destructors must be called. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp4) [Item E5, P4](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp4)

Actually, the question is simpler: does the pointer being deleted point to a single object or to an array of objects? The only way for delete to know is for you to tell it. If you don't use brackets in your use of delete, delete assumes a single object is pointed to. Otherwise, it assumes that an array is pointed to: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp5) [Item E5, P5](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp5)

string \*stringPtr1 = new string;

string \*stringPtr2 = new string[100];

...

delete stringPtr1; // delete an object

delete [] stringPtr2; // delete an array of

// objects

What would happen if you used the "[]" form on stringPtr1? The result is undefined. What would happen if you didn't use the "[]" form on stringPtr2? Well, that's undefined too. Furthermore, it's undefined even for built-in types like ints, even though such types lack destructors. The rule, then, is simple: if you use [] when you call new, you must use [] when you call delete. If you don't use [] when you call new, don't use [] when you call delete. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp6) [Item E5, P6](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp6)

This is a particularly important rule to bear in mind when you are writing a class containing a pointer data member and also offering multiple constructors, because then you've got to be careful to use the *same form* of new in all the constructors to initialize the pointer member. If you don't, how will you know what form of delete to use in your destructor? For a further examination of this issue, see [Item 11](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI11_FR.HTM#2042). [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp7) [Item E5, P7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp7)

**This rule is also important for the typedef-inclined**, because it means that a typedef's author must document which form of delete should be employed when new is used to conjure up objects of the typedef type. For example, consider this typedef: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp8) [Item E5, P8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp8)

typedef string AddressLines[4]; // a person's address

// has 4 lines, each of

// which is a string

Because AddressLines is an array, this use of new, [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp9) [Item E5, P9](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp9)

string \*pal = new AddressLines; // note that "new

// AddressLines" returns

// a string\*, just like

// "new string[4]" would

must be matched with the *array* form of delete: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp10) [Item E5, P10](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp10)

delete pal; // undefined!

delete [] pal; // fine

To avoid such confusion, you're probably best off abstaining from typedefs for array types. That should be easy, however, because the standard C++ library (see [Item 49](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI49_FR.HTM#8392)) includes string and vector templates that reduce the need for built-in arrays to nearly zero. Here, for example, AddressLines could be defined to be a vector of strings. That is, AddressLines could be of type vector<string>. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp11) [It](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp11)

## Item 6:  Use delete on pointer members in destructors. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp1) [Item E6, P1](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp1)

1. Sync up the status of pointer members
   1. Constructor: allocate memory or assign it to 0;
   2. Destructor: deallocate memor
   3. Assignment operator: delete existing, and assign new memory
2. Deleting a null pointer is always safe

Most of the time, classes performing dynamic memory allocation will use new in the constructor(s) to allocate the memory and will later use delete in the destructor to free up the memory. This isn't too difficult to get right when you first write the class, provided, of course, that you remember to employ delete on *all* the members that could have been assigned memory in *any* constructor. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp2) [Item E6, P2](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp2)

However, the situation becomes more difficult as classes are maintained and enhanced, because the programmers making the modifications to the class may not be the ones who wrote the class in the first place. Under those conditions, it's easy to forget that adding a pointer member almost always requires each of the following: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp3) [Item E6, P3](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp3)

* Initialization of the pointer in each of the constructors. If no memory is to be allocated to the pointer in a particular constructor, the pointer should be initialized to 0 (i.e., the null pointer). [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp4) [Item E6, P4](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp4)
* Deletion of the existing memory and assignment of new memory in the assignment operator. (See also [Item 17](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI17_FR.HTM#2264).) [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp5) [Item E6, P5](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp5)
* Deletion of the pointer in the destructor. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp6) [Item E6, P6](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp6)

If you forget to initialize a pointer in a constructor, or if you forget to handle it inside the assignment operator, the problem usually becomes apparent fairly quickly, so in practice those issues don't tend to plague you. Failing to delete the pointer in the destructor, however, often exhibits no obvious external symptoms. Instead, it manifests itself as a subtle memory leak, a slowly growing cancer that will eventually devour your address space and drive your program to an early demise. Because this particular problem doesn't usually call attention to itself, it's important that you keep it in mind whenever you add a pointer member to a class. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp7) [Item E6, P7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp7)

Note, by the way, that deleting a null pointer is always safe (it does nothing). Thus, if you write your constructors, your assignment operators, and your other member functions such that each pointer member of the class is always either pointing to valid memory or is null, you can merrily delete away in the destructor without regard for whether you ever used new for the pointer in question. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp8) [Item E6, P8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp8)

There's no reason to get fascist about this Item. For example, you certainly don't want to use delete on a pointer that wasn't initialized via new, and, except in the case of smart pointer objects (see [Item M28](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI28_FR.HTM#61766)), you almost *never* want to delete a pointer that was passed to you in the first place. In other words, your class destructor usually shouldn't be using delete unless your class members were the ones who used new in the first place. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp9) [Item E6, P9](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp9)

Speaking of smart pointers, one way to avoid the need to delete pointer members is to replace those members with smart pointer objects like the standard C++ Library's auto\_ptr. To see how this can work, take a look at Items [M9](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI9_FR.HTM#5292) and [M10](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI10_FR.HTM#38223). [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp10) [Item E6, P10](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp10)

## Item 7:  Be prepared for out-of-memory conditions. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp1) [Item E7, P1](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp1)

1. When operator new can't allocate the memory you request, it throws bad\_alloc or returns 0
2. There are three common syntactic forms for getting new objects of type T
3. **when operator new cannot satisfy a request, it calls a client-specifiable error-handling function — often called a *new-handler* — before it throws an exception**

When operator new can't allocate the memory you request, it throws an exception. (It used to return 0, and some older compilers still do that. You can make your compilers do it again if you want to, but I'll defer that discussion until the end of this Item.) Deep in your heart of hearts, you know that handling out-of-memory exceptions is the only truly moral course of action. At the same time, you are keenly aware of the fact that doing so is a pain in the neck. As a result, chances are that you omit such handling from time to time. Like always, perhaps. Still, you must harbor a lurking sense of guilt. I mean, what if new really *does* yield an exception? [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp2) [Item E7, P2](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp2)

You may think that one reasonable way to cope with this matter is to fall back on your days in the gutter, i.e., to use the preprocessor. For example, a common C idiom is to define a type-independent macro to allocate memory and then check to make sure the allocation succeeded. For C++, such a macro might look something like this: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp3) [Item E7, P3](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp3)

#define NEW(PTR, TYPE) \

try { (PTR) = new TYPE; } \

catch (std::bad\_alloc&) { assert(0); }

("Wait! What's this std::bad\_alloc business?", you ask. bad\_alloc is the type of exception operator new throws when it can't satisfy a memory allocation request, and std is the name of the namespace (see [Item 28](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI28_FR.HTM#6429)) where bad\_alloc is defined. "Okay," you continue, "what's this assert business?" Well, if you look in the standard C include file <assert.h> (or its namespace-savvy C++ equivalent, <cassert> — see [Item 49](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI49_FR.HTM#8392)), you'll find that assert is a macro. The macro checks to see if the expression it's passed is non-zero, and, if it's not, it issues an error message and calls abort. Okay, it does that only when the standard macro NDEBUG isn't defined, i.e., in debug mode. In production mode, i.e., when NDEBUG is defined, assert expands to nothing — to a void statement. You thus check assertions only when debugging.) [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp4) [Item E7, P4](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp4)

This NEW macro suffers from the common error of using an assert to test a condition that might occur in production code (after all, you can run out of memory at any time), but it also has a drawback specific to C++: it fails to take into account the myriad ways in which new can be used. There are three common syntactic forms for getting new objects of type T, and you need to deal with the possibility of exceptions for each of these forms: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp5) [Item E7, P5](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp5)

new T;

new T(*constructor arguments*);

new T[*size*];

This oversimplifies the problem, however, because clients can define their own (overloaded) versions of operator new, so programs may contain an arbitrary number of different syntactic forms for using new. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp6) [Item E7, P6](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp6)

How, then, to cope? If you're willing to settle for a very simple error-handling strategy, you can set things up so that if a request for memory cannot be satisfied, an error-handling function you specify is called. This strategy relies on the convention that **when operator new cannot satisfy a request, it calls a client-specifiable error-handling function — often called a *new-handler* — before it throws an exception**. (In truth, what operator new really does is slightly more complicated. Details are provided in [Item 8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_FR.HTM#120851).) [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp7) [Item E7, P7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp7)

To specify the out-of-memory-handling function, clients call set\_new\_handler, which is specified in the header <new> more or less like this: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp8) [Item E7, P8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp8)

typedef void (\*new\_handler)();

new\_handler set\_new\_handler(new\_handler p) throw();

As you can see, new\_handler is a typedef for a pointer to a function that takes and returns nothing, and set\_new\_handler is a function that takes and returns a new\_handler. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp9) [Item E7, P9](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp9)

set\_new\_handler's parameter is a pointer to the function operator new should call if it can't allocate the requested memory. The return value of set\_new\_handler is a pointer to the function in effect for that purpose before set\_new\_handler was called. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp10) [Item E7, P10](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp10)

You use set\_new\_handler like this: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp11) [Item E7, P11](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp11)

// function to call if operator new can't allocate enough memory

void noMoreMemory()

{

cerr << "Unable to satisfy request for memory\n";

abort();

}

int main()

{

set\_new\_handler(noMoreMemory);

int \*pBigDataArray = new int[100000000];

...

}

If, as seems likely, operator new is unable to allocate space for 100,000,000 integers, noMoreMemory will be called, and the program will abort after issuing an error message. This is a marginally better way to terminate the program than a simple core dump. (By the way, consider what happens if memory must be dynamically allocated during the course of writing the error message to cerr...) [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp12) [Item E7, P12](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp12)

When operator new cannot satisfy a request for memory, it calls the new-handler function not once, but *repeatedly* until it can find enough memory. The code giving rise to these repeated calls is shown in [Item 8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_FR.HTM#120851), but this high-level description is enough to conclude that a well-designed new-handler function must do one of the following: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp13) [Item E7, P13](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp13)

* **Make more memory available.** This may allow operator new's next attempt to allocate the memory to succeed. One way to implement this strategy is to allocate a large block of memory at program start-up, then release it the first time the new-handler is invoked. Such a release is often accompanied by some kind of warning to the user that memory is low and that future requests may fail unless more memory is somehow made available. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp14) [Item E7, P14](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp14)
* **Install a different new-handler.** If the current new-handler can't make any more memory available, perhaps it knows of a different new-handler that is more resourceful. If so, the current new-handler can install the other new-handler in its place (by calling set\_new\_handler). The next time operator new calls the new-handler function, it will get the one most recently installed. (A variation on this theme is for a new-handler to modify its *own* behavior, so the next time it's invoked, it does something different. One way to achieve this is to have the new-handler modify static or global data that affects the new-handler's behavior.) [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp15) [Item E7, P15](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp15)
* **Deinstall the new-handler,** i.e., pass the null pointer to set\_new\_handler. With no new-handler installed, operator new will throw an exception of type std::bad\_alloc when its attempt to allocate memory is unsuccessful. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp16) [Item E7, P16](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp16)
* **Throw an exception** of type std::bad\_alloc or some type derived from std::bad\_alloc. Such exceptions will not be caught by operator new, so they will propagate to the site originating the request for memory. (Throwing an exception of a different type will violate operator new's exception specification. The default action when that happens is to call abort, so if your new-handler is going to throw an exception, you definitely want to make sure it's from the std::bad\_alloc hierarchy. For more information on exception specifications, see [Item M14](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI14_FR.HTM#6011).) [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp17) [Item E7, P17](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp17)
* **Not return**, typically by calling abort or exit, both of which are found in the standard C library (and thus in the standard C++ library — see [Item 49](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI49_FR.HTM#8392)). [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp18) [Item E7, P18](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp18)

These choices give you considerable flexibility in implementing new-handler functions. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp19) [Item E7, P19](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp19)

Sometimes you'd like to handle memory allocation failures in different ways, depending on the class of the object being allocated: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp20) [Item E7, P20](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp20)

class X {

public:

static void outOfMemory();

...

};

class Y {

public:

static void outOfMemory();

...

};

X\* p1 = new X; // if allocation is unsuccessful,

// call X::outOfMemory

Y\* p2 = new Y; // if allocation is unsuccessful,

// call Y::outOfMemory

C++ has no support for class-specific new-handlers, but it doesn't need to. You can implement this behavior yourself. You just have each class provide its own versions of set\_new\_handler and operator new. The class's set\_new\_handler allows clients to specify the new-handler for the class (just like the standard set\_new\_handler allows clients to specify the global new-handler). The class's operator new ensures that the class-specific new-handler is used in place of the global new-handler when memory for class objects is allocated. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp21) [Item E7, P21](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp21)

Consider a class X for which you want to handle memory allocation failures. You'll have to keep track of the function to call when operator new can't allocate enough memory for an object of type X, so you'll declare a static member of type new\_handler to point to the new-handler function for the class. Your class X will look something like this: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp22) [Item E7, P22](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp22)

class X {

public:

static new\_handler set\_new\_handler(new\_handler p);

static void \* operator new(size\_t size);

private:

static new\_handler currentHandler;

};

Static class members must be defined outside the class definition. Because you'll want to use the default initialization of static objects to 0, you'll define X::currentHandler without initializing it: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp23) [Item E7, P23](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp23)

new\_handler X::currentHandler; // sets currentHandler

// to 0 (i.e., null) by

// default

The set\_new\_handler function in class X will save whatever pointer is passed to it. It will return whatever pointer had been saved prior to the call. This is exactly what the standard version of set\_new\_handler does: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp24) [Item E7, P24](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp24)

new\_handler X::set\_new\_handler(new\_handler p)

{

new\_handler oldHandler = currentHandler;

currentHandler = p;

return oldHandler;

}

Finally, X's operator new will do the following: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp25) [Item E7, P25](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp25)

1. Call the standard set\_new\_handler with X's error-handling function. This will install X's new-handler as the global new- handler. In the code below, notice how you explicitly reference the std scope (where the standard set\_new\_handler resides) by using the "::" notation. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp26) [Item E7, P26](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp26)
2. Call the global operator new to actually allocate the requested memory. If the initial attempt at allocation fails, the global operator new will invoke X's new-handler, because that function was just installed as the global new-handler. If the global operator new is ultimately unable to find a way to allocate the requested memory, it will throw a std::bad\_alloc exception, which X's operator new will catch. X's operator new will then restore the global new-handler that was originally in place, and it will return by propagating the exception. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp27) [Item E7, P27](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp27)
3. Assuming the global operator new was able to successfully allocate enough memory for an object of type X, X's operator new will again call the standard set\_new\_handler to restore the global error-handling function to what it was originally. It will then return a pointer to the allocated memory. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp28) [Item E7, P28](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp28)

Here's how you say all that in C++: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp29) [Item E7, P29](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp29)

void \* X::operator new(size\_t size)

{

new\_handler globalHandler = // install X's

std::set\_new\_handler(currentHandler); // handler

void \*memory;

try { // attempt

memory = ::operator new(size); // allocation

}

catch (std::bad\_alloc&) { // restore

std::set\_new\_handler(globalHandler); // handler;

throw; // propagate

} // exception

std::set\_new\_handler(globalHandler); // restore

// handler

return memory;

}

If the duplicated calls to std::set\_new\_handler caught your eye, turn to [Item M9](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI9_FR.HTM#5292) for information on how to eliminate them. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp30) [Item E7, P30](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp30)

Clients of class X use its new-handling capabilities like this: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp31) [Item E7, P31](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp31)

void noMoreMemory(); // decl. of function to

// call if memory allocation

// for X objects fails

X::set\_new\_handler(noMoreMemory);

// set noMoreMemory as X's

// new-handling function

X \*px1 = new X; // if memory allocation

// fails, call noMoreMemory

string \*ps = new string; // if memory allocation

// fails, call the global

// new-handling function

// (if there is one)

X::set\_new\_handler(0); // set the X-specific

// new-handling function

// to nothing (i.e., null)

X \*px2 = new X; // if memory allocation

// fails,throw an exception

// immediately. (There is

// no new-handling function

// for class X.)

You may note that the code for implementing this scheme is the same regardless of the class, so a reasonable inclination would be to reuse it in other places. As [Item 41](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI41_FR.HTM#7611) explains, both inheritance and templates can be used to create reusable code. However, in this case, it's a combination of the two that gives you what you need. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp32) [Item E7, P32](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp32)

All you have to do is create a "mixin-style" base class, i.e., a base class that's designed to allow derived classes to inherit a single specific capability — in this case, the ability to set a class-specific new-handler. Then you turn the base class into a template. The base class part of the design lets derived classes inherit the set\_new\_handler and operator new functions they all need, while the template part of the design ensures that each inheriting class gets a different currentHandler data member. The result may sound a little complicated, but you'll find that the code looks reassuringly familiar. In fact, about the only real difference is that it's now reusable by any class that wants it: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp33) [Item E7, P33](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp33)

template<class T> // "mixin-style" base class

class NewHandlerSupport { // for class-specific

public: // set\_new\_handler support

static new\_handler set\_new\_handler(new\_handler p);

static void \* operator new(size\_t size);

private:

static new\_handler currentHandler;

};

template<class T>

new\_handler NewHandlerSupport<T>::set\_new\_handler(new\_handler p)

{

new\_handler oldHandler = currentHandler;

currentHandler = p;

return oldHandler;

}

template<class T>

void \* NewHandlerSupport<T>::operator new(size\_t size)

{

new\_handler globalHandler =

std::set\_new\_handler(currentHandler);

void \*memory;

try {

memory = ::operator new(size);

}

catch (std::bad\_alloc&) {

std::set\_new\_handler(globalHandler);

throw;

}

std::set\_new\_handler(globalHandler);

return memory;

}

// this sets each currentHandler to 0

template<class T>

new\_handler NewHandlerSupport<T>::currentHandler;

With this class template, adding set\_new\_handler support to class X is easy: X just inherits from newHandlerSupport<X>: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp34) [Item E7, P34](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp34)

// note inheritance from mixin base class template. (See

// [my article on counting objects](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MAGAZINE\CO_FRAME.HTM) for information on why

// private inheritance might be preferable here.)

class X: public NewHandlerSupport<X> {

... // as before, but no declarations for

}; // set\_new\_handler or operator new

Clients of X remain oblivious to all the behind-the-scenes action; their old code continues to work. This is good, because one thing you can usually rely on your clients being is oblivious. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp35) [Item E7, P35](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp35)

Using set\_new\_handler is a convenient, easy way to cope with the possibility of out-of-memory conditions. Certainly it's a lot more attractive than wrapping every use of new inside a try block. Furthermore, templates like NewHandlerSupport make it simple to add a class-specific new-handler to any class that wants one. Mixin-style inheritance, however, invariably leads to the topic of multiple inheritance, and before starting down that slippery slope, you'll definitely want to read [Item 43](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI43_FR.HTM#7778). [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp36) [Item E7, P36](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp36)

Until 1993, C++ required that operator new return 0 when it was unable to satisfy a memory request. The current behavior is for operator new to throw a std::bad\_alloc exception, but a lot of C++ was written before compilers began supporting the revised specification. The **°**[C++ standardization committee](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=committee) didn't want to abandon the established test-for-0 code base, so they provided alternative forms of operator new (and operator new[] — see [Item 8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_FR.HTM#120851)) that continue to offer the traditional failure-yields-0 behavior. These forms are called "**nothrow**" forms because, well, they never do a throw, and they employ nothrow objects (defined in the standard header <new>) at the point where new is used: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp37) [Item E7, P37](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp37)

class Widget { ... };

Widget \*pw1 = new Widget; // throws std::bad\_alloc if

// allocation fails

if (pw1 == 0) ... // this test *must* fail

Widget \*pw2 =

new (nothrow) Widget; // returns 0 if allocation

// fails

if (pw2 == 0) ... // this test may succeed

Regardless of whether you use "normal" (i.e., exception-throwing) new or "nothrow" new, it's important that you be prepared to handle memory allocation failures. The easiest way to do that is to take advantage of set\_new\_handler, because it works with both forms. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp38) [Item E7, P38](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp38)

## Item 8:  Adhere to convention when writing operator new and operator delete. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp1) [Item E8, P1](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp1)

When you take it upon yourself to write operator new ([Item 10](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI10_FR.HTM#1986) explains why you might want to), it's important that your function(s) offer behavior that is consistent with the default operator new. In practical terms, this means having the right return value, calling an error-handling function when insufficient memory is available (see [Item 7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_FR.HTM#1894)), and being prepared to cope with requests for no memory. You also need to avoid inadvertently hiding the "normal" form of new, but that's a topic for [Item 9](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_FR.HTM#1961). [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp2) [Item E8, P2](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp2)

The return value part is easy. If you can supply the requested memory, you just return a pointer to it. If you can't, you follow the rule described in [Item 7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_FR.HTM#1894) and throw an exception of type std::bad\_alloc. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp3) [Item E8, P3](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp3)

It's not quite that simple, however, because operator new actually tries to allocate memory more than once, calling the error-handling function after each failure, the assumption being that the error-handling function might be able to do something to free up some memory. Only when the pointer to the error-handling function is null does operator new throw an exception. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp4) [Item E8, P4](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp4)

In addition, the **°**[C++ standard](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=cstandard) requires that operator new return a legitimate pointer even when 0 bytes are requested. (Believe it or not, requiring this odd-sounding behavior actually simplifies things elsewhere in the language.) [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp5) [Item E8, P5](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp5)

That being the case, pseudocode for a non-member operator new looks like this: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp6) [Item E8, P6](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp6)

void \* operator new(size\_t size) // your operator new might

{ // take additional params

if (size == 0) { // handle 0-byte requests

size = 1; // by treating them as

} // 1-byte requests

while (1) {

*attempt to allocate* size *bytes*;

if (*the allocation was successful*)

return (*a pointer to the memory*);

// allocation was unsuccessful; find out what the

// current error-handling function is (see [Item 7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_FR.HTM#1894))

new\_handler globalHandler = set\_new\_handler(0);

set\_new\_handler(globalHandler);

if (globalHandler) (\*globalHandler)();

elsethrow std::bad\_alloc();

}

}

The trick of treating requests for zero bytes as if they were really requests for one byte looks slimy, but it's simple, it's legal, it works, and how often do you expect to be asked for zero bytes, anyway? [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp7) [Item E8, P7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp7)

You may also look askance at the place in the pseudocode where the error-handling function pointer is set to null, then promptly reset to what it was originally. Unfortunately, there is no way to get at the error-handling function pointer directly, so you have to call set\_new\_handler to find out what it is. Crude, yes, but also effective. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp8) [Item E8, P8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp8)

[Item 7](file:///C:\\zLei\\%7EzLei\\%7Ezlcv\\1_Effective%20C++%202nd\\EC\\EI7_FR.HTM" \l "1894" \t "_top) remarks that operator new contains an infinite loop, and the code above shows that loop explicitly — while (1) is about as infinite as it gets. The only way out of the loop is for memory to be successfully allocated or for the new-handling function to do one of the things described in [Item 7](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI7_FR.HTM#1894): make more memory available, install a different new-handler, deinstall the new-handler, throw an exception of or derived from std::bad\_alloc, or fail to return. It should now be clear why the new-handler must do one of those things. If it doesn't, the loop inside operator new will never terminate. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp9) [Item E8, P9](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp9)

One of the things many people don't realize about operator new is that it's inherited by subclasses. That can lead to some interesting complications. In the pseudocode for operator new above, notice that the function tries to allocate size bytes (unless size is 0). That makes perfect sense, because that's the argument that was passed to the function. However, most class-specific versions of operator new (including the one you'll find in [Item 10](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI10_FR.HTM#1986)) are designed for a *specific* class, *not* for a class *or* any of its subclasses. That is, given an operator new for a class X, the behavior of that function is almost always carefully tuned for objects of size sizeof(X) — nothing larger and nothing smaller. Because of inheritance, however, it is possible that the operator new in a base class will be called to allocate memory for an object of a derived class: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp10) [Item E8, P10](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp10)

class Base {

public:

static void \* operator new(size\_t size);

...

};

class Derived: public Base // Derived doesn't declare

{ ... }; // operator new

Derived \*p = new Derived; // calls Base::operator new!

If Base's class-specific operator new wasn't designed to cope with this — and chances are slim that it was — the best way for it to handle the situation is to slough off calls requesting the "wrong" amount of memory to the standard operator new, like this: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp11) [Item E8, P11](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp11)

void \* Base::operator new(size\_t size)

{

if (size != sizeof(Base)) // if size is "wrong,"

return ::operator new(size); // have standard operator

// new handle the request

... // otherwise handle

// the request here

}

"Hold on!" I hear you cry, "You forgot to check for the pathological-but-nevertheless-possible case where size is zero!" Actually, I didn't, and please stop using hyphens when you cry out. The test is still there, it's just been incorporated into the test of size against sizeof(Base). The **°**[C++ standard](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=cstandard) works in mysterious ways, and one of those ways is to decree that all freestanding classes have nonzero size. By definition, sizeof(Base) can never be zero (even if it has no members), so if size is zero, the request will be forwarded to ::operator new, and it will become that function's responsibility to treat the request in a reasonable fashion. (Interestingly, sizeof(Base) may be zero if Base is not a freestanding class. For details, consult [my article on counting objects](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MAGAZINE\CO_FRAME.HTM).) [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp12) [Item E8, P12](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp12)

If you'd like to control memory allocation for arrays on a per-class basis, you need to implement operator new's array-specific cousin, operator new[]. (This function is usually called "array new," because it's hard to figure out how to pronounce "operator new[]".) If you decide to write operator new[], remember that all you're doing is allocating raw memory — you can't do anything to the as-yet-nonexistent objects in the array. In fact, you can't even figure out how many objects will be in the array, because you don't know how big each object is. After all, a base class's operator new[] might, through inheritance, be called to allocate memory for an array of derived class objects, and derived class objects are usually bigger than base class objects. Hence, you can't assume inside Base::operator new[] that the size of each object going into the array is sizeof(Base), and that means you can't assume that the number of objects in the array is (*bytes* *requested*)/sizeof(Base). For more information on operator new[], see [Item M8](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\MEC\MI8_FR.HTM#33985). [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp13) [Item E8, P13](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp13)

So much for the conventions you need to follow when writing operator new (and operator new[]). For operator delete (and its array counterpart, operator delete[]), things are simpler. About all you need to remember is that C++ guarantees it's always safe to delete the null pointer, so you need to honor that guarantee. Here's pseudocode for a non-member operator delete: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp14) [Item E8, P14](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp14)

void operator delete(void \*rawMemory)

{

**if (rawMemory == 0) return**; // do nothing if the null

// pointer is being deleted

*deallocate the memory pointed to by* rawMemory;

return;

}

The member version of this function is simple, too, except you've got to be sure to check the size of what's being deleted. Assuming your class-specific operator new forwards requests of the "wrong" size to ::operator new, you've got to forward "wrongly sized" deletion requests to ::operator delete: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp15) [Item E8, P15](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp15)

class Base { // same as before, but now

public: // op. delete is declared

static void \* operator new(size\_t size);

static void operator delete(void \*rawMemory, size\_t size);

...

};

void Base::operator delete(void \*rawMemory, size\_t size)

{

if (rawMemory == 0) return; // check for null pointer

if (size != sizeof(Base)) { // if size is "wrong,"

::operator delete(rawMemory); // have standard operator

return; // delete handle the request

}

*deallocate the memory pointed to by* rawMemory;

return;

}

The conventions, then, for operator new and operator delete (and their array counterparts) are not particularly onerous, but it is important that you obey them. If your allocation routines support new-handler functions and correctly deal with zero-sized requests, you're all but finished, and if your deallocation routines cope with null pointers, there's little more to do. Add support for inheritance in member versions of the functions, and *presto!* — you're done.

## Item 9:  Avoid hiding the "normal" form of new. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp1) [Item E9, P1](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp1)

A declaration of a name in an inner scope hides the same name in outer scopes, so for a function f at both global and class scope, the member function will hide the global function: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp2) [Item E9, P2](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp2)

void f(); // global function

class X {

public:

void f(); // member function

};

X x;

f(); // calls global f

x.f(); // calls X::f

This is unsurprising and normally causes no confusion, because global and member functions are usually invoked using different syntactic forms. However, if you add to this class an operator new taking additional parameters, the result is likely to be an eye-opener: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp3) [Item E9, P3](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp3)

class X {

public:

void f();

// operator new allowing specification of a

// new-handling function

static void \* operator new(size\_t size, new\_handler p);

};

void specialErrorHandler(); // definition is elsewhere

X \*px1 =

new (specialErrorHandler) X; // calls X::operator new

X \*px2 = new X; // error!

By declaring a function called "operator new" inside the class, you inadvertently block access to the "normal" form of new. Why this is so is discussed in [Item 50](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI50_FR.HTM#8569). Here we're more interested in figuring out how to avoid the problem. [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp4) [Item E9, P4](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp4)

One solution is to write a class-specific operator new that supports the "normal" invocation form. If it does the same thing as the global version, that can be efficiently and elegantly encapsulated as an inline function: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp5) [Item E9, P5](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp5)

class X {

public:

void f();

static void \* operator new(size\_t size, new\_handler p);

static void \* operator new(size\_t size)

{ return ::operator new(size); }

};

X \*px1 =

new (specialErrorHandler) X; // calls X::operator

// new(size\_t, new\_handler)

X\* px2 = new X; // calls X::operator

// new(size\_t)

An alternative is to provide a default parameter value (see [Item 24](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI24_FR.HTM#6250)) for each additional parameter you add to operator new: [¤](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp6) [Item E9, P6](file:///C:\zLei\%7EzLei\%7Ezlcv\1_Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp6)

class X {

public:

void f();

static

void \* operator new(size\_t size, // note default

new\_handler p = 0); // value for p

};

X \*px1 = new (specialErrorHandler) X; // fine

X\* px2 = new X; // also fine

Either way, if you later decide to customize the behavior of the "normal" form of new, all you need to do is rewrite the function; callers will get the customized behavior automatically when they relink

# Exceptions

## Introduction

The addition of exceptions to C++ changes things. Profoundly. Radically. Possibly uncomfortably. The use of raw, unadorned pointers, for example, becomes risky. Opportunities for resource leaks increase in number. It becomes more difficult to write constructors and destructors that behave the way we want them to. Special care must be taken to prevent program execution from abruptly halting. Executables and libraries typically increase in size and decrease in speed

And these are just the things we know. There is much the C++ community does not know about writing programs using exceptions, including, for the most part, how to do it correctly. There is as yet no agreement on a body of techniques that, when applied routinely, leads to software that behaves predictably and reliably when exceptions are thrown. (For insight into some of the issues involved, see [the article by Tom Cargill](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MAGAZINE/CA_FRAME.HTM). For information on recent progress in dealing with these issues, see the articles by [Jack Reeves](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MAGAZINE/RE_FRAME.HTM) and [Herb Sutter](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MAGAZINE/SU_FRAME.HTM).) [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MEXCEPDR.HTM#dingp3) [MEC++ Exceptions, P3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MEXCEPDR.HTM#dingp3)

We do know this much: programs that behave well in the presence of exceptions do so because they were *designed* to, not because they happen to. Exception-safe programs are not created by accident. The chances of a program behaving well in the presence of exceptions when it was not designed for exceptions are about the same as the chances of a program behaving well in the presence of multiple threads of control when it was not designed for multi-threaded execution: about zero. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MEXCEPDR.HTM#dingp4) [MEC++ Exceptions, P4](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MEXCEPDR.HTM#dingp4)

### Why exception? It cannot be ignored

That being the case, why use exceptions? Error codes have sufficed for C programmers ever since C was invented, so why mess with exceptions, especially if they're as problematic as I say? The answer is simple: **exceptions cannot be ignored**. If a function signals an exceptional condition by setting a status variable or returning an error code, there is no way to guarantee the function's caller will check the variable or examine the code. As a result, execution may continue long past the point where the condition was encountered. If the function signals the condition by throwing an exception, however, and that exception is not caught, program execution immediately ceases.

### Setjmp/longjmp does not call destructor

This is behavior that C programmers can approach only by using setjmp and longjmp. But longjmp exhibits a serious deficiency when used with C++: it fails to call destructors for local objects when it adjusts the stack. Most C++ programs depend on such destructors being called, so setjmp and longjmp make a poor substitute for true exceptions. If you need a way of signaling exceptional conditions that cannot be ignored, and if you must ensure that local destructors are called when searching the stack for code that can handle exceptional conditions, you need C++ exceptions. It's as simple as that. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MEXCEPDR.HTM#dingp6) [MEC++ Exceptions, P6](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MEXCEPDR.HTM#dingp6)

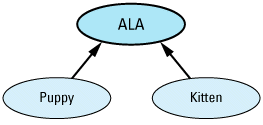
Because we have much to learn about programming with exceptions, the Items that follow comprise an incomplete guide to writing exception-safe software. Nevertheless, they introduce important considerations for anyone using exceptions in C++. By heeding the guidance in the material below (and in the [magazine articles on this CD](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MAGAZINE/MA_FRAME.HTM)), you'll improve the correctness, robustness, and efficiency of the software you write, and you'll sidestep many problems that commonly arise when working with exceptions.

## Item 9:  Use destructors to prevent resource leaks

Say good-bye to pointers. Admit it: you never really liked them that much anyway

Okay, you don't have to say good-bye to *all* pointers, but you do need to say *sayonara* to pointers that are used to manipulate local resources. Suppose, for example, you're writing software at the Shelter for Adorable Little Animals, an organization that finds homes for puppies and kittens. Each day the shelter creates a file containing information on the adoptions it arranged that day, and your job is to write a program to read these files and do the appropriate processing for each adoption.

A reasonable approach to this task is to define an abstract base class, ALA ("Adorable Little Animal"), plus concrete derived classes for puppies and kittens. A virtual function, processAdoption, handles the necessary species-specific processing: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp4) [Item M9, P4](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp4)



class ALA {

public:

virtual void processAdoption() = 0;

...

};

class Puppy: public ALA {

public:

virtual void processAdoption();

...

};

class Kitten: public ALA {

public:

virtual void processAdoption();

...

};

You'll need a function that can read information from a file and produce either a Puppy object or a Kitten object, depending on the information in the file. This is a perfect job for a *virtual constructor*, a kind of function described in [Item 25](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI25_FR.HTM#5341). For our purposes here, the function's declaration is all we need: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp5) [Item M9, P5](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp5)

// read animal information from s, then return a pointer

// to a newly allocated object of the appropriate type

ALA \* readALA(istream& s);

The heart of your program is likely to be a function that looks something like this:

void processAdoptions(istream& dataSource)

{

while (dataSource) { // while there's data

ALA \*pa = readALA(dataSource); // get next animal

pa->processAdoption(); // process adoption

delete pa; // delete object that

} // readALA returned

}

This function loops through the information in dataSource, processing each entry as it goes. The only mildly tricky thing is the need to remember to delete pa at the end of each iteration. This is necessary because readALA creates a new heap object each time it's called. Without the call to delete, the loop would contain a resource leak.

Now consider what would happen if pa->processAdoption threw an exception. processAdoptions fails to catch exceptions, so the exception would propagate to processAdoptions's caller. In doing so, all statements in processAdoptions after the call to pa->processAdoption would be skipped, and that means pa would never be deleted. As a result, anytime pa->processAdoption throws an exception, processAdoptions contains a resource leak.

Plugging the leak is easy enough, [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp9) [Item M9, P9](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp9)

void processAdoptions(istream& dataSource)

{

while (dataSource) {

ALA \*pa = readALA(dataSource);

try {

pa->processAdoption();

}

catch (...) { // catch all exceptions

delete pa; // avoid resource leak when an

// exception is thrown

throw; // propagate exception to caller

}

delete pa; // avoid resource leak when no

} // exception is thrown

}

but then you have to litter your code with try and catch blocks. More importantly, you are forced to duplicate cleanup code that is common to both normal and exceptional paths of control. In this case, the call to delete must be duplicated. Like all replicated code, this is annoying to write and difficult to maintain, but it also *feels wrong*. Regardless of whether we leave processAdoptions by a normal return or by throwing an exception, we need to delete pa, so why should we have to say that in more than one place?

We don't have to if we can somehow move the cleanup code that must always be executed into the destructor for an object local to processAdoptions. That's because local objects are always destroyed when leaving a function, regardless of how that function is exited. (The only exception to this rule is when you call longjmp, and this shortcoming of longjmp is the primary reason why C++ has support for exceptions in the first place.) Our real concern, then, is moving the delete from processAdoptions into a destructor for an object local to processAdoptions.

The solution is to replace the pointer pa with an object that *acts like* a pointer. That way, when the pointer-like object is (automatically) destroyed, we can have its destructor call delete. Objects that act like pointers, but do more, are called *smart pointers*, and, as [Item 28](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI28_FR.HTM#61766) explains, you can make pointer-like objects very smart indeed. In this case, we don't need a particularly brainy pointer, we just need a pointer-like object that knows enough to delete what it points to when the pointer-like object goes out of scope.

### Using auto\_ptr

It's not difficult to write a class for such objects, but we don't need to. The standard C++ library (see [Item E49](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI49_FR.HTM#8392)) contains a class template called auto\_ptr that does just what we want. Each auto\_ptr class takes a pointer to a heap object in its constructor and deletes that object in its destructor. Boiled down to these essential functions, auto\_ptr looks like this:

template<class T>

class auto\_ptr {

public:

auto\_ptr(T \*p = 0): ptr(p) {} // save ptr to object

~auto\_ptr() { delete ptr; } // delete ptr to object

private:

T \*ptr; // raw ptr to object

};

The standard version of auto\_ptr is much fancier, and this stripped-down implementation isn't suitable for real use[3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9.HTM#74869) (we must add at least the copy constructor, assignment operator, and pointer-emulating functions discussed in [Item 28](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI28_FR.HTM#61766)), but the concept behind it should be clear: use auto\_ptr objects instead of raw pointers, and you won't have to worry about heap objects not being deleted, not even when exceptions are thrown. (Because the auto\_ptr destructor uses the single-object form of delete, auto\_ptr is not suitable for use with pointers to *arrays* of objects. If you'd like an auto\_ptr-like template for arrays, you'll have to write your own. In such cases, however, it's often a better design decision to use a vector instead of an array, anyway.)

Using an auto\_ptr object instead of a raw pointer, processAdoptions looks like this:

void processAdoptions(istream& dataSource)

{

while (dataSource) {

auto\_ptr<ALA> pa(readALA(dataSource));

pa->processAdoption();

}

}

This version of processAdoptions differs from the original in only two ways. First, pa is declared to be an auto\_ptr<ALA> object, not a raw ALA\* pointer. Second, there is no delete statement at the end of the loop. That's it. Everything else is identical, because, except for destruction, auto\_ptr objects act just like normal pointers. Easy, huh?

### Wrap a resource into an object

The idea behind auto\_ptr — using an object to store a resource that needs to be automatically released and relying on that object's destructor to release it — applies to more than just pointer-based resources. Consider a function in a GUI application that needs to create a window to display some information: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp17) [Item M9, P17](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp17)

// this function may leak resources if an exception

// is thrown

void displayInfo(const Information& info)

{

WINDOW\_HANDLE w(createWindow());

*display info in window corresponding to w;*

destroyWindow(w);

}

Many window systems have C-like interfaces that use functions like createWindow and destroyWindow to acquire and release window resources. If an exception is thrown during the process of displaying info in w, the window for which w is a handle will be lost just as surely as any other dynamically allocated resource. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp18) [Item M9, P18](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp18)

The solution is the same as it was before. Create a class whose constructor and destructor acquire and release the resource: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp19) [Item M9, P19](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp19)

// class for acquiring and releasing a window handle

class WindowHandle {

public:

WindowHandle(WINDOW\_HANDLE handle): w(handle) {}

~WindowHandle() { destroyWindow(w); }

operator WINDOW\_HANDLE() { return w; } // see below

private:

WINDOW\_HANDLE w;

// The following functions are declared private to prevent

// multiple copies of a WINDOW\_HANDLE from being created.

// See [Item 28](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI28_FR.HTM#61766) for a discussion of a more flexible approach.

WindowHandle(const WindowHandle&);

WindowHandle& operator=(const WindowHandle&);

};

This looks just like the auto\_ptr template, except that assignment and copying are explicitly prohibited (see [Item E27](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI27_FR.HTM#6406)), and there is an implicit conversion operator that can be used to turn a WindowHandle into a WINDOW\_HANDLE. This capability is essential to the practical application of a WindowHandle object, because it means you can use a WindowHandle just about anywhere you would normally use a raw WINDOW\_HANDLE. (See [Item 5](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI5_FR.HTM#5970), however, for why you should generally be leery of implicit type conversion operators.) [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp20) [Item M9, P20](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp20)

Given the WindowHandle class, we can rewrite displayInfo as follows: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp21) [Item M9, P21](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp21)

// this function avoids leaking resources if an

// exception is thrown

void displayInfo(const Information& info)

{

WindowHandle w(createWindow());

*display info in window corresponding to w;*

}

Even if an exception is thrown within displayInfo, the window created by createWindow will always be destroyed. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp22) [Item M9, P22](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_DIR.HTM#dingp22)

By adhering to the rule that resources should be encapsulated inside objects, you can usually avoid resource leaks in the presence of exceptions. But what happens if an exception is thrown while you're in the process of acquiring a resource, e.g., while you're in the constructor of a resource-acquiring class? What happens if an exception is thrown during the automatic destruction of such resources? Don't constructors and destructors call for special techniques? They do, and you can read about them in Items [10](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_FR.HTM#38223) and [11](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_FR.HTM#39749)

## Item 10:  Prevent resource leaks in constructors.

Imagine you're developing software for a multimedia address book. Such an address book might hold, in addition to the usual textual information of a person's name, address, and phone numbers, a picture of the person and the sound of their voice (possibly giving the proper pronunciation of their name). [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp2) [Item M10, P2](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp2)

To implement the book, you might come up with a design like this: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp3) [Item M10, P3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp3)

class Image { // for image data

public:

Image(const string& imageDataFileName);

...

};

class AudioClip { // for audio data

public:

AudioClip(const string& audioDataFileName);

...

};

class PhoneNumber { ... }; // for holding phone numbers

class BookEntry { // for each entry in the

public: // address book

BookEntry(const string& name,

const string& address = "",

const string& imageFileName = "",

const string& audioClipFileName = "");

~BookEntry();

// phone numbers are added via this function

void addPhoneNumber(const PhoneNumber& number);

...

private:

string theName; // person's name

string theAddress; // their address

list<PhoneNumber> thePhones; // their phone numbers

Image \*theImage; // their image

AudioClip \*theAudioClip; // an audio clip from them

};

Each BookEntry must have name data, so you require that as a constructor argument (see [Item 3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI3_FR.HTM#84818)), but the other fields — the person's address and the names of files containing image and audio data — are optional. Note the use of the list class to hold the person's phone numbers. This is one of several container classes that are part of the standard C++ library (see [Item E49](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI49_FR.HTM#8392) and [Item 35](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI35_FR.HTM#5473)).

A straightforward way to write the BookEntry constructor and destructor is as follows:

BookEntry::BookEntry(const string& name,

const string& address,

const string& imageFileName,

Const string& audioClipFileName)

: theName(name), theAddress(address),

theImage(0), theAudioClip(0)

{

if (imageFileName != "") {

theImage = new Image(imageFileName);

}

if (audioClipFileName != "") {

theAudioClip = new AudioClip(audioClipFileName);

}

}

BookEntry::~BookEntry()

{

delete theImage;

delete theAudioClip;

}

The constructor initializes the pointers theImage and theAudioClip to null, then makes them point to real objects if the corresponding arguments are non-empty strings. The destructor deletes both pointers, thus ensuring that a BookEntry object doesn't give rise to a resource leak. Because C++ guarantees it's safe to delete null pointers, BookEntry's destructor need not check to see if the pointers actually point to something before deleting them.

Everything looks fine here, and under normal conditions everything is fine, but under abnormal conditions — under *exceptional* conditions — things are not fine at all. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp7) [Item M10, P7](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp7)

Consider what will happen if an exception is thrown during execution of this part of the BookEntry constructor: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp8) [Item M10, P8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp8)

if (audioClipFileName != "") {

theAudioClip = new AudioClip(audioClipFileName);

}

An exception might arise because operator new (see [Item 8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI8_FR.HTM#33985)) is unable to allocate enough memory for an AudioClip object. One might also arise because the AudioClip constructor itself throws an exception. Regardless of the cause of the exception, if one is thrown within the BookEntry constructor, it will be propagated to the site where the BookEntry object is being created.

Now, if an exception is thrown during creation of the object theAudioClip is supposed to point to (thus transferring control out of the BookEntry constructor), who deletes the object that theImage already points to? The obvious answer is that BookEntry's destructor does, but the obvious answer is wrong. BookEntry's destructor will never be called. Never.

### C++ destroys only fully constructed objects

**C++ destroys only *fully constructed* objects**, and an object isn't fully constructed until its constructor has run to completion. So if a BookEntry object b is created as a local object,

void testBookEntryClass()

{

BookEntry b("Addison-Wesley Publishing Company",

"One Jacob Way, Reading, MA 01867");

...

}

and an exception is thrown during construction of b, b's destructor will not be called. Furthermore, if you try to take matters into your own hands by allocating b on the heap and then calling delete if an exception is thrown, [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp12) [Item M10, P12](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp12)

void testBookEntryClass()

{

BookEntry \*pb = 0;

try {

pb = new BookEntry("Addison-Wesley Publishing Company",

"One Jacob Way, Reading, MA 01867");

...

}

catch (...) { // catch all exceptions

delete pb; // delete pb when an

// exception is thrown

throw; // propagate exception to

} // caller

delete pb; // delete pb normally

}

you'll find that the Image object allocated inside BookEntry's constructor is still lost, because no assignment is made to pb unless the new operation succeeds. If BookEntry's constructor throws an exception, pb will be the null pointer, so deleting it in the catch block does nothing except make you feel better about yourself. Using the smart pointer class auto\_ptr<BookEntry> (see [Item 9](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_FR.HTM#5292)) instead of a raw BookEntry\* won't do you any good either, because the assignment to pb still won't be made unless the new operation succeeds. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp13) [Item M10, P13](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp13)

There is a reason why C++ refuses to call destructors for objects that haven't been fully constructed, and it's not simply to make your life more difficult. It's because it would, in many cases, be a nonsensical thing — possibly a harmful thing — to do. If a destructor were invoked on an object that wasn't fully constructed, how would the destructor know what to do? The only way it could know would be if bits had been added to each object indicating how much of the constructor had been executed. Then the destructor could check the bits and (maybe) figure out what actions to take. Such bookkeeping would slow down constructors, and it would make each object larger, too. C++ avoids this overhead, but the price you pay is that partially constructed objects aren't automatically destroyed. (For an example of a similar trade-off involving efficiency and program behavior, turn to [Item E13](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI13_FR.HTM#2117).) [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp14) [Item M10, P14](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp14)

Because C++ won't clean up after objects that throw exceptions during construction, you must design your constructors so that they clean up after themselves. Often, this involves simply catching all possible exceptions, executing some cleanup code, then rethrowing the exception so it continues to propagate. This strategy can be incorporated into the BookEntry constructor like this:

BookEntry::BookEntry(const string& name,

const string& address,

const string& imageFileName,

const string& audioClipFileName)

: theName(name), theAddress(address),

theImage(0), theAudioClip(0)

{

try { // this try block is new

if (imageFileName != "") {

theImage = new Image(imageFileName);

}

if (audioClipFileName != "") {

theAudioClip = new AudioClip(audioClipFileName);

}

}

catch (...) { // catch any exception

delete theImage; // perform necessary

delete theAudioClip; // cleanup actions

throw; // propagate the exception

}

}

There is no need to worry about BookEntry's non-pointer data members. Data members are automatically initialized before a class's constructor is called, so if a BookEntry constructor body begins executing, the object's theName, theAddress, and thePhones data members have already been fully constructed. As fully constructed objects, these data members will be automatically destroyed when the BookEntry object containing them is, and there is no need for you to intervene. Of course, if these objects' constructors call functions that might throw exceptions, *those* constructors have to worry about catching the exceptions and performing any necessary cleanup before allowing them to propagate. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp16) [Item M10, P16](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp16)

You may have noticed that the statements in BookEntry's catch block are almost the same as those in BookEntry's destructor. Code duplication here is no more tolerable than it is anywhere else, so the best way to structure things is to move the common code into a private helper function and have both the constructor and the destructor call it: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp17) [Item M10, P17](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp17)

class BookEntry {

public:

... // as before

private:

...

void cleanup(); // common cleanup statements

};

void BookEntry::cleanup()

{

delete theImage;

delete theAudioClip;

}

BookEntry::BookEntry(const string& name,

const string& address,

const string& imageFileName,

const string& audioClipFileName)

: theName(name), theAddress(address),

theImage(0), theAudioClip(0)

{

try {

... // as before

}

catch (...) {

cleanup(); // release resources

throw; // propagate exception

}

}

BookEntry::~BookEntry()

{

cleanup();

}

This is nice, but it doesn't put the topic to rest. Let us suppose we design our BookEntry class slightly differently so that theImage and theAudioClip are *constant* pointers:

class BookEntry {

public:

... // as above

private:

...

Image \* const theImage; // pointers are now

AudioClip \* const theAudioClip; // const

};

Such pointers must be initialized via the member initialization lists of BookEntry's constructors, because there is no other way to give const pointers a value (see [Item E12](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI12_FR.HTM#2071)). A common temptation is to initialize theImage and theAudioClip like this,

// an implementation that may leak resources if an

// exception is thrown

BookEntry::BookEntry(const string& name,

const string& address,

const string& imageFileName,

const string& audioClipFileName)

: theName(name), theAddress(address),

theImage(imageFileName != ""

? new Image(imageFileName)

: 0),

theAudioClip(audioClipFileName != ""

? new AudioClip(audioClipFileName)

: 0)

{}

but this leads to the problem we originally wanted to eliminate: if an exception is thrown during initialization of theAudioClip, the object pointed to by theImage is never destroyed. Furthermore, we can't solve the problem by adding try and catch blocks to the constructor, because try and catch are statements, and member initialization lists allow only expressions. (That's why we had to use the ?: syntax instead of the if-then-else syntax in the initialization of theImage and theAudioClip.) [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp20) [Item M10, P20](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp20)

Nevertheless, the only way to perform cleanup chores before exceptions propagate out of a constructor is to catch those exceptions, so if we can't put try and catch in a member initialization list, we'll have to put them somewhere else. One possibility is inside private member functions that return pointers with which theImage and theAudioClip should be initialized:

class BookEntry {

public:

... // as above

private:

... // data members as above

**Image \*** initImage(const string& imageFileName);

AudioClip \* initAudioClip(const string&

audioClipFileName);

};

BookEntry::BookEntry(const string& name,

const string& address,

const string& imageFileName,

const string& audioClipFileName)

: theName(name), theAddress(address),

theImage(initImage(imageFileName)),

theAudioClip(initAudioClip(audioClipFileName))

{}

// theImage is initialized first, so there is no need to

// worry about a resource leak if this initialization

// fails. This function therefore handles no exceptions

Image \* BookEntry::initImage(const string& imageFileName)

{

if (imageFileName != "") return new Image(imageFileName);

else return 0;

}

// theAudioClip is initialized second, so it must make

// sure theImage's resources are released if an exception

// is thrown during initialization of theAudioClip. That's

// why this function uses try...catch.

AudioClip \* BookEntry::initAudioClip(const string&

audioClipFileName)

{

try {

if (audioClipFileName != "") {

return new AudioClip(audioClipFileName);

}

else return 0;

}

catch (...) {

delete theImage;

throw;

}

}

This is perfectly kosher, and it even solves the problem we've been laboring to overcome. The drawback is that code that conceptually belongs in a constructor is now dispersed across several functions, and that's a maintenance headache.

A better solution is to adopt the advice of [Item 9](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_FR.HTM#5292) and treat the objects pointed to by theImage and theAudioClip as resources to be managed by local objects. This solution takes advantage of the facts that both theImage and theAudioClip are pointers to dynamically allocated objects and that those objects should be deleted when the pointers themselves go away. This is precisely the set of conditions for which the auto\_ptr classes (see [Item 9](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI9_FR.HTM#5292)) were designed. We can therefore change the raw pointer types of theImage and theAudioClip to their auto\_ptr equivalents: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp23) [Item M10, P23](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp23)

class BookEntry {

public:

... // as above

private:

...

const auto\_ptr<Image> theImage; // these are now

const auto\_ptr<AudioClip> theAudioClip; // auto\_ptr objects

};

Doing this makes BookEntry's constructor leak-safe in the presence of exceptions, and it lets us initialize theImage and theAudioClip using the member initialization list: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp24) [Item M10, P24](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp24)

BookEntry::BookEntry(const string& name,

const string& address,

const string& imageFileName,

const string& audioClipFileName)

: theName(name), theAddress(address),

theImage(imageFileName != ""? new Image(imageFileName) : 0),

theAudioClip(audioClipFileName!="" ? new AudioClip(audioClipFileName):0)

{}

In this design, if an exception is thrown during initialization of theAudioClip, theImage is already a fully constructed object, so it will automatically be destroyed, just like theName, theAddress, and thePhones. Furthermore, because theImage and theAudioClip are now objects, they'll be destroyed automatically when the BookEntry object containing them is. Hence there's no need to manually delete what they point to. That simplifies BookEntry's destructor considerably:

BookEntry::~BookEntry()

{} // nothing to do!

This means you could eliminate BookEntry's destructor entirely. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp26) [Item M10, P26](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp26)

It all adds up to this: if you replace pointer class members with their corresponding auto\_ptr objects, you fortify your constructors against resource leaks in the presence of exceptions, you eliminate the need to manually deallocate resources in destructors, and you allow const member pointers to be handled in the same graceful fashion as non-const pointers. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp27) [Item M10, P27](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_DIR.HTM#dingp27)

Dealing with the possibility of exceptions during construction can be tricky, but auto\_ptr (and auto\_ptr-like classes) can eliminate most of the drudgery. Their use leaves behind code that's not only easy to understand, it's robust in the face of exceptions, too.

## Item 11:  Prevent exceptions from leaving destructors.

There are two situations in which a destructor is called. The first is when an object is destroyed under "normal" conditions, e.g., when it goes out of scope or is explicitly deleted. The second is when an object is destroyed by the exception-handling mechanism during the stack-unwinding part of exception propagation. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp2) [Item M11, P2](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp2)

That being the case, an exception may or may not be active when a destructor is invoked. Regrettably, there is no way to distinguish between these conditions from inside a destructor.[4](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11.HTM#79380) As a result, you must write your destructors under the conservative assumption that an exception *is* active, because if control leaves a destructor due to an exception while another exception is active, C++ calls the terminate function. That function does just what its name suggests: it terminates execution of your program. Furthermore, it terminates it *immediately*; not even local objects are destroyed. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp3) [Item M11, P3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp3)

As an example, consider a Session class for monitoring on-line computer sessions, i.e., things that happen from the time you log in through the time you log out. Each Session object notes the date and time of its creation and destruction: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp4) [Item M11, P4](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp4)

class Session {

public:

Session();

~Session();

...

private:

static void logCreation(Session \*objAddr);

static void logDestruction(Session \*objAddr);

};

The functions logCreation and logDestruction are used to record object creations and destructions, respectively. We might therefore expect that we could code Session's destructor like this: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp5) [Item M11, P5](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp5)

Session::~Session()

{

logDestruction(this);

}

This looks fine, but consider what would happen if logDestruction throws an exception. The exception would not be caught in Session's destructor, so it would be propagated to the caller of that destructor. But if the destructor was itself being called because some other exception had been thrown, the terminate function would automatically be invoked, and that would stop your program dead in its tracks. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp6) [Item M11, P6](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp6)

In many cases, this is not what you'll want to have happen. It may be unfortunate that the Session object's destruction can't be logged, it might even be a major inconvenience, but is it really so horrific a prospect that the program can't continue running? If not, you'll have to prevent the exception thrown by logDestruction from propagating out of Session's destructor. The only way to do that is by using try and catch blocks. A naive attempt might look like this,

Session::~Session()

{

try {

logDestruction(this);

}

catch (...) {

cerr << "Unable to log destruction of Session object "

<< "at address "

<< this

<< ".\n";

}

}

but this is probably no safer than our original code. If one of the calls to operator<< in the catch block results in an exception being thrown, we're back where we started, with an exception leaving the Session destructor. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp8) [Item M11, P8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp8)

We could always put a try block inside the catch block, but that seems a bit extreme. Instead, we'll just forget about logging Session destructions if logDestruction throws an exception:

Session::~Session()

{

try {

logDestruction(this);

}

catch (...) { }

}

The catch block appears to do nothing, but appearances can be deceiving. That block prevents exceptions thrown from logDestruction from propagating beyond Session's destructor. That's all it needs to do. We can now rest easy knowing that if a Session object is destroyed as part of stack unwinding, terminate will not be called.

There is a second reason why it's bad practice to allow exceptions to propagate out of destructors. If an exception is thrown from a destructor and is not caught there, that destructor won't run to completion. (It will stop at the point where the exception is thrown.) If the destructor doesn't run to completion, it won't do everything it's supposed to do. For example, consider a modified version of the Session class where the creation of a session starts a database transaction and the termination of a session ends that transaction: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp11) [Item M11, P11](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_DIR.HTM#dingp11)

Session::Session() // to keep things simple,

{ // this ctor handles no

// exceptions

logCreation(this);

startTransaction(); // start DB transaction

}

Session::~Session()

{

logDestruction(this);

endTransaction(); // end DB transaction

}

Here, if logDestruction throws an exception, the transaction started in the Session constructor will never be ended. In this case, we might be able to reorder the function calls in Session's destructor to eliminate the problem, but if endTransaction might throw an exception, we've no choice but to revert to try and catch blocks.

We thus find ourselves with two good reasons for keeping exceptions from propagating out of destructors. First, it prevents terminate from being called during the stack-unwinding part of exception propagation. Second, it helps ensure that destructors always accomplish everything they are supposed to accomplish. Each argument is convincing in its own right, but together, the case is ironclad. (If you're *still* not convinced, turn to [Herb Sutter's article](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MAGAZINE/SU_FRAME.HTM); in particular, to the section entitled, ["Destructors That Throw and Why They're Evil](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MAGAZINE/SU_FRAME.HTM#destruct).)

## Item 12: vs passing a parameter or calling a virtual function

The syntax for declaring function parameters is almost the same as that for catch clauses:

class Widget { ... }; // some class; it makes no

// difference what it is

void f1(Widget w); // all these functions

void f2(Widget& w); // take parameters of

void f3(const Widget& w); // type Widget, Widget&, or

void f4(Widget \*pw); // Widget\*

void f5(const Widget \*pw);

catch (Widget w) ... // all these catch clauses

catch (Widget& w) ... // catch exceptions of

catch (const Widget& w) ... // type Widget, Widget&, or

catch (Widget \*pw) ... // Widget\*

catch (const Widget \*pw) ...

You might therefore assume that passing an exception from a throw site to a catch clause is basically the same as passing an argument from a function call site to the function's parameter. There are some similarities, to be sure, but there are significant differences, too.

Let us begin with a similarity. You can pass both function parameters and exceptions by value, by reference, or by pointer. What *happens* when you pass parameters and exceptions, however, is quite different. This difference grows out of the fact that when you call a function, control eventually returns to the call site (unless the function fails to return), but when you throw an exception, control does *not* return to the throw site.

### Exception object is always copied

Consider a function that both passes a Widget as a parameter and throws a Widget as an exception:

// function to read the value of a Widget from a stream

istream operator>>(istream& s, Widget& w);

void passAndThrowWidget()

{

Widget localWidget;

cin >> localWidget; // pass localWidget to operator>>

throw localWidget; // throw localWidget as an exception

}

When localWidget is passed to operator>>, no copying is performed. Instead, the reference w inside operator>> is bound to localWidget, and anything done to w is really done to localWidget. It's a different story when localWidget is thrown as an exception. Regardless of whether the exception is caught by value or by reference (it can't be caught by pointer — that would be a type mismatch), a copy of localWidget will be made, and it is the *copy* that is passed to the catch clause. This must be the case, because localWidget will go out of scope once control leaves passAndThrowWidget, and when localWidget goes out of scope, its destructor will be called. If localWidget itself were passed to a catch clause, the clause would receive a destructed Widget, an ex-Widget, a former Widget, the carcass of what once was but is no longer a Widget. That would not be useful, and that's why C++ specifies that an object thrown as an exception is *always* copied.

### Hence, catch block cannot modify

This copying occurs even if the object being thrown is not in danger of being destroyed. For example, if passAndThrowWidget declares localWidget to be static, [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp7) [Item M12, P7](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp7)

void passAndThrowWidget()

{

static Widget localWidget; // this is now static; it

// will exist until the

// end of the program

cin >> localWidget; // this works as before

throw localWidget; // a copy of localWidget is

} // still made and thrown

a copy of localWidget would still be made when the exception was thrown. This means that even if the exception is caught by reference, it is not possible for the catch block to modify localWidget; it can only modify a *copy* of localWidget. This mandatory copying of exception objects helps explain another difference between parameter passing and throwing an exception: the latter is typically much slower than the former (see [Item 15](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_FR.HTM#40989)).

### Copy constructor on object’s *static type*

When an object is copied for use as an exception, the copying is performed by the object's copy constructor. This copy constructor is the one in the class corresponding to the object's [*static* type](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MIINTRFR.HTM#72671), not its [dynamic type](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MIINTRFR.HTM#72671). For example, consider this slightly modified version of passAndThrowWidget: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp9) [Item M12, P9](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp9)

class Widget { ... };

class SpecialWidget: public Widget { ... };

void passAndThrowWidget()

{

SpecialWidget localSpecialWidget;

...

Widget& rw = localSpecialWidget; // rw refers to a

// SpecialWidget

throw rw; // this throws an

// exception of type

} // Widget!

Here a Widget exception is thrown, even though rw refers to a SpecialWidget. That's because rw's static type is Widget, not SpecialWidget. That rw actually refers to a SpecialWidget is of no concern to your compilers; all they care about is rw's static type. This behavior may not be what you want, but it's consistent with all other cases in which C++ copies objects. Copying is always based on an object's static type (but see [Item 25](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI25_FR.HTM#5341) for a technique that lets you make copies on the basis of an object's dynamic type).

### Throw vs throw object

The fact that exceptions are copies of other objects has an impact on how you propagate exceptions from a catch block. Consider these two catch blocks, which at first glance appear to do the same thing: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp11) [Item M12, P11](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp11)

catch (Widget& w) // catch Widget exceptions

{

... // handle the exception

throw; // rethrow the exception so it

} // continues to propagate

catch (Widget& w) // catch Widget exceptions

{

... // handle the exception

throw w; // propagate a copy of the

} // caught exception

The only difference between these blocks is that the first one rethrows the current exception, while the second one throws a new copy of the current exception. Setting aside the performance cost of the additional copy operation, is there a difference between these approaches? [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp12) [Item M12, P12](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp12)

There is. The first block rethrows the *current* exception, regardless of its type. In particular, if the exception originally thrown was of type SpecialWidget, the first block would propagate a SpecialWidget exception, even though w's static type is Widget. This is because no copy is made when the exception is rethrown. The second catch block throws a *new* exception, which will always be of type Widget, because that's w's static type. In general, you'll want to use the

throw;

syntax to rethrow the current exception, because there's no chance that that will change the type of the exception being propagated. Furthermore, it's more efficient, because there's no need to generate a new exception object. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp14) [Item M12, P14](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp14)

(Incidentally, the copy made for an exception is a *temporary* object. As [Item 19](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI19_FR.HTM#41177) explains, this gives compilers the right to optimize it out of existence. I wouldn't expect your compilers to work that hard, however. Exceptions are supposed to be rare, so it makes little sense for compiler vendors to pour a lot of energy into their optimization.) [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp15) [Item M12, P15](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp15)

Let us examine the three kinds of catch clauses that could catch the Widget exception thrown by passAndThrowWidget. They are: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp16) [Item M12, P16](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp16)

catch (Widget w) ... // catch exception by value

catch (Widget& w) ... // catch exception by

// reference

catch (const Widget& w) ... // catch exception by

// reference-to-const

Right away we notice another difference between parameter passing and exception propagation. A thrown object (which, as explained above, is always a temporary) may be caught by simple reference; it need not be caught by reference-to-const. Passing a temporary object to a non-const reference parameter is not allowed for function calls (see [Item 19](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI19_FR.HTM#41177)), but it is for exceptions.

Let us overlook this difference, however, and return to our examination of copying exception objects. We know that when we pass a function argument by value, we make a copy of the passed object (see [Item E22](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI22_FR.HTM#6133)), and we store that copy in a function parameter. The same thing happens when we pass an exception by value. Thus, when we declare a catch clause like this,

catch (Widget w) ... // catch by value

we expect to pay for the creation of *two* copies of the thrown object, one to create the temporary that all exceptions generate, the second to copy that temporary into w. Similarly, when we catch an exception by reference, [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp19) [Item M12, P19](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp19)

catch (Widget& w) ... // catch by reference

catch (const Widget& w) ... // also catch by reference

we still expect to pay for the creation of a copy of the exception: the copy that is the temporary. In contrast, when we pass function parameters by reference, no copying takes place. When throwing an exception, then, we expect to construct (and later destruct) one more copy of the thrown object than if we passed the same object to a function. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp20) [Item M12, P20](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp20)

We have not yet discussed throwing exceptions by pointer, but throw by pointer is equivalent to pass by pointer. Either way, a copy of the *pointer* is passed. About all you need to remember is not to throw a pointer to a local object, because that local object will be destroyed when the exception leaves the local object's scope. The catch clause would then be initialized with a pointer to an object that had already been destroyed. This is the behavior the mandatory copying rule is designed to avoid. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp21) [Item M12, P21](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp21)

The way in which objects are moved from call or throw sites to parameters or catch clauses is one way in which argument passing differs from exception propagation. A second difference lies in what constitutes a type match between caller or thrower and callee or catcher. Consider the sqrt function from the standard math library: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp22) [Item M12, P22](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp22)

double sqrt(double); // from <cmath> or <math.h>

We can determine the square root of an integer like this: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp23) [Item M12, P23](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp23)

int i;

double sqrtOfi = sqrt(i);

There is nothing surprising here. The language allows implicit conversion from int to double, so in the call to sqrt, i is silently converted to a double, and the result of sqrt corresponds to that double. (See [Item 5](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI5_FR.HTM#5970) for a fuller discussion of implicit type conversions.) In general, such conversions are not applied when matching exceptions to catch clauses. In this code,

void f(int value)

{

try {

if (someFunction()) { // if someFunction() returns

throw value; // true, throw an int

...

}

}

catch (double d) { // handle exceptions of

... // type double here

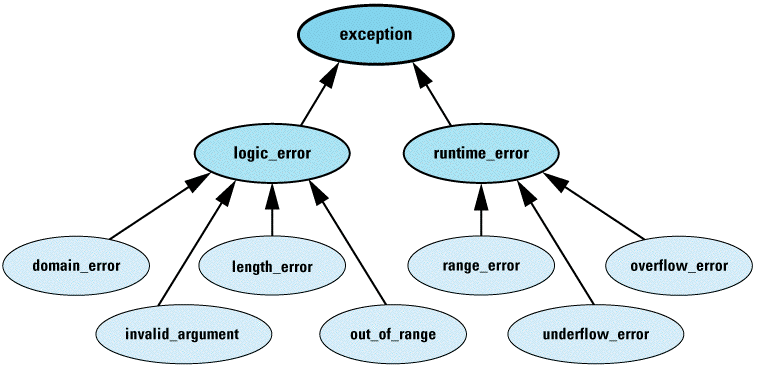
}

...

}

the int exception thrown inside the try block will never be caught by the catch clause that takes a double. That clause catches only exceptions that are exactly of type double; no type conversions are applied. As a result, if the int exception is to be caught, it will have to be by some other (dynamically enclosing) catch clause taking an int or an int& (possibly modified by const or volatile). [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp25) [Item M12, P25](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp25)

Two kinds of conversions *are* applied when matching exceptions to catch clauses. The first is inheritance-based conversions. A catch clause for base class exceptions is allowed to handle exceptions of derived class types, too. For example, consider the diagnostics portion of the hierarchy of exceptions defined by the standard C++ library (see [Item E49](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI49_FR.HTM#8392)): [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp26) [Item M12, P26](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp26)



A catch clause for runtime\_errors can catch exceptions of type range\_error and overflow\_error, too, and a catch clause accepting an object of the root class exception can catch any kind of exception derived from this hierarchy. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp27) [Item M12, P27](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp27)

This inheritance-based exception-conversion rule applies to values, references, and pointers in the usual fashion: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp28) [Item M12, P28](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp28)

catch (runtime\_error) ... // can catch errors of type

catch (runtime\_error&) ... // runtime\_error,

catch (const runtime\_error&) ... // range\_error, or

// overflow\_error

catch (runtime\_error\*) ... // can catch errors of type

catch (const runtime\_error\*) ... // runtime\_error\*,

// range\_error\*, or

// overflow\_error\*

The second type of allowed conversion is from a typed to an untyped pointer, so a catch clause taking a const void\* pointer will catch an exception of any pointer type: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp29) [Item M12, P29](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp29)

catch (const void\*) ... // catches any exception

// that's a pointer

The final difference between passing a parameter and propagating an exception is that catch clauses are always tried *in the order of their appearance*. Hence, it is possible for an exception of a derived class type to be handled by a catch clause for one of its base class types — even when a catch clause for the derived class is associated with the same try block! For example,

try {

...

}

catch (logic\_error& ex) { // this block will catch

... // *all* logic\_error

} // exceptions, even those

// of derived types

catch (invalid\_argument& ex) { // this block can *never* be

... // executed, because all

} // invalid\_argument

// exceptions will be caught

// by the clause above

Contrast this behavior with what happens when you call a virtual function. When you call a virtual function, the function invoked is the one in the class *closest* to the dynamic type of the object invoking the function. You might say that virtual functions employ a "best fit" algorithm, while exception handling follows a "first fit" strategy. Compilers may warn you if a catch clause for a derived class comes after one for a base class (some issue an error, because such code used to be illegal in C++), but your best course of action is preemptive: never put a catch clause for a base class before a catch clause for a derived class. The code above, for example, should be reordered like this: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp31) [Item M12, P31](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_DIR.HTM#dingp31)

try {

...

}

catch (invalid\_argument& ex) { // handle invalid\_argument

... // exceptions here

}

catch (logic\_error& ex) { // handle all other

... // logic\_errors here

}

There are thus three primary ways in which passing an object to a function or using that object to invoke a virtual function differs from throwing the object as an exception. First, exception objects are always copied; when caught by value, they are copied twice. Objects passed to function parameters need not be copied at all. Second, objects thrown as exceptions are subject to fewer forms of type conversion than are objects passed to functions. Finally, catch clauses are examined in the order in which they appear in the source code, and the first one that can succeed is selected for execution. When an object is used to invoke a virtual function, the function selected is the one that provides the *best* match for the type of the object, even if it's not the first one listed in the source code

## Item 13: Catch exceptions by reference

When you write a catch clause, you must specify how exception objects are to be passed to that clause. You have three choices, just as when specifying how parameters should be passed to functions: by pointer, by value, or by reference. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp2) [Item M13, P2](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp2)

### By pointer

Let us consider first catch by pointer. In theory, this should be the least inefficient way to implement the invariably slow process of moving an exception from throw site to catch clause (see [Item 15](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_FR.HTM#40989)). That's because throw by pointer is the only way of moving exception information without copying an object (see [Item 12](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_FR.HTM#76790)). For example: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp3) [Item M13, P3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp3)

class exception { ... }; // from the standard C++

// library exception

// hierarchy (see [Item 12](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_FR.HTM#76790))

void someFunction()

{

static exception ex; // exception object

...

throw &ex; // throw a pointer to ex

...

}

void doSomething()

{

try {

someFunction(); // may throw an exception\*

}

catch (exception \*ex) { // catches the exception\*;

... // no object is copied

}

}

This looks neat and tidy, but it's not quite as well-kept as it appears. For this to work, programmers must define exception objects in a way that guarantees the objects exist after control leaves the functions throwing pointers to them. Global and static objects work fine, but it's easy for programmers to forget the constraint. If they do, they typically end up writing code like this: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp4) [Item M13, P4](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp4)

void someFunction()

{

exception ex; // local exception object;

// will be destroyed when

// this function's scope is

... // exited

throw &ex; // throw a pointer to an

... // object that's about to

} // be destroyed

This is worse than useless, because the catch clause handling this exception receives a pointer to an object that no longer exists. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp5) [Item M13, P5](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp5)

An alternative is to throw a pointer to a new heap object: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp6) [Item M13, P6](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp6)

void someFunction()

{

...

throw new exception; // throw a pointer to a new heap-

... // based object (and hope that

} // operator new — see [Item 8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI8_FR.HTM#33985) —

// doesn't itself throw an

// exception!)

This avoids the I-just-caught-a-pointer-to-a-destroyed-object problem, but now authors of catch clauses confront a nasty question: should they delete the pointer they receive? If the exception object was allocated on the heap, they must, otherwise they suffer a resource leak. If the exception object wasn't allocated on the heap, they mustn't, otherwise they suffer undefined program behavior. What to do? [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp7) [Item M13, P7](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp7)

It's impossible to know. Some clients might pass the address of a global or static object, others might pass the address of an exception on the heap. Catch by pointer thus gives rise to the Hamlet conundrum: to delete or not to delete? It's a question with no good answer. You're best off ducking it. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp8) [Item M13, P8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp8)

Furthermore, catch-by-pointer runs contrary to the convention established by the language itself. The four standard exceptions — bad\_alloc (thrown when operator new (see [Item 8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI8_FR.HTM#33985)) can't satisfy a memory request), bad\_cast (thrown when a dynamic\_cast to a reference fails; see [Item 2](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI2_FR.HTM#77216)), bad\_typeid (thrown when dynamic\_cast is applied to a null pointer), and bad\_exception (available for unexpected exceptions; see [Item 14](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_FR.HTM#6011)) — are all objects, not pointers to objects, so you have to catch them by value or by reference, anyway. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp9) [Item M13, P9](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp9)

### By value

Catch-by-value eliminates questions about exception deletion and works with the standard exception types. However, it requires that exception objects be copied *twice* each time they're thrown (see [Item 12](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI12_FR.HTM#76790)). It also gives rise to the specter of the *slicing problem*, whereby derived class exception objects caught as base class exceptions have their derivedness "sliced off." Such "sliced" objects *are* base class objects: they lack derived class data members, and when virtual functions are called on them, they resolve to virtual functions of the base class. (Exactly the same thing happens when an object is passed to a function by value — see [Item E22](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI22_FR.HTM#6133).) For example, consider an application employing an exception class hierarchy that extends the standard one:

class exception { // as above, this is a

public: // standard exception class

virtual const char \* what() throw();

// returns a brief descrip.

... // of the exception (see

// [Item 14](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_FR.HTM#6011) for info about

}; // the "throw()" at the

// end of the declaration)

class runtime\_error: // also from the standard

public exception { ... }; // C++ exception hierarchy

class Validation\_error: // this is a class added by

public runtime\_error { // a client

public:

virtual const char \* what() throw();

// this is a redefinition

... // of the function declared

}; // in class exception above

void someFunction() // may throw a validation

{ // exception

...

if (*a validation test fails*) {

throw Validation\_error();

}

...

}

void doSomething()

{

try {

someFunction(); // may throw a validation

} // exception

catch (exception ex) { // catches all exceptions

// in or derived from

// the standard hierarchy

cerr << ex.what(); // calls exception::what(),

... // never

} // Validation\_error::what()

}

The version of what that is called is that of the base class, even though the thrown exception is of type Validation\_error and Validation\_error redefines that virtual function. This kind of slicing behavior is almost never what you want. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp11) [Item M13, P11](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp11)

### By reference

That leaves only catch-by-reference. Catch-by-reference suffers from none of the problems we have discussed. Unlike catch-by-pointer, the question of object deletion fails to arise, and there is no difficulty in catching the standard exception types. Unlike catch-by-value, there is no slicing problem, and exception objects are copied only once. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp12) [Item M13, P12](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp12)

If we rewrite the last example using catch-by-reference, it looks like this:

void someFunction() // nothing changes in this

{ // function

...

if (*a validation test fails*) {

throw Validation\_error();

}

...

}

void doSomething()

{

try {

someFunction(); // no change here

}

catch (exception& ex) { // here we catch by reference

// instead of by value

cerr << ex.what(); // now calls

// Validation\_error::what(),

... // not exception::what()

}

}

There is no change at the throw site, and the only change in the catch clause is the addition of an ampersand. This tiny modification makes a big difference, however, because virtual functions in the catch block now work as we expect: functions in Validation\_error are invoked if they redefine those in exception. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp14) [Item M13, P14](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI13_DIR.HTM#dingp14)

What a happy confluence of events! If you catch by reference, you sidestep questions about object deletion that leave you damned if you do and damned if you don't; you avoid slicing exception objects; you retain the ability to catch standard exceptions; and you limit the number of times exception objects need to be copied. So what are you waiting for? Catch exceptions by reference!

## Item 14: Use exceptions specification judiciously

There's no denying it: exception specifications have appeal. They make code easier to understand, because they explicitly state what exceptions a function may throw. But they're more than just fancy comments. Compilers are sometimes able to detect inconsistent exception specifications during compilation. Furthermore, if a function throws an exception not listed in its exception specification, that fault is detected at runtime, and the special function unexpected is automatically invoked. Both as a documentation aid and as an enforcement mechanism for constraints on exception usage, then, exception specifications seem attractive. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp2) [Item M14, P2](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp2)

### Problem with default behavior

As is often the case, however, beauty is only skin deep. The default behavior for unexpected is to call terminate, and the default behavior for terminate is to call abort, so the default behavior for a program with a violated exception specification is to halt. Local variables in active stack frames are not destroyed, because abort shuts down program execution without performing such cleanup. A violated exception specification is therefore a cataclysmic thing, something that should almost never happen. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp3) [Item M14, P3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp3)

Unfortunately, it's easy to write functions that make this terrible thing occur. Compilers only *partially* check exception usage for consistency with exception specifications. What they do not check for — what the **°**[language standard](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=cstandard) *prohibits* them from rejecting (though they may issue a warning) — is a call to a function that *might* violate the exception specification of the function making the call. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp4) [Item M14, P4](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp4)

Consider a declaration for a function f1 that has no exception specification. Such a function may throw any kind of exception: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp5) [Item M14, P5](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp5)

extern void f1(); // might throw anything

Now consider a function f2 that claims, through its exception specification, it will throw only exceptions of type int: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp6) [Item M14, P6](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp6)

void f2() throw(int);

It is perfectly legal C++ for f2 to call f1, even though f1 might throw an exception that would violate f2's exception specification: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp7) [Item M14, P7](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp7)

void f2() throw(int)

{

...

f1(); // legal even though f1 might throw

// something besides an int

...

}

This kind of flexibility is essential if new code with exception specifications is to be integrated with older code lacking such specifications. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp8) [Item M14, P8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp8)

Because your compilers are content to let you call functions whose exception specifications are inconsistent with those of the routine containing the calls, and because such calls might result in your program's execution being terminated, it's important to write your software in such a way that these kinds of inconsistencies are minimized.

A good way to start is to avoid putting exception specifications on templates that take type arguments. Consider this template, which certainly looks as if it couldn't throw any exceptions:

// a poorly designed template wrt exception specifications

template<class T>

bool operator==(const T& lhs, const T& rhs) throw()

{

return &lhs == &rhs;

}

This template defines an operator== function for all types. For any pair of objects of the same type, it returns true if the objects have the same address, otherwise it returns false.

This template contains an exception specification stating that the functions generated from the template will throw no exceptions. But that's not necessarily true, because it's possible that operator& (the address-of operator — see [Item E45](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI45_FR.HTM#8160)) has been overloaded for some types. If it has, operator& may throw an exception when called from inside operator==. If it does, our exception specification is violated, and off to unexpected we go. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp11) [Item M14, P11](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp11)

This is a specific example of a more general problem, namely, that there is no way to know *anything* about the exceptions thrown by a template's type parameters. We can almost never provide a meaningful exception specification for a template, because templates almost invariably use their type parameter in some way. The conclusion? Templates and exception specifications don't mix.

A second technique you can use to avoid calls to unexpected is to omit exception specifications on functions making calls to functions that themselves lack exception specifications. This is simple common sense, but there is one case that is easy to forget. That's when allowing users to register callback functions: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp13) [Item M14, P13](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp13)

// Function pointer type for a window system callback

// when a window system event occurs

typedef void (\*CallBackPtr)(int eventXLocation,

int eventYLocation,

void \*dataToPassBack);

// Window system class for holding onto callback

// functions registered by window system clients

class CallBack {

public:

CallBack(CallBackPtr fPtr, void \*dataToPassBack)

: func(fPtr), data(dataToPassBack) {}

void makeCallBack(int eventXLocation,

int eventYLocation) const throw();

private:

CallBackPtr func; // function to call when

// callback is made

void \*data; // data to pass to callback

}; // function

// To implement the callback, we call the registered func-

// tion with event's coordinates and the registered data

void CallBack::makeCallBack(int eventXLocation,

int eventYLocation) const throw()

{

func(eventXLocation, eventYLocation, data);

}

Here the call to func in makeCallBack runs the risk of a violated exception specification, because there is no way of knowing what exceptions func might throw. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp14) [Item M14, P14](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp14)

This problem can be eliminated by tightening the exception specification in the CallBackPtr typedef:[5](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14.HTM#9602) [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp15) [Item M14, P15](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp15)

typedef void (\*CallBackPtr)(int eventXLocation,

int eventYLocation,

void \*dataToPassBack) throw();

Given this typedef, it is now an error to register a callback function that fails to guarantee it throws nothing: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp16) [Item M14, P16](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp16)

// a callback function without an exception specification

void callBackFcn1(int eventXLocation, int eventYLocation,

void \*dataToPassBack);

void \*callBackData;

...

CallBack c1(callBackFcn1, callBackData);

// error! callBackFcn1

// might throw an exception

// a callback function with an exception specification

void callBackFcn2(int eventXLocation,

int eventYLocation,

void \*dataToPassBack) throw();

CallBack c2(callBackFcn2, callBackData);

// okay, callBackFcn2 has a

// conforming ex. spec.

This checking of exception specifications when passing function pointers is a relatively recent addition to the language, so don't be surprised if your compilers don't yet support it. If they don't, it's up to you to ensure you don't make this kind of mistake. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp17) [Item M14, P17](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp17)

A third technique you can use to avoid calls to unexpected is to handle exceptions "the system" may throw. Of these exceptions, the most common is bad\_alloc, which is thrown by operator new and operator new[] when a memory allocation fails (see [Item 8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI8_FR.HTM#33985)). If you use the new operator (again, see [Item 8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI8_FR.HTM#33985)) in any function, you must be prepared for the possibility that the function will encounter a bad\_alloc exception. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp18) [Item M14, P18](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp18)

Now, an ounce of prevention may be better than a pound of cure, but sometimes prevention is hard and cure is easy. That is, sometimes it's easier to cope with unexpected exceptions directly than to prevent them from arising in the first place. If, for example, you're writing software that uses exception specifications rigorously, but you're forced to call functions in libraries that don't use exception specifications, it's impractical to prevent unexpected exceptions from arising, because that would require changing the code in the libraries. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp19) [Item M14, P19](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp19)

If preventing unexpected exceptions isn't practical, you can exploit the fact that C++ allows you to replace unexpected exceptions with exceptions of a different type. For example, suppose you'd like all unexpected exceptions to be replaced by UnexpectedException objects. You can set it up like this, [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp20) [Item M14, P20](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp20)

class UnexpectedException {}; // all unexpected exception

// objects will be replaced

// by objects of this type

void convertUnexpected() // function to call if

{ // an unexpected exception

throw UnexpectedException(); // is thrown

}

and make it happen by replacing the default unexpected function with convertUnexpected: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp21) [Item M14, P21](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp21)

set\_unexpected(convertUnexpected);

Once you've done this, any unexpected exception results in convertUnexpected being called. The unexpected exception is then replaced by a new exception of type UnexpectedException. Provided the exception specification that was violated includes UnexpectedException, exception propagation will then continue as if the exception specification had always been satisfied. (If the exception specification does not include UnexpectedException, terminate will be called, just as if you had never replaced unexpected.) [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp22) [Item M14, P22](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp22)

Another way to translate unexpected exceptions into a well known type is to rely on the fact that if the unexpected function's replacement rethrows the current exception, that exception will be replaced by a new exception of the standard type bad\_exception. Here's how you'd arrange for that to happen: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp23) [Item M14, P23](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp23)

void convertUnexpected() // function to call if

{ // an unexpected exception

throw; // is thrown; just rethrow

} // the current exception

set\_unexpected(convertUnexpected);

// install convertUnexpected

// as the unexpected

// replacement

If you do this and you include bad\_exception (or its base class, the standard class exception) in all your exception specifications, you'll never have to worry about your program halting if an unexpected exception is encountered. Instead, any wayward exception will be replaced by a bad\_exception, and that exception will be propagated in the stead of the original one.

By now you understand that exception specifications can be a lot of trouble. Compilers perform only partial checks for their consistent usage, they're problematic in templates, they're easy to violate inadvertently, and, by default, they lead to abrupt program termination when they're violated. Exception specifications have another drawback, too, and that's that they result in unexpected being invoked even when a higher-level caller is prepared to cope with the exception that's arisen. For example, consider this code, which is taken almost verbatim from [Item 11](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI11_FR.HTM#39749):

class Session { // for modeling online

public: // sessions

~Session();

...

private:

static void logDestruction(Session \*objAddr) throw();

};

Session::~Session()

{

try {

logDestruction(this);

}

catch (...) { }

}

The Session destructor calls logDestruction to record the fact that a Session object is being destroyed, but it explicitly catches any exceptions that might be thrown by logDestruction. However, logDestruction comes with an exception specification asserting that it throws no exceptions. Now, suppose some function called by logDestruction throws an exception that logDestruction fails to catch. This isn't supposed to happen, but as we've seen, it isn't difficult to write code that leads to the violation of exception specifications. When this unanticipated exception propagates through logDestruction, unexpected will be called, and, by default, that will result in termination of the program. This is correct behavior, to be sure, but is it the behavior the author of Session's destructor wanted? That author took pains to handle *all possible* exceptions, so it seems almost unfair to halt the program without giving Session's destructor's catch block a chance to work. If logDestruction had no exception specification, this I'm-willing-to-catch-it-if-you'll-just-give-me-a-chance scenario would never arise. (One way to prevent it is to replace unexpected as described above.) [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp26) [Item M14, P26](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI14_DIR.HTM#dingp26)

It's important to keep a balanced view of exception specifications. They provide excellent documentation on the kinds of exceptions a function is expected to throw, and for situations in which violating an exception specification is so dire as to justify immediate program termination, they offer that behavior by default. At the same time, they are only partly checked by compilers and they are easy to violate inadvertently. Furthermore, they can prevent high-level exception handlers from dealing with unexpected exceptions, even when they know how to. That being the case, exception specifications are a tool to be applied judiciously. Before adding them to your functions, consider whether the behavior they impart to your software is really the behavior you want.

## Item 15: Costs of exception

To handle exceptions at runtime, programs must do a fair amount of bookkeeping. At each point during execution, they must be able to identify the objects that require destruction if an exception is thrown; they must make note of each entry to and exit from a try block; and for each try block, they must keep track of the associated catch clauses and the types of exceptions those clauses can handle. This bookkeeping is not free. Nor are the runtime comparisons necessary to ensure that exception specifications are satisfied. Nor is the work expended to destroy the appropriate objects and find the correct catch clause when an exception is thrown. No, exception handling has costs, and you pay at least some of them even if you never use the keywords try, throw, or catch. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp2) [Item M15, P2](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp2)

Let us begin with the things you pay for even if you never use any exception-handling features. You pay for the space used by the data structures needed to keep track of which objects are fully constructed (see [Item 10](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI10_FR.HTM#38223)), and you pay for the time needed to keep these data structures up to date. These costs are typically quite modest. Nevertheless, programs compiled without support for exceptions are typically both faster and smaller than their counterparts compiled with support for exceptions. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp3) [Item M15, P3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp3)

In theory, you don't have a choice about these costs: exceptions are part of C++, compilers have to support them, and that's that. You can't even expect compiler vendors to eliminate the costs if you use no exception-handling features, because programs are typically composed of multiple independently generated object files, and just because one object file doesn't do anything with exceptions doesn't mean others don't. Furthermore, even if none of the object files linked to form an executable use exceptions, what about the libraries they're linked with? If *any part* of a program uses exceptions, the rest of the program must support them, too. Otherwise it may not be possible to provide correct exception-handling behavior at runtime. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp4) [Item M15, P4](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp4)

That's the theory. In practice, most vendors who support exception handling allow you to control whether support for exceptions is included in the code they generate. If you know that no part of your program uses try, throw, or catch, and you also know that no library with which you'll link uses try, throw, or catch, you might as well compile without exception-handling support and save yourself the size and speed penalty you'd otherwise probably be assessed for a feature you're not using. As time goes on and libraries employing exceptions become more common, this strategy will become less tenable, but given the current state of C++ software development, compiling without support for exceptions is a reasonable performance optimization if you have already decided not to use exceptions. It may also be an attractive optimization for libraries that eschew exceptions, provided they can guarantee that exceptions thrown from client code never propagate into the library. This is a difficult guarantee to make, as it precludes client redefinitions of library-declared virtual functions; it also rules out client-defined callback functions. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp5) [Item M15, P5](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp5)

A second cost of exception-handling arises from try blocks, and you pay it whenever you use one, i.e., whenever you decide you want to be able to catch exceptions. Different compilers implement try blocks in different ways, so the cost varies from compiler to compiler. As a rough estimate, expect your overall code size to increase by 5-10% and your runtime to go up by a similar amount if you use try blocks. This assumes no exceptions are thrown; what we're discussing here is just the cost of *having* try blocks in your programs. To minimize this cost, you should avoid unnecessary try blocks. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp6) [Item M15, P6](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp6)

Compilers tend to generate code for exception specifications much as they do for try blocks, so an exception specification generally incurs about the same cost as a try block. Excuse me? You say you thought exception specifications were just specifications, you didn't think they generated code? Well, now you have something new to think about. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp7) [Item M15, P7](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp7)

Which brings us to the heart of the matter, the cost of throwing an exception. In truth, this shouldn't be much of a concern, because exceptions should be rare. After all, they indicate the occurrence of events that are *exceptional*. The 80-20 rule (see [Item 16](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI16_FR.HTM#40995)) tells us that such events should almost never have much impact on a program's overall performance. Nevertheless, I know you're curious about just how big a hit you'll take if you throw an exception, and the answer is it's probably a big one. Compared to a normal function return, returning from a function by throwing an exception may be as much as *three orders of magnitude* slower. That's quite a hit. But you'll take it only if you throw an exception, and that should be almost never. If, however, you've been thinking of using exceptions to indicate relatively common conditions like the completion of a data structure traversal or the termination of a loop, now would be an excellent time to think again. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp8) [Item M15, P8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp8)

But wait. How can I know this stuff? If support for exceptions is a relatively recent addition to most compilers (it is), and if different compilers implement their support in different ways (they do), how can I say that a program's size will generally grow by about 5-10%, its speed will decrease by a similar amount, and it may run orders of magnitude slower if lots of exceptions are thrown? The answer is frightening: a little rumor and a handful of benchmarks (see [Item 23](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI23_FR.HTM#41253)). The fact is that most people — including most compiler vendors — have little experience with exceptions, so though we know there are costs associated with them, it is difficult to predict those costs accurately. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp9) [Item M15, P9](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI15_DIR.HTM#dingp9)

The prudent course of action is to be aware of the costs described in this item, but not to take the numbers very seriously. Whatever the cost of exception handling, you don't want to pay any more than you have to. To minimize your exception-related costs, compile without support for exceptions when that is feasible; limit your use of try blocks and exception specifications to those locations where you honestly need them; and throw exceptions only under conditions that are truly exceptional. If you still have performance problems, profile your software (see [Item 16](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI16_FR.HTM#40995)) to determine if exception support is a contributing factor. If it is, consider switching to different compilers, ones that provide more efficient implementations of C++'s exception-handling features

## <http://en.wikipedia.org/wiki/Return_value_optimization>

## http://www.informit.com/articles/article.aspx?p=25033

## Item 20:  Facilitate the return value optimization. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp1) [Item M20, P1](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp1)

A function that returns an object is frustrating to efficiency aficionados, because the by-value return, including the constructor and destructor calls it implies (see [Item 19](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI19_FR.HTM#41177)), cannot be eliminated. The problem is simple: a function either has to return an object in order to offer correct behavior or it doesn't. If it does, there's no way to get rid of the object being returned. Period. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp2) [Item M20, P2](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp2)

Consider the operator\* function for rational numbers: [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp3) [Item M20, P3](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp3)

class Rational {

public:

Rational(int numerator = 0, int denominator = 1);

...

int numerator() const;

int denominator() const;

};

// For an explanation of why the return value is const,

// see [Item 6](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI6_FR.HTM#5262)

const Rational operator\*(const Rational& lhs,

const Rational& rhs);

Without even looking at the code for operator\*, we know it must return an object, because it returns the product of two arbitrary numbers. These are *arbitrary* numbers. How can operator\* possibly avoid creating a new object to hold their product? It can't, so it must create a new object and return it. C++ programmers have nevertheless expended Herculean efforts in a search for the legendary elimination of the by-value return (see Items [E23](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI23_FR.HTM#6210) and [E31](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI31_FR.HTM#6650)). [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp4) [Item M20, P4](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp4)

Sometimes people return pointers, which leads to this syntactic travesty: [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp5) [Item M20, P5](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp5)

// an unreasonable way to avoid returning an object

const Rational \* operator\*(const Rational& lhs,

const Rational& rhs);

Rational a = 10;

Rational b(1, 2);

Rational c = \*(a \* b); // Does this look "natural"

// to you?

It also raises a question. Should the caller delete the pointer returned by the function? The answer is usually yes, and that usually leads to resource leaks. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp6) [Item M20, P6](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp6)

Other developers return references. That yields an acceptable syntax, [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp7) [Item M20, P7](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp7)

// a dangerous (and incorrect) way to avoid returning

// an object

const Rational& operator\*(const Rational& lhs,

const Rational& rhs);

Rational a = 10;

Rational b(1, 2);

Rational c = a \* b; // looks perfectly reasonable

but such functions can't be implemented in a way that behaves correctly. A common attempt looks like this: [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp8) [Item M20, P8](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp8)

// another dangerous (and incorrect) way to avoid

// returning an object

const Rational& operator\*(const Rational& lhs,

const Rational& rhs)

{

Rational result(lhs.numerator() \* rhs.numerator(),

lhs.denominator() \* rhs.denominator());

return result;

}

This function returns a reference to an object that no longer exists. In particular, it returns a reference to the local object result, but result is automatically destroyed when operator\* is exited. Returning a reference to an object that's been destroyed is hardly useful. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp9) [Item M20, P9](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp9)

Trust me on this: some functions (operator\* among them) just have to return objects. That's the way it is. Don't fight it. You can't win. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp10) [Item M20, P10](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp10)

That is, you can't win in your effort to eliminate by-value returns from functions that require them. But that's the wrong war to wage. From an efficiency point of view, you shouldn't care that a function returns an object, you should only care about the *cost* of that object. What you need to do is channel your efforts into finding a way to reduce the cost of returned objects, not to eliminate the objects themselves (which we now recognize is a futile quest). If no cost is associated with such objects, who cares how many get created? [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp11) [Item M20, P11](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp11)

It is frequently possible to write functions that return objects in such a way that compilers can eliminate the cost of the temporaries. The trick is to return *constructor arguments* instead of objects, and you can do it like this: [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp12) [Item M20, P12](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp12)

// an efficient and correct way to implement a

// function that returns an object

const Rational operator\*(const Rational& lhs,

const Rational& rhs)

{

return Rational(lhs.numerator() \* rhs.numerator(),

lhs.denominator() \* rhs.denominator());

}

Look closely at the expression being returned. It looks like you're calling a Rational constructor, and in fact you are. You're creating a temporary Rational object through this expression, [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp13) [Item M20, P13](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp13)

Rational(lhs.numerator() \* rhs.numerator(),

lhs.denominator() \* rhs.denominator());

and it is this temporary object the function is copying for its return value. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp14) [Item M20, P14](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp14)

This business of returning constructor arguments instead of local objects doesn't appear to have bought you a lot, because you still have to pay for the construction and destruction of the temporary created inside the function, and you still have to pay for the construction and destruction of the object the function returns. But you have gained something. The rules for C++ allow compilers to optimize temporary objects out of existence. As a result, if you call operator\* in a context like this, [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp15) [Item M20, P15](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp15)

Rational a = 10;

Rational b(1, 2);

Rational c = a \* b; // operator\* is called here

your compilers are allowed to eliminate both the temporary inside operator\* *and* the temporary returned by operator\*. They can construct the object defined by the return expression *inside the memory allotted for the object c*. If your compilers do this, the total cost of temporary objects as a result of your calling operator\* is zero: no temporaries are created. Instead, you pay for only one constructor call — the one to create c. Furthermore, you can't do any better than this, because c is a named object, and named objects can't be eliminated (see also [Item 22](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI22_FR.HTM#41251)).[7](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20.HTM#10118) You can, however, eliminate the overhead of the call to operator\* by declaring that function inline (but first see [Item E33](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\EC\EI33_FR.HTM#6729)): [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp16) [Item M20, P16](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp16)

// the most efficient way to write a function returning

// an object

inline const Rational operator\*(const Rational& lhs,

const Rational& rhs)

{

return Rational(lhs.numerator() \* rhs.numerator(),

lhs.denominator() \* rhs.denominator());

}

"Yeah, yeah," you mutter, "optimization, schmoptimization. Who cares what compilers *can* do? I want to know what they *do* do. Does any of this nonsense work with real compilers?" It does. This particular optimization — eliminating a local temporary by using a function's return location (and possibly replacing that with an object at the function's call site) — is both well-known and commonly implemented. It even has a name: the *return value optimization*. In fact, the existence of a name for this optimization may explain why it's so widely available. Programmers looking for a C++ compiler can ask vendors whether the return value optimization is implemented. If one vendor says yes and another says "The what?," the first vendor has a notable competitive advantage. Ah, capitalism. Sometimes you just gotta love it. [¤](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp17) [Item M](file:///J:\Dropbox\~Family\z-lei\~zNotes\1_Effective%20C++%202nd\MEC\MI20_DIR.HTM#dingp17)

**Item 5:  Use the same form in corresponding uses of new and delete.**  ¤[Item E5, P1](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp1)

What's wrong with this picture?  ¤[Item E5, P2](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp2)

string \*stringArray = new string[100];

...

delete stringArray;

Everything here appears to be in order — the use of new is matched with a use of delete — but something is still quite wrong: your program's behavior is undefined. At the very least, 99 of the 100 string objects pointed to by stringArray are unlikely to be properly destroyed, because their destructors will probably never be called.  ¤[Item E5, P3](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp3)

When you use new, two things happen. First, memory is allocated (via the function operator new, about which I'll have more to say in Items [7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_FR.HTM#1894)-[10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_FR.HTM#1986) as well as [Item M8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI8_FR.HTM#33985)). Second, one or more constructors are called for that memory. When you use delete, two other things happen: one or more destructors are called for the memory, then the memory is deallocated (via the function operator delete — see Items [8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851) and [M8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI8_FR.HTM#33985)). The big question for delete is this: *how many* objects reside in the memory being deleted? The answer to that determines how many destructors must be called.  ¤[Item E5, P4](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp4)

Actually, the question is simpler: does the pointer being deleted point to a single object or to an array of objects? The only way for delete to know is for you to tell it. If you don't use brackets in your use of delete, delete assumes a single object is pointed to. Otherwise, it assumes that an array is pointed to:  ¤[Item E5, P5](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp5)

string \*stringPtr1 = new string;

string \*stringPtr2 = new string[100];

...

delete stringPtr1; // delete an object

delete [] stringPtr2; // delete an array of

// objects

What would happen if you used the "[]" form on stringPtr1? The result is undefined. What would happen if you didn't use the "[]" form on stringPtr2? Well, that's undefined too. Furthermore, it's undefined even for built-in types like ints, even though such types lack destructors. The rule, then, is simple: if you use [] when you call new, you must use [] when you call delete. If you don't use [] when you call new, don't use [] when you call delete.  ¤[Item E5, P6](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp6)

This is a particularly important rule to bear in mind when you are writing a class containing a pointer data member and also offering multiple constructors, because then you've got to be careful to use the *same form* of new in all the constructors to initialize the pointer member. If you don't, how will you know what form of delete to use in your destructor? For a further examination of this issue, see [Item 11](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI11_FR.HTM#2042).  ¤[Item E5, P7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp7)

This rule is also important for the typedef-inclined, because it means that a typedef's author must document which form of delete should be employed when new is used to conjure up objects of the typedef type. For example, consider this typedef:  ¤[Item E5, P8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp8)

typedef string AddressLines[4]; // a person's address

// has 4 lines, each of

// which is a string

Because AddressLines is an array, this use of new,  ¤[Item E5, P9](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp9)

string \*pal = new AddressLines; // note that "new

// AddressLines" returns

// a string\*, just like

// "new string[4]" would

must be matched with the *array* form of delete:  ¤[Item E5, P10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp10)

delete pal; // undefined!

delete [] pal; // fine

To avoid such confusion, you're probably best off abstaining from typedefs for array types. That should be easy, however, because the standard C++ library (see [Item 49](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI49_FR.HTM#8392)) includes string and vector templates that reduce the need for built-in arrays to nearly zero. Here, for example, AddressLines could be defined to be a vector of strings. That is, AddressLines could be of type vector<string>.  ¤[Item E5, P11](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI5_DIR.HTM#dingp11)

**Item 6:  Use delete on pointer members in destructors.**  ¤[Item E6, P1](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp1)

Most of the time, classes performing dynamic memory allocation will use new in the constructor(s) to allocate the memory and will later use delete in the destructor to free up the memory. This isn't too difficult to get right when you first write the class, provided, of course, that you remember to employ delete on *all* the members that could have been assigned memory in *any* constructor.  ¤[Item E6, P2](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp2)

However, the situation becomes more difficult as classes are maintained and enhanced, because the programmers making the modifications to the class may not be the ones who wrote the class in the first place. Under those conditions, it's easy to forget that adding a pointer member almost always requires each of the following:  ¤[Item E6, P3](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp3)

* Initialization of the pointer in each of the constructors. If no memory is to be allocated to the pointer in a particular constructor, the pointer should be initialized to 0 (i.e., the null pointer).  ¤[Item E6, P4](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp4)
* Deletion of the existing memory and assignment of new memory in the assignment operator. (See also [Item 17](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI17_FR.HTM#2264).)  ¤[Item E6, P5](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp5)
* Deletion of the pointer in the destructor.  ¤[Item E6, P6](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp6)

If you forget to initialize a pointer in a constructor, or if you forget to handle it inside the assignment operator, the problem usually becomes apparent fairly quickly, so in practice those issues don't tend to plague you. Failing to delete the pointer in the destructor, however, often exhibits no obvious external symptoms. Instead, it manifests itself as a subtle memory leak, a slowly growing cancer that will eventually devour your address space and drive your program to an early demise. Because this particular problem doesn't usually call attention to itself, it's important that you keep it in mind whenever you add a pointer member to a class.  ¤[Item E6, P7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp7)

Note, by the way, that deleting a null pointer is always safe (it does nothing). Thus, if you write your constructors, your assignment operators, and your other member functions such that each pointer member of the class is always either pointing to valid memory or is null, you can merrily delete away in the destructor without regard for whether you ever used new for the pointer in question.  ¤[Item E6, P8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp8)

There's no reason to get fascist about this Item. For example, you certainly don't want to use delete on a pointer that wasn't initialized via new, and, except in the case of smart pointer objects (see [Item M28](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI28_FR.HTM#61766)), you almost *never* want to delete a pointer that was passed to you in the first place. In other words, your class destructor usually shouldn't be using delete unless your class members were the ones who used new in the first place.  ¤[Item E6, P9](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp9)

Speaking of smart pointers, one way to avoid the need to delete pointer members is to replace those members with smart pointer objects like the standard C++ Library's auto\_ptr. To see how this can work, take a look at Items [M9](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI9_FR.HTM#5292) and [M10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI10_FR.HTM#38223).  ¤[Item E6, P10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI6_DIR.HTM#dingp10)

**Item 7:  Be prepared for out-of-memory conditions.**  ¤[Item E7, P1](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp1)

When operator new can't allocate the memory you request, it throws an exception. (It used to return 0, and some older compilers still do that. You can make your compilers do it again if you want to, but I'll defer that discussion until the end of this Item.) Deep in your heart of hearts, you know that handling out-of-memory exceptions is the only truly moral course of action. At the same time, you are keenly aware of the fact that doing so is a pain in the neck. As a result, chances are that you omit such handling from time to time. Like always, perhaps. Still, you must harbor a lurking sense of guilt. I mean, what if new really *does* yield an exception?  ¤[Item E7, P2](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp2)

You may think that one reasonable way to cope with this matter is to fall back on your days in the gutter, i.e., to use the preprocessor. For example, a common C idiom is to define a type-independent macro to allocate memory and then check to make sure the allocation succeeded. For C++, such a macro might look something like this:  ¤[Item E7, P3](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp3)

#define NEW(PTR, TYPE) \

try { (PTR) = new TYPE; } \

catch (std::bad\_alloc&) { assert(0); }

("Wait! What's this std::bad\_alloc business?", you ask. bad\_alloc is the type of exception operator new throws when it can't satisfy a memory allocation request, and std is the name of the namespace (see [Item 28](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI28_FR.HTM#6429)) where bad\_alloc is defined. "Okay," you continue, "what's this assert business?" Well, if you look in the standard C include file <assert.h> (or its namespace-savvy C++ equivalent, <cassert> — see [Item 49](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI49_FR.HTM#8392)), you'll find that assert is a macro. The macro checks to see if the expression it's passed is non-zero, and, if it's not, it issues an error message and calls abort. Okay, it does that only when the standard macro NDEBUG isn't defined, i.e., in debug mode. In production mode, i.e., when NDEBUG is defined, assert expands to nothing — to a void statement. You thus check assertions only when debugging.)  ¤[Item E7, P4](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp4)

This NEW macro suffers from the common error of using an assert to test a condition that might occur in production code (after all, you can run out of memory at any time), but it also has a drawback specific to C++: it fails to take into account the myriad ways in which new can be used. There are three common syntactic forms for getting new objects of type T, and you need to deal with the possibility of exceptions for each of these forms:  ¤[Item E7, P5](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp5)

new T;

new T(*constructor arguments*);

new T[*size*];

This oversimplifies the problem, however, because clients can define their own (overloaded) versions of operator new, so programs may contain an arbitrary number of different syntactic forms for using new.  ¤[Item E7, P6](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp6)

How, then, to cope? If you're willing to settle for a very simple error-handling strategy, you can set things up so that if a request for memory cannot be satisfied, an error-handling function you specify is called. This strategy relies on the convention that when operator new cannot satisfy a request, it calls a client-specifiable error-handling function — often called a *new-handler* — before it throws an exception. (In truth, what operator new really does is slightly more complicated. Details are provided in [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851).)  ¤[Item E7, P7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp7)

To specify the out-of-memory-handling function, clients call set\_new\_handler, which is specified in the header <new> more or less like this:  ¤[Item E7, P8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp8)

typedef void (\*new\_handler)();

new\_handler set\_new\_handler(new\_handler p) throw();

As you can see, new\_handler is a typedef for a pointer to a function that takes and returns nothing, and set\_new\_handler is a function that takes and returns a new\_handler.  ¤[Item E7, P9](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp9)

set\_new\_handler's parameter is a pointer to the function operator new should call if it can't allocate the requested memory. The return value of set\_new\_handler is a pointer to the function in effect for that purpose before set\_new\_handler was called.  ¤[Item E7, P10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp10)

You use set\_new\_handler like this:  ¤[Item E7, P11](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp11)

// function to call if operator new can't allocate enough memory

void noMoreMemory()

{

cerr << "Unable to satisfy request for memory\n";

abort();

}

int main()

{

set\_new\_handler(noMoreMemory);

int \*pBigDataArray = new int[100000000];

...

}

If, as seems likely, operator new is unable to allocate space for 100,000,000 integers, noMoreMemory will be called, and the program will abort after issuing an error message. This is a marginally better way to terminate the program than a simple core dump. (By the way, consider what happens if memory must be dynamically allocated during the course of writing the error message to cerr...)  ¤[Item E7, P12](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp12)

When operator new cannot satisfy a request for memory, it calls the new-handler function not once, but *repeatedly* until it can find enough memory. The code giving rise to these repeated calls is shown in [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851), but this high-level description is enough to conclude that a well-designed new-handler function must do one of the following:  ¤[Item E7, P13](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp13)

* **Make more memory available.** This may allow operator new's next attempt to allocate the memory to succeed. One way to implement this strategy is to allocate a large block of memory at program start-up, then release it the first time the new-handler is invoked. Such a release is often accompanied by some kind of warning to the user that memory is low and that future requests may fail unless more memory is somehow made available.  ¤[Item E7, P14](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp14)
* **Install a different new-handler.** If the current new-handler can't make any more memory available, perhaps it knows of a different new-handler that is more resourceful. If so, the current new-handler can install the other new-handler in its place (by calling set\_new\_handler). The next time operator new calls the new-handler function, it will get the one most recently installed. (A variation on this theme is for a new-handler to modify its *own* behavior, so the next time it's invoked, it does something different. One way to achieve this is to have the new-handler modify static or global data that affects the new-handler's behavior.)  ¤[Item E7, P15](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp15)
* **Deinstall the new-handler,** i.e., pass the null pointer to set\_new\_handler. With no new-handler installed, operator new will throw an exception of type std::bad\_alloc when its attempt to allocate memory is unsuccessful.  ¤[Item E7, P16](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp16)
* **Throw an exception** of type std::bad\_alloc or some type derived from std::bad\_alloc. Such exceptions will not be caught by operator new, so they will propagate to the site originating the request for memory. (Throwing an exception of a different type will violate operator new's exception specification. The default action when that happens is to call abort, so if your new-handler is going to throw an exception, you definitely want to make sure it's from the std::bad\_alloc hierarchy. For more information on exception specifications, see [Item M14](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI14_FR.HTM#6011).)  ¤[Item E7, P17](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp17)
* **Not return**, typically by calling abort or exit, both of which are found in the standard C library (and thus in the standard C++ library — see [Item 49](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI49_FR.HTM#8392)).  ¤[Item E7, P18](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp18)

These choices give you considerable flexibility in implementing new-handler functions.  ¤[Item E7, P19](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp19)

Sometimes you'd like to handle memory allocation failures in different ways, depending on the class of the object being allocated:  ¤[Item E7, P20](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp20)

class X {

public:

static void outOfMemory();

...

};

class Y {

public:

static void outOfMemory();

...

};

X\* p1 = new X; // if allocation is unsuccessful,

// call X::outOfMemory

Y\* p2 = new Y; // if allocation is unsuccessful,

// call Y::outOfMemory

C++ has no support for class-specific new-handlers, but it doesn't need to. You can implement this behavior yourself. You just have each class provide its own versions of set\_new\_handler and operator new. The class's set\_new\_handler allows clients to specify the new-handler for the class (just like the standard set\_new\_handler allows clients to specify the global new-handler). The class's operator new ensures that the class-specific new-handler is used in place of the global new-handler when memory for class objects is allocated.  ¤[Item E7, P21](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp21)

Consider a class X for which you want to handle memory allocation failures. You'll have to keep track of the function to call when operator new can't allocate enough memory for an object of type X, so you'll declare a static member of type new\_handler to point to the new-handler function for the class. Your class X will look something like this:  ¤[Item E7, P22](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp22)

class X {

public:

static new\_handler set\_new\_handler(new\_handler p);

static void \* operator new(size\_t size);

private:

static new\_handler currentHandler;

};

Static class members must be defined outside the class definition. Because you'll want to use the default initialization of static objects to 0, you'll define X::currentHandler without initializing it:  ¤[Item E7, P23](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp23)

new\_handler X::currentHandler; // sets currentHandler

// to 0 (i.e., null) by

// default

The set\_new\_handler function in class X will save whatever pointer is passed to it. It will return whatever pointer had been saved prior to the call. This is exactly what the standard version of set\_new\_handler does:  ¤[Item E7, P24](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp24)

new\_handler X::set\_new\_handler(new\_handler p)

{

new\_handler oldHandler = currentHandler;

currentHandler = p;

return oldHandler;

}

Finally, X's operator new will do the following:  ¤[Item E7, P25](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp25)

1. Call the standard set\_new\_handler with X's error-handling function. This will install X's new-handler as the global new- handler. In the code below, notice how you explicitly reference the std scope (where the standard set\_new\_handler resides) by using the "::" notation.  ¤[Item E7, P26](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp26)
2. Call the global operator new to actually allocate the requested memory. If the initial attempt at allocation fails, the global operator new will invoke X's new-handler, because that function was just installed as the global new-handler. If the global operator new is ultimately unable to find a way to allocate the requested memory, it will throw a std::bad\_alloc exception, which X's operator new will catch. X's operator new will then restore the global new-handler that was originally in place, and it will return by propagating the exception.  ¤[Item E7, P27](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp27)
3. Assuming the global operator new was able to successfully allocate enough memory for an object of type X, X's operator new will again call the standard set\_new\_handler to restore the global error-handling function to what it was originally. It will then return a pointer to the allocated memory.  ¤[Item E7, P28](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp28)

Here's how you say all that in C++:  ¤[Item E7, P29](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp29)

void \* X::operator new(size\_t size)

{

new\_handler globalHandler = // install X's

std::set\_new\_handler(currentHandler); // handler

void \*memory;

try { // attempt

memory = ::operator new(size); // allocation

}

catch (std::bad\_alloc&) { // restore

std::set\_new\_handler(globalHandler); // handler;

throw; // propagate

} // exception

std::set\_new\_handler(globalHandler); // restore

// handler

return memory;

}

If the duplicated calls to std::set\_new\_handler caught your eye, turn to [Item M9](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI9_FR.HTM#5292) for information on how to eliminate them.  ¤[Item E7, P30](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp30)

Clients of class X use its new-handling capabilities like this:  ¤[Item E7, P31](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp31)

void noMoreMemory(); // decl. of function to

// call if memory allocation

// for X objects fails

X::set\_new\_handler(noMoreMemory);

// set noMoreMemory as X's

// new-handling function

X \*px1 = new X; // if memory allocation

// fails, call noMoreMemory

string \*ps = new string; // if memory allocation

// fails, call the global

// new-handling function

// (if there is one)

X::set\_new\_handler(0); // set the X-specific

// new-handling function

// to nothing (i.e., null)

X \*px2 = new X; // if memory allocation

// fails, throw an exception

// immediately. (There is

// no new-handling function

// for class X.)

You may note that the code for implementing this scheme is the same regardless of the class, so a reasonable inclination would be to reuse it in other places. As [Item 41](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI41_FR.HTM#7611) explains, both inheritance and templates can be used to create reusable code. However, in this case, it's a combination of the two that gives you what you need.  ¤[Item E7, P32](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp32)

All you have to do is create a "mixin-style" base class, i.e., a base class that's designed to allow derived classes to inherit a single specific capability — in this case, the ability to set a class-specific new-handler. Then you turn the base class into a template. The base class part of the design lets derived classes inherit the set\_new\_handler and operator new functions they all need, while the template part of the design ensures that each inheriting class gets a different currentHandler data member. The result may sound a little complicated, but you'll find that the code looks reassuringly familiar. In fact, about the only real difference is that it's now reusable by any class that wants it:  ¤[Item E7, P33](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp33)

template<class T> // "mixin-style" base class

class NewHandlerSupport { // for class-specific

public: // set\_new\_handler support

static new\_handler set\_new\_handler(new\_handler p);

static void \* operator new(size\_t size);

private:

static new\_handler currentHandler;

};

template<class T>

new\_handler NewHandlerSupport<T>::set\_new\_handler(new\_handler p)

{

new\_handler oldHandler = currentHandler;

currentHandler = p;

return oldHandler;

}

template<class T>

void \* NewHandlerSupport<T>::operator new(size\_t size)

{

new\_handler globalHandler =

std::set\_new\_handler(currentHandler);

void \*memory;

try {

memory = ::operator new(size);

}

catch (std::bad\_alloc&) {

std::set\_new\_handler(globalHandler);

throw;

}

std::set\_new\_handler(globalHandler);

return memory;

}

// this sets each currentHandler to 0

template<class T>

new\_handler NewHandlerSupport<T>::currentHandler;

With this class template, adding set\_new\_handler support to class X is easy: X just inherits from newHandlerSupport<X>:  ¤[Item E7, P34](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp34)

// note inheritance from mixin base class template. (See

// [my article on counting objects](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MAGAZINE\CO_FRAME.HTM) for information on why

// private inheritance might be preferable here.)

class X: public NewHandlerSupport<X> {

... // as before, but no declarations for

}; // set\_new\_handler or operator new

Clients of X remain oblivious to all the behind-the-scenes action; their old code continues to work. This is good, because one thing you can usually rely on your clients being is oblivious.  ¤[Item E7, P35](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp35)

Using set\_new\_handler is a convenient, easy way to cope with the possibility of out-of-memory conditions. Certainly it's a lot more attractive than wrapping every use of new inside a try block. Furthermore, templates like NewHandlerSupport make it simple to add a class-specific new-handler to any class that wants one. Mixin-style inheritance, however, invariably leads to the topic of multiple inheritance, and before starting down that slippery slope, you'll definitely want to read [Item 43](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI43_FR.HTM#7778).  ¤[Item E7, P36](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp36)

Until 1993, C++ required that operator new return 0 when it was unable to satisfy a memory request. The current behavior is for operator new to throw a std::bad\_alloc exception, but a lot of C++ was written before compilers began supporting the revised specification. The **°**[C++ standardization committee](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=committee) didn't want to abandon the established test-for-0 code base, so they provided alternative forms of operator new (and operator new[] — see [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851)) that continue to offer the traditional failure-yields-0 behavior. These forms are called "nothrow" forms because, well, they never do a throw, and they employ nothrow objects (defined in the standard header <new>) at the point where new is used:  ¤[Item E7, P37](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp37)

class Widget { ... };

Widget \*pw1 = new Widget; // throws std::bad\_alloc if

// allocation fails

if (pw1 == 0) ... // this test *must* fail

Widget \*pw2 =

new (nothrow) Widget; // returns 0 if allocation

// fails

if (pw2 == 0) ... // this test may succeed

Regardless of whether you use "normal" (i.e., exception-throwing) new or "nothrow" new, it's important that you be prepared to handle memory allocation failures. The easiest way to do that is to take advantage of set\_new\_handler, because it works with both forms.  ¤[Item E7, P38](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_DIR.HTM#dingp38)

**Item 8:  Adhere to convention when writing operator new and operator delete.**  ¤[Item E8, P1](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp1)

When you take it upon yourself to write operator new ([Item 10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_FR.HTM#1986) explains why you might want to), it's important that your function(s) offer behavior that is consistent with the default operator new. In practical terms, this means having the right return value, calling an error-handling function when insufficient memory is available (see [Item 7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_FR.HTM#1894)), and being prepared to cope with requests for no memory. You also need to avoid inadvertently hiding the "normal" form of new, but that's a topic for [Item 9](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI9_FR.HTM#1961).  ¤[Item E8, P2](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp2)

The return value part is easy. If you can supply the requested memory, you just return a pointer to it. If you can't, you follow the rule described in [Item 7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_FR.HTM#1894) and throw an exception of type std::bad\_alloc.  ¤[Item E8, P3](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp3)

It's not quite that simple, however, because operator new actually tries to allocate memory more than once, calling the error-handling function after each failure, the assumption being that the error-handling function might be able to do something to free up some memory. Only when the pointer to the error-handling function is null does operator new throw an exception.  ¤[Item E8, P4](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp4)

In addition, the **°**[C++ standard](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=cstandard) requires that operator new return a legitimate pointer even when 0 bytes are requested. (Believe it or not, requiring this odd-sounding behavior actually simplifies things elsewhere in the language.)  ¤[Item E8, P5](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp5)

That being the case, pseudocode for a non-member operator new looks like this:  ¤[Item E8, P6](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp6)

void \* operator new(size\_t size) // your operator new might

{ // take additional params

if (size == 0) { // handle 0-byte requests

size = 1; // by treating them as

} // 1-byte requests

while (1) {

*attempt to allocate* size *bytes*;

if (*the allocation was successful*)

return (*a pointer to the memory*);

// allocation was unsuccessful; find out what the

// current error-handling function is (see [Item 7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_FR.HTM#1894))

new\_handler globalHandler = set\_new\_handler(0);

set\_new\_handler(globalHandler);

if (globalHandler) (\*globalHandler)();

elsethrow std::bad\_alloc();

}

}

The trick of treating requests for zero bytes as if they were really requests for one byte looks slimy, but it's simple, it's legal, it works, and how often do you expect to be asked for zero bytes, anyway?  ¤[Item E8, P7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp7)

You may also look askance at the place in the pseudocode where the error-handling function pointer is set to null, then promptly reset to what it was originally. Unfortunately, there is no way to get at the error-handling function pointer directly, so you have to call set\_new\_handler to find out what it is. Crude, yes, but also effective.  ¤[Item E8, P8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp8)

[Item 7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_FR.HTM#1894) remarks that operator new contains an infinite loop, and the code above shows that loop explicitly — while (1) is about as infinite as it gets. The only way out of the loop is for memory to be successfully allocated or for the new-handling function to do one of the things described in [Item 7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI7_FR.HTM#1894): make more memory available, install a different new-handler, deinstall the new-handler, throw an exception of or derived from std::bad\_alloc, or fail to return. It should now be clear why the new-handler must do one of those things. If it doesn't, the loop inside operator new will never terminate.  ¤[Item E8, P9](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp9)

One of the things many people don't realize about operator new is that it's inherited by subclasses. That can lead to some interesting complications. In the pseudocode for operator new above, notice that the function tries to allocate size bytes (unless size is 0). That makes perfect sense, because that's the argument that was passed to the function. However, most class-specific versions of operator new (including the one you'll find in [Item 10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_FR.HTM#1986)) are designed for a *specific* class, *not* for a class *or* any of its subclasses. That is, given an operator new for a class X, the behavior of that function is almost always carefully tuned for objects of size sizeof(X) — nothing larger and nothing smaller. Because of inheritance, however, it is possible that the operator new in a base class will be called to allocate memory for an object of a derived class:  ¤[Item E8, P10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp10)

class Base {

public:

static void \* operator new(size\_t size);

...

};

class Derived: public Base // Derived doesn't declare

{ ... }; // operator new

Derived \*p = new Derived; // calls Base::operator new!

If Base's class-specific operator new wasn't designed to cope with this — and chances are slim that it was — the best way for it to handle the situation is to slough off calls requesting the "wrong" amount of memory to the standard operator new, like this:  ¤[Item E8, P11](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp11)

void \* Base::operator new(size\_t size)

{

if (size != sizeof(Base)) // if size is "wrong,"

return ::operator new(size); // have standard operator

// new handle the request

... // otherwise handle

// the request here

}

"Hold on!" I hear you cry, "You forgot to check for the pathological-but-nevertheless-possible case where size is zero!" Actually, I didn't, and please stop using hyphens when you cry out. The test is still there, it's just been incorporated into the test of size against sizeof(Base). The **°**[C++ standard](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=cstandard) works in mysterious ways, and one of those ways is to decree that all freestanding classes have nonzero size. By definition, sizeof(Base) can never be zero (even if it has no members), so if size is zero, the request will be forwarded to ::operator new, and it will become that function's responsibility to treat the request in a reasonable fashion. (Interestingly, sizeof(Base) may be zero if Base is not a freestanding class. For details, consult [my article on counting objects](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MAGAZINE\CO_FRAME.HTM).)  ¤[Item E8, P12](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp12)

If you'd like to control memory allocation for arrays on a per-class basis, you need to implement operator new's array-specific cousin, operator new[]. (This function is usually called "array new," because it's hard to figure out how to pronounce "operator new[]".) If you decide to write operator new[], remember that all you're doing is allocating raw memory — you can't do anything to the as-yet-nonexistent objects in the array. In fact, you can't even figure out how many objects will be in the array, because you don't know how big each object is. After all, a base class's operator new[] might, through inheritance, be called to allocate memory for an array of derived class objects, and derived class objects are usually bigger than base class objects. Hence, you can't assume inside Base::operator new[] that the size of each object going into the array is sizeof(Base), and that means you can't assume that the number of objects in the array is (*bytes* *requested*)/sizeof(Base). For more information on operator new[], see [Item M8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI8_FR.HTM#33985).  ¤[Item E8, P13](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp13)

So much for the conventions you need to follow when writing operator new (and operator new[]). For operator delete (and its array counterpart, operator delete[]), things are simpler. About all you need to remember is that C++ guarantees it's always safe to delete the null pointer, so you need to honor that guarantee. Here's pseudocode for a non-member operator delete:  ¤[Item E8, P14](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp14)

void operator delete(void \*rawMemory)

{

if (rawMemory == 0) return; // do nothing if the null

// pointer is being deleted

*deallocate the memory pointed to by* rawMemory;

return;

}

The member version of this function is simple, too, except you've got to be sure to check the size of what's being deleted. Assuming your class-specific operator new forwards requests of the "wrong" size to ::operator new, you've got to forward "wrongly sized" deletion requests to ::operator delete:  ¤[Item E8, P15](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp15)

class Base { // same as before, but now

public: // op. delete is declared

static void \* operator new(size\_t size);

static void operator delete(void \*rawMemory, size\_t size);

...

};

void Base::operator delete(void \*rawMemory, size\_t size)

{

if (rawMemory == 0) return; // check for null pointer

if (size != sizeof(Base)) { // if size is "wrong,"

::operator delete(rawMemory); // have standard operator

return; // delete handle the request

}

*deallocate the memory pointed to by* rawMemory;

return;

}

The conventions, then, for operator new and operator delete (and their array counterparts) are not particularly onerous, but it is important that you obey them. If your allocation routines support new-handler functions and correctly deal with zero-sized requests, you're all but finished, and if your deallocation routines cope with null pointers, there's little more to do. Add support for inheritance in member versions of the functions, and *presto!* — you're done.  ¤[Item E8, P16](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_DIR.HTM#dingp16)

**Item 9:  Avoid hiding the "normal" form of new.**  ¤[Item E9, P1](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp1)

A declaration of a name in an inner scope hides the same name in outer scopes, so for a function f at both global and class scope, the member function will hide the global function:  ¤[Item E9, P2](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp2)

void f(); // global function

class X {

public:

void f(); // member function

};

X x;

f(); // calls global f

x.f(); // calls X::f

This is unsurprising and normally causes no confusion, because global and member functions are usually invoked using different syntactic forms. However, if you add to this class an operator new taking additional parameters, the result is likely to be an eye-opener:  ¤[Item E9, P3](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp3)

class X {

public:

void f();

// operator new allowing specification of a

// new-handling function

static void \* operator new(size\_t size, new\_handler p);

};

void specialErrorHandler(); // definition is elsewhere

X \*px1 =

new (specialErrorHandler) X; // calls X::operator new

X \*px2 = new X; // error!

By declaring a function called "operator new" inside the class, you inadvertently block access to the "normal" form of new. Why this is so is discussed in [Item 50](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI50_FR.HTM#8569). Here we're more interested in figuring out how to avoid the problem.  ¤[Item E9, P4](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp4)

One solution is to write a class-specific operator new that supports the "normal" invocation form. If it does the same thing as the global version, that can be efficiently and elegantly encapsulated as an inline function:  ¤[Item E9, P5](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp5)

class X {

public:

void f();

static void \* operator new(size\_t size, new\_handler p);

static void \* operator new(size\_t size)

{ return ::operator new(size); }

};

X \*px1 =

new (specialErrorHandler) X; // calls X::operator

// new(size\_t, new\_handler)

X\* px2 = new X; // calls X::operator

// new(size\_t)

An alternative is to provide a default parameter value (see [Item 24](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI24_FR.HTM#6250)) for each additional parameter you add to operator new:  ¤[Item E9, P6](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp6)

class X {

public:

void f();

static

void \* operator new(size\_t size, // note default

new\_handler p = 0); // value for p

};

X \*px1 = new (specialErrorHandler) X; // fine

X\* px2 = new X; // also fine

Either way, if you later decide to customize the behavior of the "normal" form of new, all you need to do is rewrite the function; callers will get the customized behavior automatically when they relink.  ¤[Item E9, P7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI9_DIR.HTM#dingp7)

**Item 10:  Write operator delete if you write operator new.**  ¤[Item E10, P1](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp1)

Let's step back for a moment and return to fundamentals. Why would anybody want to write their own version of operator new or operator delete in the first place?  ¤[Item E10, P2](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp2)

More often than not, the answer is efficiency. The default versions of operator new and operator delete are perfectly adequate for general-purpose use, but their flexibility inevitably leaves room for improvements in their performance in a more circumscribed context. This is especially true for applications that dynamically allocate a large number of small objects.  ¤[Item E10, P3](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp3)

As an example, consider a class for representing airplanes, where the Airplane class contains only a pointer to the actual representation for airplane objects (a technique discussed in [Item 34](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI34_FR.HTM#6793)):  ¤[Item E10, P4](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp4)

class AirplaneRep { ... }; // representation for an

// Airplane object

class Airplane {

public:

...

private:

AirplaneRep \*rep; // pointer to representation

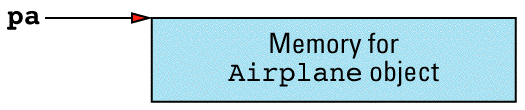
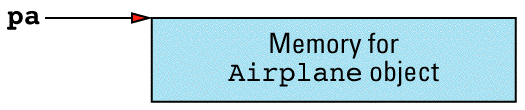
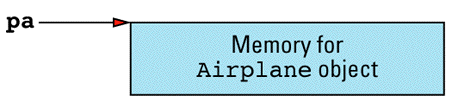
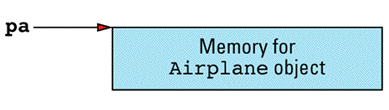
};

An Airplane object is not very big; it contains but a single pointer. (As explained in Items [14](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI14_FR.HTM#223029) and [M24](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI24_FR.HTM#41284), it may implicitly contain a second pointer if the Airplane class declares virtual functions.) When you allocate an Airplane object by calling operator new, however, you probably get back more memory than is needed to store this pointer (or pair of pointers). The reason for this seemingly wayward behavior has to do with the need for operator new and operator delete to communicate with one another.  ¤[Item E10, P5](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp5)

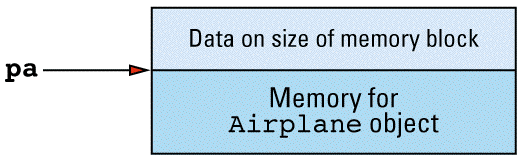
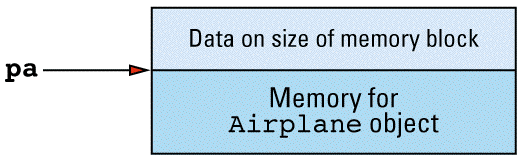
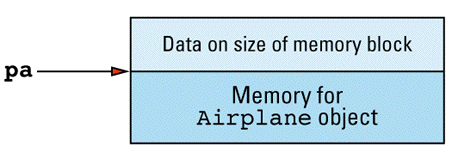
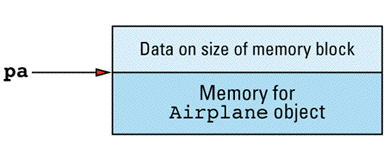
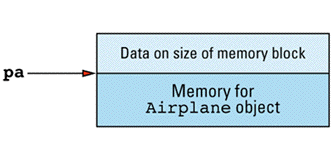
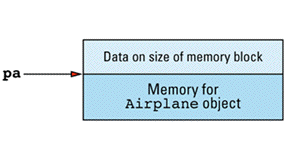
Because the default version of operator new is a general-purpose allocator, it must be prepared to allocate blocks of any size. Similarly, the default version of operator delete must be prepared to deallocate blocks of whatever size operator new allocated. For operator delete to know how much memory to deallocate, it must have some way of knowing how much memory operator new allocated in the first place. A common way for operator new to tell operator delete how much memory it allocated is by prepending to the memory it returns some additional data that specifies the size of the allocated block. That is, when you say this,  ¤[Item E10, P6](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp6)

Airplane \*pa = new Airplane;

you don't necessarily get back a block of memory that looks like this:  ¤[Item E10, P7](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp7)



Instead, you often get back a block of memory that looks more like this:  ¤[Item E10, P8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp8)



For small objects like those of class Airplane, this additional bookkeeping data can more than double the amount of memory needed for each dynamically allocated object (especially if the class contains no virtual functions).  ¤[Item E10, P9](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp9)

If you're developing software for an environment in which memory is precious, you may not be able to afford this kind of spendthrift allocation. By writing your own operator new for the Airplane class, you can take advantage of the fact that all Airplane objects are the same size, so there isn't any need for bookkeeping information to be kept with each allocated block.  ¤[Item E10, P10](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp10)

One way to implement your class-specific operator new is to ask the default operator new for big blocks of raw memory, each block of sufficient size to hold a large number of Airplane objects. The memory chunks for Airplane objects themselves will be taken from these big blocks. Currently unused chunks will be organized into a linked list — the *free* list — of chunks that are available for future Airplane use. This may make it sound like you'll have to pay for the overhead of a next field in every object (to support the list), but you won't: the space for the rep field (which is necessary only for memory chunks in use as Airplane objects) will also serve as the place to store the next pointer (because that pointer is needed only for chunks of memory *not* in use as Airplane objects). You'll arrange for this job-sharing in the usual fashion: you'll use a union.  ¤[Item E10, P11](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp11)

To turn this design into reality, you have to modify the definition of Airplane to support custom memory management. You do it as follows:  ¤[Item E10, P12](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp12)

class Airplane { // modified class — now supports

public: // custom memory management

static void \* operator new(size\_t size);

...

private:

union {

AirplaneRep \*rep; // for objects in use

Airplane \*next; // for objects on free list

};

// this class-specific constant (see [Item 1](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI1_FR.HTM#1790)) specifies how

// many Airplane objects fit into a big memory block;

// it's initialized below

static const int BLOCK\_SIZE;

static Airplane \*headOfFreeList;

};

Here you've added the declarations for operator new, the union that allows the rep and next fields to occupy the same memory, a class-specific constant for specifying how big each allocated block should be, and a static pointer to keep track of the head of the free list. It's important to use a static member for this last task, because there's one free list for the entire *class*, not one free list for each Airplane *object*.  ¤[Item E10, P13](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp13)

The next thing to do is to write the new operator new:  ¤[Item E10, P14](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp14)

void \* Airplane::operator new(size\_t size)

{

// send requests of the "wrong" size to ::operator new();

// for details, see [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851)

if (size != sizeof(Airplane))

return ::operator new(size);

Airplane \*p = // p is now a pointer to the

headOfFreeList; // head of the free list

// if p is valid, just move the list head to the

// next element in the free list

if (p)

headOfFreeList = p->next;

else {

// The free list is empty. Allocate a block of memory

// big enough to hold BLOCK\_SIZE Airplane objects

Airplane \*newBlock =

static\_cast<Airplane\*>(::operator new(BLOCK\_SIZE \*

sizeof(Airplane)));

// form a new free list by linking the memory chunks

// together; skip the zeroth element, because you'll

// return that to the caller of operator new

for (int i = 1; i < BLOCK\_SIZE-1; ++i)

newBlock[i].next = &newBlock[i+1];

// terminate the linked list with a null pointer

newBlock[BLOCK\_SIZE-1].next = 0;

// set p to front of list, headOfFreeList to

// chunk immediately following

p = newBlock;

headOfFreeList = &newBlock[1];

}

return p;

}

If you've read [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851), you know that when operator new can't satisfy a request for memory, it's supposed to perform a series of ritualistic steps involving new-handler functions and exceptions. There is no sign of such steps above. That's because this operator new gets all the memory it manages from ::operator new. That means this operator new can fail only if ::operator new does. But if ::operator new fails, *it* must engage in the new-handling ritual (possibly culminating in the throwing of an exception), so there is no need for Airplane's operator new to do it, too. In other words, the new-handler behavior is there, you just don't see it, because it's hidden inside ::operator new.  ¤[Item E10, P15](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp15)

Given this operator new, the only thing left to do is provide the obligatory definitions of Airplane's static data members:  ¤[Item E10, P16](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp16)

Airplane \*Airplane::headOfFreeList; // these definitions

// go in an implemen-

const int Airplane::BLOCK\_SIZE = 512; // tation file, not

// a header file

There's no need to explicitly set headOfFreeList to the null pointer, because static members are initialized to 0 by default. The value for BLOCK\_SIZE, of course, determines the size of each memory block we get from ::operator new.  ¤[Item E10, P17](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp17)

This version of operator new will work just fine. Not only will it use a lot less memory for Airplane objects than the default operator new, it's also likely to be faster, possibly as much as two orders of magnitude faster. That shouldn't be surprising. After all, the general version of operator new has to cope with memory requests of different sizes, has to worry about internal and external fragmentation, etc., whereas your version of operator new just manipulates a couple of pointers in a linked list. It's easy to be fast when you don't have to be flexible.  ¤[Item E10, P18](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp18)

At long last we are in a position to discuss operator delete. Remember operator delete? This Item is *about* operator delete. As currently written, your Airplane class declares operator new, but it does not declare operator delete. Now consider what happens when a client writes the following, which is nothing if not eminently reasonable:  ¤[Item E10, P19](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp19)

Airplane \*pa = new Airplane; // calls

// Airplane::operator new

...

delete pa; // calls ::operator delete

If you listen closely when you read this code, you can hear the sound of an airplane crashing and burning, with much weeping and wailing by the programmers who knew it. The problem is that operator new (the one defined in Airplane) returns a pointer to memory *without any header information*, but operator delete (the default, global one) assumes that the memory it's passed *does* contain header information! Surely this is a recipe for disaster.  ¤[Item E10, P20](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp20)

This example illustrates the general rule: operator new and operator delete must be written in concert so that they share the same assumptions. If you're going to roll your own memory allocation routine, be sure to roll one for deallocation, too. (For another reason why you should follow this advice, turn to [the sidebar on placement new and placement delete](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MAGAZINE\CO_FRAME.HTM#sidebar) in my [article on counting objects](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MAGAZINE\CO_FRAME.HTM) in C++.)  ¤[Item E10, P21](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp21)

Here's how you solve the problem with the Airplane class:  ¤[Item E10, P22](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp22)

class Airplane { // same as before, except there's

public: // now a decl. for operator delete

...

static void operator delete(void \*deadObject,

size\_t size);

};

// operator delete is passed a memory chunk, which,

// if it's the right size, is just added to the

// front of the list of free chunks

void Airplane::operator delete(void \*deadObject,

size\_t size)

{

if (deadObject == 0) return; // see [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851)

if (size != sizeof(Airplane)) { // see [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851)

::operator delete(deadObject);

return;

}

Airplane \*carcass =

static\_cast<Airplane\*>(deadObject);

carcass->next = headOfFreeList;

headOfFreeList = carcass;

}

Because you were careful in operator new to ensure that calls of the "wrong" size were forwarded to the global operator new (see [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851)), you must demonstrate equal care in ensuring that such "improperly sized" objects are handled by the global version of operator delete. If you did not, you'd run into precisely the problem you have been laboring so arduously to avoid — a semantic mismatch between new and delete.  ¤[Item E10, P23](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp23)

Interestingly, the size\_t value C++ passes to operator delete may be *incorrect* if the object being deleted was derived from a base class lacking a virtual destructor. This is reason enough for making sure your base classes have virtual destructors, but [Item 14](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI14_FR.HTM#223029) describes a second, arguably better reason. For now, simply note that if you omit virtual destructors in base classes, operator delete functions may not work correctly.  ¤[Item E10, P24](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp24)

All of which is well and good, but I can tell by the furrow in your brow that what you're really concerned about is the memory leak. With all the software development experience you bring to the table, there's no way you'd fail to notice that Airplane's operator new calls ::operator new to get big blocks of memory, but Airplane's operator delete fails to release those blocks.[4](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10.HTM#14365#14365) *Memory leak! Memory leak!* I can almost hear the alarm bells going off in your head.  ¤[Item E10, P25](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp25)

Listen to me carefully: *there is no memory leak*.  ¤[Item E10, P26](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp26)

A memory leak arises when memory is allocated, then all pointers to that memory are lost. Absent garbage collection or some other extralinguistic mechanism, such memory cannot be reclaimed. But this design has no memory leak, because it's never the case that all pointers to memory are lost. Each big block of memory is first broken down into Airplane-sized chunks, and these chunks are then placed on the free list. When clients call Airplane::operator new, chunks are removed from the free list, and clients receive pointers to them. When clients call operator delete, the chunks are put back on the free list. With this design, all memory chunks are either in use as Airplane objects (in which case it's the clients' responsibility to avoid leaking their memory) or are on the free list (in which case there's a pointer to the memory). There is no memory leak.  ¤[Item E10, P27](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp27)

Nevertheless, the blocks of memory returned by ::operator new are never released by Airplane::operator delete, and there has to be *some* name for that. There is. You've created a memory *pool*. Call it semantic gymnastics if you must, but there is an important difference between a memory leak and a memory pool. A memory leak may grow indefinitely, even if clients are well-behaved, but a memory pool never grows larger than the maximum amount of memory requested by its clients.  ¤[Item E10, P28](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp28)

It would not be difficult to modify Airplane's memory management routines so that the blocks of memory returned by ::operator new were automatically released when they were no longer in use, but there are two reasons why you might not want to do it.  ¤[Item E10, P29](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp29)

The first concerns your likely motivation for tackling custom memory management. There are many reasons why you might do it, but the most common one is that you've determined (see [Item M16](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\MEC\MI16_FR.HTM#40995)) that the default operator new and operator delete use too much memory or are too slow (or both). That being the case, every additional byte and every additional statement you devote to tracking and releasing those big memory blocks comes straight off the bottom line: your software runs slower and uses more memory than it would if you adopted the pool strategy. For libraries and applications in which performance is at a premium and you can expect pool sizes to be reasonably bounded, the pool approach may well be best.  ¤[Item E10, P30](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp30)

The second reason has to do with pathological behavior. Suppose Airplane's memory management routines are modified so Airplane's operator delete releases any big block of memory that has no active objects in it. Now consider this program:  ¤[Item E10, P31](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp31)

int main()

{

Airplane \*pa = new Airplane; // first allocation: get big

// block, make free list, etc.

delete pa; // block is now empty;

// release it

pa = new Airplane; // uh oh, get block again,

// make free list, etc.

delete pa; // okay, block is empty

// again; release it

... // you get the idea...

return 0;

}

This nasty little program will run slower and use more memory than with even the *default* operator new and operator delete, much less the pool-based versions of those functions!  ¤[Item E10, P32](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp32)

Of course, there are ways to deal with this pathology, but the more you code for uncommon special cases, the closer you get to reimplementing the default memory management functions, and then what have you gained? A memory pool is not the answer to all memory management questions, but it's a reasonable answer to many of them.  ¤[Item E10, P33](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp33)

In fact, it's a reasonable answer often enough that you may be bothered by the need to reimplement it for different classes. "Surely," you think to yourself, "there should be a way to package the notion of a fixed-sized memory allocator so it's easily reused." There is, though this Item has droned on long enough that I'll leave the details in the form of the dreaded exercise for the reader.  ¤[Item E10, P34](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp34)

Instead, I'll simply show a minimal interface (see [Item 18](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI18_FR.HTM#17774)) to a *Pool* class, where each object of type Pool is an allocator for objects of the size specified in the Pool's constructor:  ¤[Item E10, P35](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp35)

class Pool {

public:

Pool(size\_t n); // Create an allocator for

// objects of size n

void \* alloc(size\_t n) ; // Allocate enough memory

// for one object; follow

// operator new conventions

// from [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851)

void free( void \*p, size\_t n); // Return to the pool the

// memory pointed to by p;

// follow operator delete

// conventions from [Item 8](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI8_FR.HTM#120851)

~Pool(); // Deallocate all memory in

// the pool

};

This class allows Pool objects to be created, to perform allocation and deallocation operations, and to be destroyed. When a Pool object is destroyed, it releases all the memory it allocated. This means there is now a way to avoid the memory leak-like behavior that Airplane's functions exhibited. However, this also means that if a Pool's destructor is called too soon (before all the objects using its memory have been destroyed), some objects will find their memory yanked out from under them before they're done using it. To say that the resulting behavior is undefined is being generous.  ¤[Item E10, P36](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp36)

Given this Pool class, even a Java programmer can add custom memory management capabilities to Airplane without breaking a sweat:  ¤[Item E10, P37](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp37)

class Airplane {

public:

... // usual Airplane functions

static void \* operator new(size\_t size);

static void operator delete(void \*p, size\_t size);

private:

AirplaneRep \*rep; // pointer to representation

static Pool memPool; // memory pool for Airplanes

};

inline void \* Airplane::operator new(size\_t size)

{ return memPool.alloc(size); }

inline void Airplane::operator delete(void \*p,

size\_t size)

{ memPool.free(p, size); }

// create a new pool for Airplane objects; this goes in

// the class implementation file

Pool Airplane::memPool(sizeof(Airplane));

This is a much cleaner design than the one we saw earlier, because the Airplane class is no longer cluttered with non-airplane details. Gone are the union, the head of the free list, the constant defining how big each raw memory block should be, etc. That's all hidden inside Pool, which is really where it should be. Let Pool's author worry about memory management minutiae. Your job is to make the Airplane class work properly.  ¤[Item E10, P38](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp38)

Now, it's interesting to see how custom memory management routines can improve program performance, and it's worthwhile to see how such routines can be encapsulated inside a class like Pool, but let us not lose sight of the main point. That point is that operator new and operator delete need to work together, so if you write operator new, be sure to write operator delete, as well.  ¤[Item E10, P39](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp39)

4 I write this with certainty, because I failed to address this issue in the first edition of this book, and *many* readers upbraided me for the omission. There's nothing quite like a few thousand proofreaders to demonstrate one's fallibility, sigh.  ¤[Item E10, P40](file:///C:\downloads\ec\Addison-Wesley%20Effective%20C++%202nd\EC\EI10_DIR.HTM#dingp40)