C++ Gotchas
To John Carolan
This page intentionally left blank
# Contents

Preface xi
Acknowledgments xv

## Chapter 1 Basics

<table>
<thead>
<tr>
<th>Gotcha #1:</th>
<th>Excessive Commenting</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gotcha #2:</td>
<td>Magic Numbers</td>
<td>4</td>
</tr>
<tr>
<td>Gotcha #3:</td>
<td>Global Variables</td>
<td>6</td>
</tr>
<tr>
<td>Gotcha #4:</td>
<td>Failure to Distinguish Overloading from Default Initialization</td>
<td>8</td>
</tr>
<tr>
<td>Gotcha #5:</td>
<td>Misunderstanding References</td>
<td>10</td>
</tr>
<tr>
<td>Gotcha #6:</td>
<td>Misunderstanding Const</td>
<td>13</td>
</tr>
<tr>
<td>Gotcha #7:</td>
<td>Ignorance of Base Language Subtleties</td>
<td>14</td>
</tr>
<tr>
<td>Gotcha #8:</td>
<td>Failure to Distinguish Access and Visibility</td>
<td>19</td>
</tr>
<tr>
<td>Gotcha #9:</td>
<td>Using Bad Language</td>
<td>24</td>
</tr>
<tr>
<td>Gotcha #10:</td>
<td>Ignorance of Idiom</td>
<td>26</td>
</tr>
<tr>
<td>Gotcha #11:</td>
<td>Unnecessary Cleverness</td>
<td>29</td>
</tr>
<tr>
<td>Gotcha #12:</td>
<td>Adolescent Behavior</td>
<td>31</td>
</tr>
</tbody>
</table>

## Chapter 2 Syntax

| Gotcha #13:       | Array/Initializer Confusion | 35 |
| Gotcha #14:       | Evaluation Order Indecision | 36 |
| Gotcha #15:       | Precedence Problems        | 42 |
| Gotcha #16:       | for Statement Debacle      | 45 |
| Gotcha #17:       | Maximal Munch Problems     | 48 |
| Gotcha #18:       | Creative Declaration-Specifier Ordering | 50 |
| Gotcha #19:       | Function/Object Ambiguity  | 51 |
| Gotcha #20:       | Migrating Type-Qualifiers  | 52 |
| Gotcha #21:       | Self-Initialization        | 53 |
| Gotcha #22:       | Static and Extern Types    | 55 |
| Gotcha #23:       | Operator Function Lookup Anomaly | 56 |
| Gotcha #24:       | Operator -> Subtleties     | 58 |
Chapter 3  The Preprocessor  
Gotcha #25: \#define Literals 61  
Gotcha #26: \#define Pseudofunctions 64  
Gotcha #27: Overuse of \#if 66  
Gotcha #28: Side Effects in Assertions 72  

Chapter 4  Conversions  
Gotcha #29: Converting through void * 75  
Gotcha #30: Slicing 79  
Gotcha #31: Misunderstanding Pointer-to-Const Conversion 81  
Gotcha #32: Misunderstanding Pointer-to-Pointer-to-Const Conversion 82  
Gotcha #33: Misunderstanding Pointer-to-Pointer-to-Base Conversion 86  
Gotcha #34: Pointer-to-Multidimensional-Array Problems 87  
Gotcha #35: Unchecked Downcasting 89  
Gotcha #36: Misusing Conversion Operators 90  
Gotcha #37: Unintended Constructor Conversion 95  
Gotcha #38: Casting under Multiple Inheritance 98  
Gotcha #39: Casting Incomplete Types 100  
Gotcha #40: Old-Style Casts 102  
Gotcha #41: Static Casts 103  
Gotcha #42: Temporary Initialization of Formal Arguments 106  
Gotcha #43: Temporary Lifetime 110  
Gotcha #44: References and Temporaries 112  
Gotcha #45: Ambiguity Failure of dynamic_cast 116  
Gotcha #46: Misunderstanding Contravariance 120  

Chapter 5  Initialization  
Gotcha #47: Assignment/Initialization Confusion 125  
Gotcha #48: Improperly Scoped Variables 129  
Gotcha #49: Failure to Appreciate C++’s Fixation on Copy Operations 132  
Gotcha #50: Bitwise Copy of Class Objects 136  
Gotcha #51: Confusing Initialization and Assignment in Constructors 139  
Gotcha #52: Inconsistent Ordering of the Member Initialization List 141  
Gotcha #53: Virtual Base Default Initialization 142  
Gotcha #54: Copy Constructor Base Initialization 147  
Gotcha #55: Runtime Static Initialization Order 150  
Gotcha #56: Direct versus Copy Initialization 153  
Gotcha #57: Direct Argument Initialization 156  
Gotcha #58: Ignorance of the Return Value Optimizations 158  
Gotcha #59: Initializing a Static Member in a Constructor 163
| Gotcha #92: Public Inheritance for Code Reuse       | 281 |
| Gotcha #93: Concrete Public Base Classes          | 285 |
| Gotcha #94: Failure to Employ Degenerate Hierarchies | 286 |
| Gotcha #95: Overuse of Inheritance                | 287 |
| Gotcha #96: Type-Based Control Structures          | 292 |
| Gotcha #97: Cosmic Hierarchies                    | 295 |
| Gotcha #98: Asking Personal Questions of an Object | 299 |
| Gotcha #99: Capability Queries                     | 302 |

Bibliography 307
Index 309
This book is the result of nearly two decades of minor frustrations, serious bugs, late nights, and weekends spent involuntarily at the keyboard. This collection consists of 99 of some of the more common, severe, or interesting C++ gotchas, most of which I have (I’m sorry to say) experienced personally.

The term “gotcha” has a cloudy history and a variety of definitions. For purposes of this book, we’ll define C++ gotchas as common and preventable problems in C++ programming and design. The gotchas described here run the gamut from minor syntactic annoyances to basic design flaws to full-blown sociopathic behavior.

Almost ten years ago, I started including notes about individual gotchas in my C++ course material. My feeling was that pointing out these common misconceptions and misapplications in apposition to correct use would inoculate the student against them and help prevent new generations of C++ programmers from repeating the gotchas of the past. By and large, the approach worked, and I was induced to collect sets of related gotchas for presentation at conferences. These presentations proved to be popular (misery loves company?), and I was encouraged to write a “gotcha” book.

Any discussion of avoiding or recovering from C++ gotchas involves other subjects, most commonly design patterns, idioms, and technical details of C++ language features.

This is not a book about design patterns, but we often find ourselves referring to patterns as a means of avoiding or recovering from a particular gotcha. Conventionally, the pattern name is capitalized, as in “Template Method” pattern or “Bridge” pattern. When we mention a pattern, we describe its mechanics briefly if they’re simple but delegate detailed discussion of patterns to works devoted to them. Unless otherwise noted, a fuller description of a pattern, as well as a richer discussion of patterns in general, may be found in Erich Gamma et al.’s Design Patterns. Descriptions of the Acyclic Visitor, Monostate, and Null Object patterns may be found in Robert Martin’s Agile Software Development.

From the perspective of gotchas, design patterns have two important properties. First, they describe proven, successful design techniques that can be customized in a context-dependent way to new design situations. Second, and perhaps more
important, mentioning the application of a particular pattern serves to document not only the technique applied but also the reasons for its application and the effect of having applied it.

For example, when we see that the Bridge pattern has been applied to a design, we know at a mechanical level that an abstract data type implementation has been separated into an interface class and an implementation class. Additionally, we know this was done to separate strongly the interface from the implementation, so changes to the implementation won't affect users of the interface. We also know this separation entails a runtime cost, how the source code for the abstract data type should be arranged, and many other details.

A pattern name is an efficient, unambiguous handle to a wealth of information and experience about a technique. Careful, accurate use of patterns and pattern terminology in design and documentation clarifies code and helps prevent gotchas from occurring.

C++ is a complex programming language, and the more complex a language, the more important is the use of idiom in programming. For a programming language, an idiom is a commonly used and generally understood combination of lower-level language features that produces a higher-level construct, in much the same way patterns do at higher levels of design. Therefore, in C++ we can discuss copy operations, function objects, smart pointers, and throwing an exception without having to specify these concepts at their lowest level of implementation.

It’s important to emphasize that an idiom is not only a common combination of language features but also a common set of expectations about how these combined features should behave. What do copy operations mean? What can we expect to happen when an exception is thrown? Much of the advice found in this book involves being aware of and employing idioms in C++ coding and design. Many of the gotchas listed here could be described simply as departing from a particular C++ idiom, and the accompanying solution to the problem could often be described simply as following the appropriate idiom (see Gotcha #10).

A significant portion of this book is spent describing the nuances of certain areas of the C++ language that are commonly misunderstood and frequently lead to gotchas. While some of this material may have an esoteric feel to it, unfamiliarity with these areas is a source of problems and a barrier to expert use of C++. These “dark corners” also make an interesting and profitable study in themselves. They are in C++ for a reason, and expert C++ programmers often find use for them in advanced programming and design.
Another area of connection between gotchas and design patterns is the similar importance of describing relatively simple instances. Simple patterns are important. In some respects, they may be more important than technically difficult patterns, because they’re likely to be more commonly employed. The benefits obtained from the pattern description will, therefore, be leveraged over a larger body of code and design.

In much the same way, the gotchas described in this book cover a wide range of difficulty, from a simple exhortation to act like a responsible professional (Gotcha #12) to warnings to avoid misunderstanding the dominance rule under virtual inheritance (Gotcha #79). But, as in the analogous case with patterns, acting responsibly is probably more commonly applicable on a day-to-day basis than is the dominance rule.

Two common themes run through the presentation. The first is the overriding importance of convention. This is especially important in a complex language like C++. Adherence to established convention allows us to communicate efficiently and accurately with others. The second theme is the recognition that others will maintain the code we write. The maintenance may be direct, so that our code must be readily and generally understood by competent maintainers, or it may be indirect, in which case we must ensure that our code remains correct even as its behavior is modified by remote changes.

The gotchas in this book are presented as a collection of short essays, each of which describes a gotcha or set of related gotchas, along with suggestions for avoiding or correcting them. I’m not sure any book about gotchas can be entirely cohesive, due to the anarchistic nature of the subject. However, the gotchas are grouped into chapters according to their general nature or area of (mis)applicability.

Additionally, discussion of one gotcha inevitably touches on others. Where it makes sense to do so—and it generally does—I’ve made these links explicit. Cohesion within each item is sometimes at risk as well. Often it’s necessary, before getting to the description of a gotcha, to describe the context in which it appears. That description, in turn, may require discussion of a technique, idiom, pattern, or language nuance that may lead us even further afield before we return to the advertised gotcha. I’ve tried to keep this meandering to a minimum, but it would have been dishonest, I think, to attempt to avoid it entirely. Effective programming in C++ involves intelligent coordination of so many disparate areas that it’s impractical to imagine one can examine its etiology effectively without involving a similar eclectic collection of topics.

It’s certainly not necessary—and possibly inadvisable—to read this book straight through, from Gotcha #1 to Gotcha #99. Such a concentrated dose of mayhem
may put you off programming in C++ altogether. A better approach may be to start with a gotcha you’ve experienced or that sounds interesting and follow links to related gotchas. Alternatively, you may sample the gotchas at random.

The text employs a number of devices intended to clarify the presentation. First, incorrect or inadvisable code is indicated by a gray background, whereas correct and proper code is presented with no background. Second, code that appears in the text has been edited for brevity and clarity. As a result, the examples as presented often won’t compile without additional, supporting code. The source code for nontrivial examples is available from the author’s Web site: www.semantics.org. All such code is indicated in the text by an abbreviated pathname near the code example, as in `➤➤ gotcha00/somecode.cpp`.

Finally, a warning: the one thing you should not do with gotchas is elevate them to the same status as idioms or patterns. One of the signs that you’re using patterns and idioms properly is that the pattern or idiom appropriate to the design or coding context will arise “spontaneously” from your subconscious just when you need it.

Recognition of a gotcha is analogous to a conditioned response to danger: once burned, twice shy. However, as with matches and firearms, it’s not necessary to suffer a burn or a gunshot wound to the head personally to learn how to recognize and avoid a dangerous situation; generally, all that’s necessary is advance warning. Consider this collection a means to keep your head in the face of C++ gotchas.

Stephen C. Dewhurst
Carver, Massachusetts
July 2002
C++ offers tremendous flexibility in managing memory, but few C++ programmers fully understand the available mechanisms. In this area of the language, overloading, name hiding, constructors and destructors, exceptions, static and virtual functions, operator and non-operator functions all come together to provide great flexibility and customizability of memory management. Unfortunately, and perhaps unavoidably, things can also get a bit complex.

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

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

Gotcha #60: Failure to Distinguish Scalar and Array Allocation

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

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

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

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

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

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

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

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

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

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

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

```cpp
int *ip = new int(12);
// . . .
```
free( ip ); // wrong!

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

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

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

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

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

delete [] dp; // correct

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

class Widget {
    public:
        void *operator new( size_t );
        void operator delete( void *, size_t );
    // . . .
};
The author of the `Widget` class has decided to customize memory management of `Widget`s but has failed to take into account that array operator `new` and `delete` functions have different names from their non-array counterparts and are therefore not hidden by the member versions:

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

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

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

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

The inline member versions of these functions cost nothing at runtime and should convince even the most inattentive maintainer not to second-guess the author’s decision to invoke the global versions of array `new` and `delete` functions for `Widget`s.
Gotcha #61: Checking for Allocation Failure

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

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

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

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

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

```cpp
String **array = new String *[n];
for( String **p = array; p < array+n; ++p )
    *p = new String;
```
This code is shorter, clearer, faster, and correct. The standard behavior of `new` is to throw a `bad_alloc` exception in the event of allocation failure. This allows us to encapsulate error-handling code for allocation failure from the rest of the program, resulting in a cleaner, clearer, and generally more efficient design.

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

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

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

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

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

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

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

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

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

### Gotcha #62: Replacing Global New and Delete

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

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

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

```cpp
void *operator new( size_t n, const string &pat ) {
    char *p = static_cast<char *>(::operator new( n ));
    const char *pattern = pat.c_str();
    if( !pattern[0] )
        pattern = "\0"; // note: two null chars
    const char *f = pattern;
    for( int i = 0; i < n; ++i ) {
        if( !*f )
            f = pattern;
        p[i] = *f++;
    }
    return p;
}
```
This version of `operator new` accepts a string pattern argument that is copied into the newly allocated storage. The compiler distinguishes between the standard `operator new` and our two-argument version through overload resolution.

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

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

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

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

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

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

```cpp
sr->~StatusRegister(); // explicit dtor call, no operator delete
```
Placement new and explicit destruction are clearly useful features, but they’re just as clearly dangerous if not used sparingly and with caution. (See Gotcha #47 for one example from the standard library.)

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

```cpp
template <typename T, typename N, class B>
T *operator new( size_t n, B &buffer ); // overloaded new
T *operator new( size_t n, B &buffer, const T *p ); // corresponding delete

Thing *thing1 = new Thing; // use standard operator new
Buffer buf;
Thing *thing2 = new (buf) Thing; // use overloaded operator new
delete thing2; // incorrect, should have used overloaded delete
delete thing1; // correct, uses standard operator delete
```

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

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

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

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

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

Often, the best approach is to avoid doing anything at all with global scope memory-management operator functions, but instead customize memory management on a class or hierarchy basis through the use of member `operator new`, `operator delete`, `array new`, and `array delete`. 
We noted at the end of Gotcha #61 that an “appropriate” operator delete would be invoked by the runtime system in the event of an exception propagating out of an initialization in a new-expression:

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

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

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

In this case, the appropriate operator delete is the two-argument version corresponding to the overloaded operator new used for the allocation of `Thing`;

```cpp
operator delete( void *, Buffer &),
```

and this is the version the runtime system will invoke.

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

---

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

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

```cpp
class String {
public:
    void *operator new( size_t ); // member operator new
    void operator delete( void *); // member operator delete
    void *operator new[]( size_t ); // member operator new[]
    void operator delete [] ( void *); // member operator delete[]
    String( const char * = "" );
    // ... 
};
```
void f() {
    String *sp = new String( "Heap" ); // uses String::operator new
    int *ip = new int( 12 ); // uses ::operator new
    delete ip; // uses :: operator delete
    delete sp; // uses String::delete
}

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

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

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

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

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

---

### Gotcha #64: Throwing String Literals

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

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

They know this is a reprehensible practice, but they do it anyway, because it’s a “pedagogic example.” Unfortunately, these authors often neglect to mention to their readers that actually following the implicit advice to imitate the example will spell mayhem and doom.
Never throw exception objects that are string literals. The principal reason is that these exception objects should eventually be caught, and they're caught based on their type, not on their value:

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

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

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

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

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

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

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

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

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

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

```cpp
class ContainerFault {
  public:
    virtual ~ContainerFault();
    virtual const char *what() const = 0;
    // . . .
};
```
class StackUnderflow
  : public std::runtime_error, public ContainerFault {
public:
  explicit StackUnderflow( const char *msg = "stack underflow" )
    : std::runtime_error( msg ) {} 
  const char *what() const 
    { return std::runtime_error::what(); } 
};

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

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

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

---

**Gotcha #65: Improper Exception Mechanics**

Issues of general exception-handling policy and architecture are still subject to debate. However, lower-level guidelines concerning how exceptions should be thrown and caught are both well understood and commonly violated.
When a throw-expression is executed, the runtime exception-handling mechanism copies the exception object to a temporary in a “safe” location. The location of the temporary is highly platform dependent, but the temporary is guaranteed to persist until the exception has been handled. This means that the temporary will be usable until the last catch clause that uses the temporary has completed, even if several different catch clauses are executed for that temporary exception object. This is an important property because, to put it bluntly, when you throw an exception, all hell breaks loose. That temporary is the calm in the eye of the exception-handling maelstrom.

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

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

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

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

Here, the storage to which the pointer exception object is referring may no longer exist when the exception is caught. (By the way, when a string literal is thrown, the entire array of characters is copied to the temporary, not just the address of the first character. This information is of little practical use, because we should never throw string literals. See Gotcha #64.) Additionally, a pointer may be null. Who needs this additional complexity? Don’t throw pointers, throw objects:

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

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

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

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

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

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

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

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

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

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

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

```c++
catch( StackUnderflow fault ) {
    // do partial recovery . . .
    fault.modifyState(); // my fault
    throw; // re-throw current exception
}
```
It’s not uncommon for a catch clause to perform a partial recovery, record the state of the recovery in the exception object, and re-throw the exception object for additional processing. Unfortunately, that’s not what’s happening here. This catch clause has performed a partial recovery, recorded the state of the recovery in a local copy of the exception object, and re-thrown the (unchanged) exception object.

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

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

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

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

```cpp
catch( ContainerFault &fault ) {
    // do partial recovery . . .
    if( !condition )
        fault.modifyState();
    throw;
}
```
Fortunately, the `ContainerFault` base class is abstract, so this particular manifestation of the error is not possible; in general, base classes should be abstract. Obviously, this advice doesn’t apply if you must throw an entirely different type of exception:

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

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

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

catch( StackUnderflow &fault ) {
    // . . .
}

catch( exception & ) {
    // . . .
}
```

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

The mechanics of exception handling offer much opportunity for complexity, but it’s not necessary to accept the offer. When throwing and catching exceptions, keep things simple.
**Gotcha #66: Abusing Local Addresses**

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

---

**Disappearing Stack Frames**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Idiomatic Difficulties

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

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

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

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

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

This is a particularly bad idea for overloaded operator functions:

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

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

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

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

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

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

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

The Static Fix

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

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

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

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

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

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

char buf[MAX];
long count = 0;
// . . .
Gotcha #67: Failure to Employ Resource Acquisition Is Initialization

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

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

The order of execution of construction and destruction are mirror images of each other. When a class is constructed, the order of initialization is always the same: the virtual base class subobjects first (“in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes,” according to the standard), followed by the immediate base classes in the order of their appearance on the base-list in the class’s definition, followed by the non-static data members of the class, in the order of their declaration, followed by the body of the constructor. The destructor implements the reverse order: destructor body, members in the reverse order of their declarations, immediate base classes in the inverse order of their appearance, and virtual base classes. It’s helpful to think of construction as pushing a sequence onto a stack and destruction as popping the stack to implement the reverse sequence. The symmetry of construction and destruction is considered so important that all of a class’s constructors perform their initializations in the same sequence, even if their member initialization lists are written in different orders (see Gotcha #52).
As a side effect or result of initialization, a constructor gathers resources for the object’s use as the object is constructed. Often, the order in which these resources are seized is essential (for example, you have to lock the database before you write it; you have to get a file handle before you write to the file), and typically, the destructor has the job of releasing these resources in the inverse order in which they were seized. That there may be many constructors but only a single destructor implies that all constructors must execute their component initializations in the same sequence.

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

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

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

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

```c++
Trace a( "global" );
void loopy( int cond1, int cond2 ) {
    Trace b( "function body" );
    it: Trace c( "later in body" );
    if( cond1 == cond2 )
        return;
```
if( cond1-1 ) {
    Trace d( "if" );
    static Trace stat( "local static" );
    while( --cond1 ) {
        Trace e( "loop" );
        if( cond1 == cond2 )
            goto it;
    }
    Trace f( "after loop" );
}
Trace g( "after if" );

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

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

The messages show clearly how the lifetime of a Trace object is associated with the current scope of execution. In particular, note the effect the goto and return have on the lifetimes of the active Trace objects. Neither of these branches is exemplary coding practice, but they’re the kinds of constructs that tend to appear as code is maintained.
void doDB() {
    lockDB();
    // do stuff with database . . .
    unlockDB();
}

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

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

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

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

void doDB() {
    lockDB();
    try {
        // do stuff with database . . .
    }
    catch( . . . ) {
        unlockDB();
        throw;
    }
    unlockDB();
}
This approach is wordy, low-tech, slow, hard to maintain, and will cause you to be mistaken for a member of the Department of Redundancy Department. Properly written, exception-safe code usually employs few try blocks. Instead, it uses resource acquisition is initialization:

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

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

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

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

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

The declaration of lock1 is correct; it’s a DBLock object that comes into scope just before the terminating semicolon of the declaration and goes out of scope at the end of the block that contains its declaration (in this case, at the end of the function). The declaration of lock2 declares it to be a function that takes no argument and returns a DBLock (see Gotcha #19). It’s not an error, but it’s probably not what was intended, since no locking or unlocking will be performed.
The following line is an expression-statement that creates an anonymous temporary DBLock object. This will indeed lock the database, but because the anonymous temporary goes out of scope at the end of the expression (just before the semicolon), the database will be immediately unlocked. Probably not what you want.

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

**Gotcha #68: Improper Use of auto_ptr**

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

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

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

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

```cpp
void calc( double src[], int len ) {
  double *tmp = new double[len];
  // . . .
  delete [] tmp;
}
```
The `calc` function is fragile, in that the allocated `tmp` array will not be recovered in the event that an exception occurs during execution of the function or if improper maintenance causes an early exit from the function. A resource handle is what’s required, and `auto_ptr` is our standard resource handle:

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

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

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

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

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

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

In one common outdated implementation of `auto_ptr`, its copy semantics are actually suitable for use as the element type of a container, and they can be used successfully. That is, until you get a different or newer version of the standard library, at which time your code will fail to compile. Very annoying, but usually a straightforward fix.
A worse situation occurs when the implementation of auto_ptr is not fully standard, so that it’s possible to use it to instantiate an STL container, but the copy semantics are not what is required by the STL. As described in Gotcha #10, copying an auto_ptr transfers control of the pointed-to object and sets the source of the copy to null:

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

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

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

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

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

, (comma operator), 39–40
?: (conditional operator), 15–16, 40–41
[ ] (allocating and deleting arrays), 35, 36, 168
( ) (allocating arrays), 35
-> (arrow operator), 58–60, 257–258
[ ] (index operator), 16–17
&& (logical operator), 40
|| (logical operator), 40
<<< (Sergeant operator), 48–49

aggregation vs. acquaintance, 253–258
Alexandrescu, Andrei, xv
algorithms, variant and invariant, 212–214
aliases
  aggregation/acquaintance relationships, 253
  references as, 10–13, 112–115
allocation failure, 171–173
ampersands (&&) logical operator, 40
anonymous namespace, 55
anonymous temporaries
  function object for pass by value, 109, 111
initialization of a reference formal argument, 107
initialize reference to const, 112
lifetime, 110–111
result of a postfix ++ or --, 266
as a result of copy initialization, 153–154
throwing, 181–182
array names vs. constant pointers, 87
arrays
  of arrays, 52, 87–88
  of class objects, 271–273
confused with initializers, 35–36
freeing, 167–170
migrating type-qualifiers, 52
pointer-to-multidimensional array conversions, 87–88
references to, 12
vs. vectors, 168

A
access protection
  Bridge pattern, 21–22
  vs. data abstraction, 241–242
description, 19
inheritance, 23, 277–280
naming conventions, 23
vs. visibility, 19–23
accessor functions. See get/set interfaces.
ACM Code of Ethics..., 32
acquaintance vs. aggregation, 253–258
acronyms, 25–26
Acyclic Visitor pattern, 306
addresses
  arithmetic errors, 77–78, 89, 101, 286
  base class subobjects, 206–208
  members. See pointers, to members.
  of a non-lvalue return, 267
adolescent behavior, 31–33

309
arrow operator (->), 58–60, 257–258
assert macro, 72–74
assertions, side effects, 72–74
assignment vs. initialization, 125–129, 139–141
associativity
and precedence, 42
problems, 44–45
auto_ptr template, 28–29, 195–197

cast operators vs. conversion operators, 24–25
casting. See also void *. base class pointers to derived class
pointers. See downcasting.
incomplete types, 100–101
under multiple inheritance, 98–100, 147
non-dynamic casts. See static casts.
old-style casts, 102–103
to references, 12–13
reinterpret_cast, 76, 100–101
static casts, 103–105
casting away const, 252
casts, maintenance, 76, 84, 102, 103
catch clauses, ordering, 184
catching
exceptions, 182–183
string literals, 178–180
change logs, 2–3
Cheshire Cat technique. See Bridge pattern.
Clamage, Steve, xv
class design
aggregation vs. acquaintance, 253–258
const data members, 245–247
const member functions
casting away const, 252
meaning of, 250–252
semantics and mechanics, 249–250
syntax, 248
decrement operator, 264–268
friend vs. member operators, 262–264
get/set interfaces, 241–245
increment operator, 264–268
operator overloading, 258–264
operator precedence, 261–262
overloading, 258–262
reference data members, 245–247
templated copy operations, 268–270

B
base class subobject
addresses, 206–208
initialization, 142–147, 147–150, 218–219
base class types, arrays of class objects, 271–273
base language. See C++ base language.
battleship. in pencil cup, 295
binding
dynamic, 200
reference to function, 13
reference to lvalue, 11, 112–115
Bridge pattern, 21–22

C
C++ base language
conditional operator, 15–16
fallthrough, 17–19
index operator, 16–17
logical operators, 14–15
switch-statements, 17
calling a pure virtual function, 212, 220
capability queries, 302–306
class hierarchies. See hierarchy design.
class implementation, varying with #if, 70–71
class objects, bitwise copy, 136–138
classes
  access protection, 19–23
  interface, 145
  POD (Plain Old Data), 136
  pure virtual base, 24
  cleverness, unnecessary, 29–31
Cline, Marshall, xvi
code reuse, 281–285
coders. See programmers.
coding, conciseness, 2
Comeau, Greg, xv
comma (,) operator, 39–40
Command pattern, 281–285
comments. See also maintenance;
  readability.
  avoiding, 2–4
  change logs, 2–3
  excessive, 1–4
  fallthrough, 18
  maintaining, 1–4
  self-documenting code, 2
  specifying ownership, 254–255
compilation, avoiding recompilation, 21, 202–203
compiler-generated assignment of virtual base subobjects, 145
component coupling
  defeating access protection, 280
  global variables, 6
  polymorphism, 202
Composite pattern, 281, 283–284
computational constructors, 161
concrete public base classes, 285–286
conditional operator (?:), 15–16, 40
const data members, 245–247
const member functions
  casting away const, 252
  meaning of, 250–252
  semantics and mechanics, 249–250
  syntax, 248
const objects vs. literals, 13–14
const pointers
  vs. array names, 87
  definition, 50
  vs. pointer to const, 81
  pointer-to-const conversion, 81–82
const type-qualifier
  migrating, 52
  references, 10–11
constant-expressions, 67
constants
  assigning to, 246
  vs. literals, 4–6, 13–14
  const_cast operator, 103, 112, 246, 252
constructors
  calling virtual functions, 218–220
  computational, 161
  conversions, 95–98
  implementing with template member functions, 268–270
  initialization vs. assignment, 139–141
  initializing static members, 163–165
  virtual constructor idiom, 223
container substitutability, 273–277
containers of pointers, 255–258
contravariance, 120–123
conversion functions, explicit, 90, 136–138
conversion operators
  alternative to, 90
  ambiguous, 90–94
  diction, 24–25
  purpose of, 92
 conversions
  array names vs. constant pointers, 87
 casting
  incomplete types, 100–101
  multiple inheritance, 98–100
  under multiple inheritance, 98–100
  old-style casts, 102–103
  reinterpret_cast, 76, 100–101
  static casts, 103–105
 const pointers
  vs. array names, 87
  pointer-to-const conversion, 81–82
 const_cast operator, 103–105, 112, 246, 252
 contravariance, 120–123
 delta arithmetic
  casting incomplete types, 101
  class object addresses, 98–100
  correcting this value in virtual function call, 235–236
  downcasting, 89
 downcasting, 89
 dynamic_cast operator
  ambiguity, 116–120
  to ask a personal question of a type, 300
  for a capability query, 303–304
  of pointer to virtual base sub-object, 146
  in preference to static cast, 89
  static_cast in preference to, 146
 formal arguments
  passing by reference, 109
  passing by value, 108–109
  temporary initialization, 106–109
  functions for, 90–94
  implicit
    ambiguous results, 90–94
 constructor conversions, 95–98
 contravariance, 120–123
 from derived class to public base, 122
 initialization of formal arguments, 106–109
 references, 112–115
 initializing
  formal arguments, 106–109
  references, 112–115
 Meyers, Scott, 105
 objects, temporary lifetime, 110–111
 old-style casts, 102–103
 platform dependency, 76
 pointer-to-const conversion, 81–82
 pointer-to-multidimensional arrays, 87–88
 pointer-to-pointer-to-base conversion, 86–87
 pointer-to-pointer-to-const conversion, 82–86
 pointers
  converting, 82–86
  to incomplete class types, 100–101
  to members, converting, 120–123
  to pointers to derived classes, 86–87
 qualification conversions, 82–86
 references
  as aliases, 112–115
  conversions, 112–115
  to incomplete class types, 100–101
 reinterpret_cast, 76, 100–101
 slicing derived class objects, 79–81
 static casts, 103–105
 temporaries, 110–115. See also anonymous temporaries.
 void *, 75–78
 converting types. See casting; void *.
 copy constructor base, initializing, 147–150
copy operations
  denying, 135
idiom, 27–29
initialization, 132–136
templated, 268–270
cosmic hierarchies, 295–299
coupling. See component coupling.
covariant return types, 228
cowpath simile for natural language, 26
cross-cast, 304
cv-qualifiers. See const type-qualifier;
volatile type-qualifier.

D
dark corners (of the C++ language)
  address of a non-lvalue return, 267
calling a pure virtual function, 212, 220
  compiler-generated assignment of
    virtual base subobjects, 145
  dynamic scope of invocation member
    operator delete, 206
dynamic_cast to inaccessible base, 177
  guarantees associated with
    reinterpret_cast, 100
  ignoring qualifiers on reference type
    name, 10
  indexing an integer, 16
  lvalue result of conditional operator, 16
  overriding invisible functions, 228
  point of declaration of an
    enumerator, 54
  qualification of function typedef, 52
  string literal temporary for throw
    expression, 181
  switch-statement structure, 18
data abstraction
  for exception types, 178
  purpose of, 241
data hiding. See access protection.
debug code, in executable modules, 67
debugging
  #if, 66–69
  unreachable code, 67–69
declaration-specifiers, ordering, 50–51
  Decorator pattern, 212, 281
decrement operator, 264–268
default argument initializer. See default
  initialization.
default initialization
  vs. overloading, 8–9
  uses for, 7, 9
  and virtual base objects, 244
#define, and namespace, 62
#define literals, 61–63
#define pseudofunctions, 64–66
degenerate hierarchies, 286
delete [] operator, 168–170
  array allocation, 168–170
  replacing, 173–176
  scalar allocation, 168–170
  scope and activation, 176–177
delta arithmetic
  casting incomplete types, 101
  class object addresses, 98–100
  correcting this value in virtual
    function call, 235–236
downcasting, 89
derived class objects, slicing, 79–81
derived classes, overriding functions,
  228–229
design firewalls, 202
destroyed vs. destructed, 24
destructors, calling virtual functions,
  218–220
developers. See programmers.
Dewhurst, David, xv
diction. See also idiom.
acronyms, 25–26
cast operators, 24–25
conversion operators, 24–25
destroyed vs. destroyed, 24
function calls, 24
member functions, 24
methods, 24
null pointers, 25
pure virtual base classes, 24
direct argument initialization, 156–158
dominance, 236–239
downcasting, 89
dynamic binding, 200
dynamic scope of invocation member
operator delete, 206
dynamic_cast operator
ambiguity, 116–120
to ask a personal question of a type, 300
for a capability query, 303–304
conversions, 116–120
of pointer to virtual base subobject, 146
in preference to static cast, 89
static_cast in preference to, 146
virtual base default initialization, 146
dynamic_cast to inaccessible base, 177

E
Eiffel, 29
The Elements of Style, 26
enumerators
#define literals, 63
initializing static members, 163
magic numbers, 5
point of declaration, 54
ethics of programming, 32–33
evaluation order. See precedence.
examples, source code, xiv
exception handling, 172–173, 177–184,
193–194
exception types, 178–180
extern types, 55

F
Factory Method pattern, 229, 274–275
fallthrough, 17–19
for statement
variable scope restriction, 45–48
vs. while statement, 47
formal arguments
passing by reference, 109
passing by value, 108–109
specifying ownership, 254–255
temporary initialization, 106–109
forward class declaration. See incomplete
declaration.
free vs. delete, 168–169
freeing
arrays and scalars, 167–170
resources of heap-allocated objects, 195
friend vs. member operators, 262–264
function matching. See overloading.
function/object ambiguity, 51
function object idiom, 282
function typedef, qualification, 52
functions
binding references to, 13
for conversions, 90–94
invisible, overriding, 228
pointers to, 13
references for return values, 11–12
references to, 13
G

Gamma, Erich, xi
get/set interfaces, 241–245
global variables, 6–8
gotchas, definition, xi
Gschwind, Thomas, xv
guarantees associated with
  reinterpret_cast, 100

value semantics, 286
hyphen angle bracket (->) operator,
  58–60

I

idiom. See also diction.
  auto_ptr template, 28–29
copy operation, 27–29
function object, 282
natural language, as a cowpath, 26
and natural selection, 27
resource acquisition is initialization, 28
rules of natural language, 26
smart pointer, 59, 282
virtual constructor, 223. See also
  Prototype pattern.
#if
  debugging, 66–69
platform independence, 69
portability, 69–70
in the real world, 71–72
varying class implementation, 70–71
ignoring qualifiers on reference type
  name, 10
implicit conversions
  ambiguous results, 90–94
  constructor conversions, 95–98
  contravariance, 120–123
  from derived class to public base, 122
  initialization of formal arguments,
    106–109
  references, 112–115
incomplete declaration, 20–21
incomplete types
  casting, 100–101
  for decoupling, 20–21

header files
  reference counting inclusions, 152–153
  Schwarz counter, 152–153
Hewins, Sarah, xv
hiding
  nonvirtual functions, 209–212
  vs. overloading and overriding, 224–230
hierarchy design
  address arithmetic errors, 286
  arrays of class objects, 271–273
capability queries, 302–306
code reuse, 281–285
concrete public base classes, 285–286
container substitutability, 273–277
cosmic hierarchies, 295–299
degenerate hierarchies, 286
inheritance, 287–292
interface classes, 281
protected access, 277–280
protocol classes, 281
public inheritance, 281–285
runtime type queries, 299–301
slicing, 286
switching on type codes, 292–295
type-based control structures, 292–295

hyphen angle bracket (->) operator,
  58–60

Index
increment operator, 264–268
index operator, predefined, 16–17
index operator ([ ]), 16–17
indexing
  array names, 16
  integers with pointers, 16–17
  pointers, 16
infix notation, 56–58, 258–259
inheritance
  access protection, 23
  hierarchy design, 281–285, 287–292
initialization
  vs. assignment, 125–129, 139–141
  bitwise copy of class objects, 136–138
  copy constructor base, 147–150
  copy operations, 132–136
default
  vs. overloading, 8–9
  uses for, 7, 9
direct argument, 156–158
direct vs. copy, 153–156
formal arguments, temporary, 106–109
implicit copy operations
  bitwise copy of class objects, 136–138
  description, 132–136
initializers, confused with arrays, 35–36
member initialization list, ordering,
  141–142
passing arguments, 126
references, 112–115
return value optimizations, 158–162
runtime static, ordering, 150–153
scoping variables, 129–132
self-initialization, 53–55
Singleton pattern, 7–8
static members in constructors,
  163–165
  virtual base default, 142–147
initializers, confused with arrays, 35–36
integers, indexing, 16
  See also mix-in classes.

Josuttis, Nicolai, xvi

Kernighan, Brian, xv

Lafferty, Debbie, xv
language (natural), 26
left angle brackets (<<), Sergeant
  operator, 48–49
lexical analysis, 49
literals
  vs. const objects, 13–14
  vs. constants, 4–6
defining, 61–63
local addresses
  disappearing stack frames, 185
  idiomatic problems, 186–187
  static interference, 186
  local scope problems, 187
  local variable lifetimes, 187
logical operators, 14–15, 40
lvalue
  binding references, 11, 112–115
definition, 13–14
function return, 11
initializing references. See binding, reference to lvalue.
nonmodifiable, 14, 62
result of conditional operator, 15–16

<table>
<thead>
<tr>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>macros, side effects, 16, 64–66</td>
</tr>
<tr>
<td>magic numbers, 4–6</td>
</tr>
<tr>
<td>maintenance. See also comments; readability.</td>
</tr>
<tr>
<td>and casts, 76, 84, 102, 103</td>
</tr>
<tr>
<td>easing assertions, 73</td>
</tr>
<tr>
<td>coding standards, 32, 40–41</td>
</tr>
<tr>
<td>container ownership, 255</td>
</tr>
<tr>
<td>container substitutability, 275</td>
</tr>
<tr>
<td>declaration-specifier ordering, 51</td>
</tr>
<tr>
<td>fallthrough, 17–18</td>
</tr>
<tr>
<td>idioms, 27</td>
</tr>
<tr>
<td>incomplete declarations, 20</td>
</tr>
<tr>
<td>initializing static members, 164</td>
</tr>
<tr>
<td>mnemonic names, 3</td>
</tr>
<tr>
<td>naming conventions, 3</td>
</tr>
<tr>
<td>precedence, 43–44</td>
</tr>
<tr>
<td>scalar allocation vs. array, 170</td>
</tr>
<tr>
<td>for statement, 45, 48</td>
</tr>
<tr>
<td>type codes, 202</td>
</tr>
<tr>
<td>typed-base control structures, 293–294</td>
</tr>
<tr>
<td>made difficult asking personal questions of objects, 299</td>
</tr>
<tr>
<td>auto_ptr, 196</td>
</tr>
<tr>
<td>code reuse, 281, 285</td>
</tr>
<tr>
<td>comments, 1–4</td>
</tr>
<tr>
<td>const and reference data members, 245</td>
</tr>
<tr>
<td>dynamic_cast, 116</td>
</tr>
<tr>
<td>global variables, 6–8</td>
</tr>
<tr>
<td>implicit conversions, 92</td>
</tr>
<tr>
<td>memory allocation failure, 171</td>
</tr>
<tr>
<td>miscommunication of container ownership, 256</td>
</tr>
<tr>
<td>naming conventions, 23</td>
</tr>
<tr>
<td>overloading virtual functions, 215</td>
</tr>
<tr>
<td>public inheritance, 281, 285</td>
</tr>
<tr>
<td>resource acquisition is initialization, 193–194</td>
</tr>
<tr>
<td>switch on type codes, 200</td>
</tr>
<tr>
<td>throwing/catching string literals, 178</td>
</tr>
<tr>
<td>type-based control structures, 293</td>
</tr>
<tr>
<td>unnecessary cleverness, 29–31, 115</td>
</tr>
<tr>
<td>made impossible capability queries, 304</td>
</tr>
<tr>
<td>cosmic hierarchies, 296–297</td>
</tr>
<tr>
<td>#if, 69, 71</td>
</tr>
<tr>
<td>initializing static members, 165</td>
</tr>
<tr>
<td>local address abuse, 185</td>
</tr>
<tr>
<td>platform dependence, 69</td>
</tr>
<tr>
<td>static interference, 186</td>
</tr>
<tr>
<td>varying class implementation, 71</td>
</tr>
<tr>
<td>remote changes (bugs caused by), 77, 101, 103, 105</td>
</tr>
<tr>
<td>malloc vs. new, 168–169</td>
</tr>
<tr>
<td>maximal munch description, 48–49</td>
</tr>
<tr>
<td>examples, 30, 48–49</td>
</tr>
<tr>
<td>McKillen, Patrick, xv</td>
</tr>
<tr>
<td>meaningful names. See naming conventions.</td>
</tr>
<tr>
<td>member functions diction, 24</td>
</tr>
<tr>
<td>template, 29</td>
</tr>
<tr>
<td>virtual static, 205–206</td>
</tr>
<tr>
<td>member initialization list, ordering, 29, 141–142</td>
</tr>
</tbody>
</table>
member vs. friend operators, 262–264
members
  pointers to, 9, 120–123
  requiring initialization, 139
memory and resource management
  allocation failure, 171–173
  auto_ptr, 195–197
catch clauses, ordering, 184
catching exceptions, 182–183
catching string literals, 178–180
exception handling, 177–184
exception types, 178–180
freeing
  arrays and scalars, 167–170
  resources of heap-allocated objects, 195
local addresses
  disappearing stack frames, 185
  idiomatic problems, 186–187
  static interference, 186
local scope problems, 187
local variable lifetimes, 187
memory leaks, 187, 253
replacing global new and delete,
  173–176
resource acquisition is initialization,
  190–195
scalar vs. array allocation, 167–170
scope and activation, new and delete,
  176–177
static fix, 188–190
throwing anonymous temporaries,
  181–182
throwing pointers, 181
throwing string literals, 177–180
memory leaks, 187, 253–258
methods, 24
Meyers, Scott, xvi, 105
migrating type-qualifiers, 52
mix-in classes, 281. See also interface
classes.
mnemonic names, 3
Monostate pattern, 203–204
multiple inheritance, casting, 98–100

N
named return value optimization
  (NRV), 161
namespace
  anonymous, 55
  and #define, 62
naming conventions
  access protection, 23
  mnemonic names, 3
  self-documenting code, 3
  simplicity, 23
  specifying ownership, 254–255
  variable type in variable name, 23
NDEBUG, mysterious failures, 67
nonvirtual base class destructor
  addresses of base class subobjects,
  206–208
  exceptions, 208–209
  undefined behavior, 205
  virtual static member functions,
  205–206
NRV (named return value optimization), 161
null
  dynamic_cast result, 117
  pointers, 25
  references, 11
Null Object pattern, 282, 284, 294
numeric literals vs. constants, 4–6
O

object-oriented to non-object-oriented communication, 202
objects, temporary lifetime, 110–111
old-style casts, 102–103, 278
Oldham, Jeffrey, xv
operator delete, 206
operator new
  allocation failure, 172
  replacing, 173–176
  scalar allocation, 168–170
  scope and activation, 176–177
operator overloading, 258–264
operator precedence, 261–262
operators
  , (comma operator), 39–40
  ?: (conditional operator), 40
  -> (arrow operator), 58–60
  && (logical operator), 40
  || (logical operator), 40
  <<<< (Sergeant operator), 48–49
C++ base language, 14–17
cast, 24–25
cast vs. conversion, 24–25
conversion, 24–25, 90–94
evaluation order, 39–41
function lookup, 56–58
index operator, predefined, 16–17
logical, 14–15
lvalue, result of conditional operator, 15–16
new [] operator, 39
operator function lookup, 56–58
overloading
  -> (arrow operator), 58–60
  evaluation order, 41
  operator function lookup, 56–58
  operators, 56–58
  precedence, 41
precedence, 17, 39–40
predefined index operator, 16–17
overloading
  -> (arrow operator), 58–60
ambiguities, 5
in class design, 258–262
vs. default initialization, 8–9
vs. hiding and overriding, 224–230
increment/decrement operators, 264
infix notation, 56–58
operators, 56–58, 258–264
virtual functions, 214–215
overriding
definition, 232
invisible functions, 228
mechanisms, 230–236
vs. overloading and hiding, 224–230
ownership. See aggregation vs. acquaintance.

P

parallel hierarchies, 274–275
parentheses ( ), allocating arrays, 35
passing arguments, 126
patterns
  Acyclic Visitor, 306
  Bridge, 21–22
  Command, 281–285
  Composite, 281, 283–284
  Decorator, 212, 281
  Factory Method, 229, 274–275
  Monostate, 203–204
  Null Object, 282, 284, 294
  Prototype, 223, 229, 282–283
  Proxy, 294
  Singleton, 7–8, 152, 204
  Strategy, 291–292
Template Method, 212–214
Visitor, 215, 227, 306
pencil cup, battleship in, 295
personal questions (about an object’s
type), 116, 200, 275, 299–301
pimpl idiom. See Bridge pattern.
placement new
evaluation order of arguments, 39
invoking constructor, 127, 138, 155
replacing global new and delete, 174
platform dependence
conversions, 76
literals vs. constants, 5
POD (Plain Old Data) classes, 136
point of declaration of an enumerator, 54
pointer formal arguments, 49
pointer-to-const conversion, 81–82
pointer-to-multidimensional array, 87–88
pointer-to-pointer-to-base conversion,
86–87
pointer-to-pointer-to-const conversion,
82–86
pointers
containers of, 255–258
converting, 82–86
to functions, 13
to incomplete class types, 100–101
to local variables, 185
to members, 9
to members, converting, 120–123
ownership, 255–258
to pointers to derived classes, 86–87
precedence problems, 43–44
vs. references, 10–13
throwing, 181
polymorphism
algorithms, variant and invariant,
212–214
component coupling, 202
design firewalls, 202
dominance, 236–239
dynamic binding, 200
flexibility of template methods, 212–214
hiding
nonvirtual functions, 209–212
vs. overloading and overriding,
224–230
nonvirtual base class destructor
addresses of base class subobjects,
206–208
exceptions, 208–209
undefined behavior, 205
virtual static member functions,
205–206
object-oriented to non-object-oriented
communication, 202
overloading
vs. hiding and overriding, 224–230
virtual functions, 214–215. See also
default initialization.
overriding
definition, 232
mechanism, 230–236
vs. overloading and hiding, 224–230
switching on type codes, 200
type codes, 199–204
virtual assignment, 220–224
virtual copy construction, 223
virtual functions
calling in a nonvirtual manner, 211
calling in constructors and
destructors, 218–220
default argument initializers,
216–217. See also overloading,
virtual functions.
overloading, 214–215. See also default
initialization.
Index

portability
  #if, 69–70
  null pointers, 25
precedence
  , (comma operator), 39–40
  ?: (conditional operator), 40
  && (logical operator), 40
  || (logical operator), 40
  and associativity, 42, 44–45
  fixing, 39–41
  index operators, 17
  levels of precedence, 42
  new [] operator, 39
  operator overloading, 41
  operators, 39–40, 261–262
  overview, 36–37
  pointers, 43–44
predefined index operator, 16–17
preprocessor
  assert macro, 72–74
  assertions, side effects, 72–74
  class implementation, varying with #if, 70–71
  constant-expressions, 67
  debug code, in executable modules, 67
  debugging, 66–69
  #define literals, 61–63
  #define pseudofunctions, 64–66
  #if
    debugging, 66–69
    platform independence, 69
    portability, 69–70
    in the real world, 71–72
    varying class implementation, 70–71
  literals, defining, 61–63
  NDEBUG, mysterious failures, 67
  pseudofunctions, defining, 64–66
  scope, #define literals, 61–63
  preprocessor macros, side effects, 16
programmers
  adolescent behavior, 31–33
  ethical duties, 32–33
  unnecessary cleverness, 29–31
  Programming in C++, 3
  protected access, 277–280
  protocol classes, 281. See also interface classes.
  Prototype pattern, 223, 229, 282–283
  Proxy pattern, 294
  pseudofunctions, defining, 64–66
  public inheritance, 281–285
  pure virtual base classes, 24
  pure virtual functions, calling, 212

Q

qualification conversions, 82–86
qualification of function typedef, 52
question mark colon (?:) conditional operator, 15–16, 40

R

readability. See also comments; maintenance.
  formatting code, 29–31
  unnecessary cleverness, 29–31
  recompilation, avoiding, 21, 283
  reference counting, 257
  reference counting inclusions, 152–153.
  See also Schwarz counter.
  reference data members, 245–247
  reference type name, ignoring qualifiers, 10
  references
    as aliases, 10–13, 112–115
to arrays, 12
binding to functions, 13
binding to lvalue, 11
casting objects, 12–13
const type-qualifier, 10–11
conversions, 112–115
to functions, 13
to incomplete class types, 100–101
initializing, 112–115
to local variables, 185
null, 11
vs. pointers, 10–13
return values for functions, 11–12
underusing, 10–13
volatile qualifiers, 10–11
reinterpret_cast, 76, 100–101, 146
remote changes (bugs caused by), 77, 101, 103, 105
resource acquisition is initialization, 28, 190–195
resource handle. See resource acquisition is initialization.
resource management. See memory and resource management.
resources, freeing heap-allocated objects, 195
return value optimization (RVO), 158–162
reuse
code, 281–285
global variables, 6
white-box. See inheritance.
runtime static initialization, ordering, 150–153
runtime type queries, 299–301
RVO (return value optimization), 158–162

S

Saks, Dan, xv, xvi
scalars, freeing, 167–170
Schwarz, Jerry, 152
Schwarz counter, 152–153. See also reference counting inclusions.
scope
#define literals, 61–63
local scope problems, 187
restriction, variables, 45–48
scoping variables, initialization, 129–132
self-documenting code
avoiding comments, 2
naming conventions, 3
self-initialization, 53–55
Semantics, xiv
Sergeant operator (<<<), 48–49
set/get interfaces, 241–245
Singleton pattern, 7–8, 152, 204
slicing
derived class objects, 79–81
hierarchy design, 286
smart pointer idiom, 59, 282
social commentary
adolescent behavior, 31–33
array/initializer confusion, 36
capability queries, 302
increment/decrement operators, 267
old-style casts, 102
operator overloading, 258
personal questions of an object, 299–300
unnecessary cleverness, 29
Software Engineering Code of Ethics..., 32
square brackets ([ ]), allocating arrays, 35
Stark, Kathy, 3
static casts, 103–105, 278
static members in constructors,
   initializing, 163–165
static types, 55
static variables, runtime static initialization
   problems, 151
static_cast, 103–105, 278
Strategy pattern, 291–292
string literal temporary for throw
   expression, 181
string literals, throwing, 177–181
Stroustrup, Bjarne, 295
Strunk, William, 26
subexpressions, evaluation order, 37–39
Sutter, Herb, xvi
switch-statement structure, 18
switch-statements, 17
switching on type codes, 200, 292–295
syntax
   arrays
      confused with initializers, 35–36
      migrating type-qualifiers, 52
   associativity
      and precedence, 42
      problems, 44–45
   const member functions, 248
   const pointers, 50
   const type-qualifier, migrating, 52
   declaration-specifiers, ordering, 50–51
   evaluation order
      , (comma operator), 39–40
      ?: (conditional operator), 40
      && (logical operator), 40
      || (logical operator), 40
      fixing, 39–41
      new operator, 39
      operator overloading, 41
   overview, 36–37
   placement new, 39
   subexpressions, 37–39
   extern types, 55
   function/object ambiguity, 51
   infix notation, 56–58
   initialization
      initializers, confused with arrays,
         35–36
      self-initialization, 53–55
   initializers, confused with arrays, 35–36
   lexical analysis, 49
   maximal munch, 48–49
   migrating type-qualifiers, 52
   new [] operator, 39
   operator function lookup, 56–58
   operator overloading
      evaluation order, 41
      operator function lookup, 56–58
   overloading
      -> (arrow operator), 58–60
      infix notation, 56–58
      operators, 56–58
   placement new, evaluation order, 39
   pointer formal arguments, 49
   pointers, precedence problems, 43–44
   precedence problems
      and associativity, 42, 44–45
      levels of precedence, 42
      pointers, 43–44
   scope restriction, variables, 45–48
   self-initialization, 53–55
   for statement
      variable scope restriction, 45–48
      vs. while statement, 47
   static types, 55
   subexpressions, evaluation order, 37–39
   templates, instantiating, 49
token identification, 48–49
type-qualifiers, migrating, 52
types, linkage-specifiers, 55
variables, scope restriction, 45–48
volatile type-qualifiers, migrating, 52
while statement vs. for statement, 47

U
unnecessary cleverness, 29–31
uses. See acquaintance.

V
value semantics, 286
variables
 encoding type in name, 23
 scope restriction, 45–48
 vectors vs. arrays, 36, 168, 196
 vertical lines (||) logical operator, 40
 virtual assignment, 220–224
 virtual base default, initializing,
 142–147
 virtual constructor idiom, 223. See also
 Prototype pattern.
 virtual copy construction, 223
 virtual functions
  calling in a nonvirtual manner, 211
  calling in constructors and destructors, 218–220
  default argument initializers, 216–217
  overloading, 214–215
  pure, calling, 212
 virtual static member functions,
 205–206
 visibility vs. access protection, 19–23
 Visitor pattern, 215, 227, 306
 void *, 75–78. See also casting.
 volatile type-qualifier
 migrating, 52
 references, 10–11
 vptr (pointer to a vtbl), 231
 vtbl (virtual function table), 231–236
W

while statement vs. for statement, 47
White, E.B., 26
white-box reuse. See inheritance.
Wilson, Matthew, xv
word choice. See diction.
Writings from The New Yorker, 26

Z

Zolman, Leor, xv