

# Errata List for *Exceptional C++*

Sutter, H. *Exceptional C++* (Addison-Wesley, 2000) — ISBN 0-201-61562-2

Updated 2000.12.12

This errata list is maintained by the author. To suggest changes or corrections not already in this list, please submit them by email to [hsutter@peerdirect.com](mailto:hsutter@peerdirect.com) with a subject line containing the words “XC++ Errata.”

| Severity Category   | # Entries |
|---|-----------|
| <b>Format</b><br>(change to page layout or text formatting only)  | 3         |
| <b>Typo</b><br>(correction of simple typographical errors, cut-and-paste errors, and dyslexic mistakes)                                       | 28        |
| <b>Enhancement</b><br>(addition of new or clarifying material)  | 22        |
| <b>Correction</b><br>(change made to correct a substantive error that could mislead a reader; does not include typos and occasional dyslexia) | 13        |

The individual errata entries are listed in page number order. For each one, I have included the page number (including “xref” cross-references to related entries for other pages), the severity (summarized above), the person who first reported the erratum and when, the earliest printing incorporating the correction, and a description of the erratum and its correction.

| Page | Severity    | First Reported |  | Corrected<br>Printing # | Description   |
|------|-------------|----------------|--|-------------------------|---|
|      |             | Date           | By   |                         |   |
| xii  | Enhancement | 2000.04.14     | Michel Michaud<br>micm19@mail2.csjjean.qc.ca   | —                       | <p>This book includes many guidelines, and in them I use the terms “always,” “prefer,” “consider,” “avoid,” and “never” with specific meanings. Those meanings were clearly explained in the coding standards appendix, but that appendix was held over to the next book and I never duplicated the explanations in the existing book.</p> <p>Immediately before the subhead “How We Got Here: <i>GotW</i> and <i>PeerDirect</i>,” add the following new paragraph:</p> <p>This book includes many guidelines, In which the following words usually carry a specific meaning:</p> <ul style="list-style-type: none"> <li>· <b>always</b> = This is absolutely necessary. Never fail to do this.</li> <li>· <b>prefer</b> = This is usually the right way. Do it another way only when a situation specifically warrants it.</li> <li>· <b>consider</b> = This may or may not apply, but it's something to think about.</li> <li>· <b>avoid</b> = This is usually not the best way, and might even be dangerous. Look for alternatives, and do it this way only when a situation specifically warrants it.</li> <li>· <b>never</b> = This is extremely bad. Don't even think about it. Career limiting move.</li> </ul>  |
| 9    | Enhancement | 2000.10.31     | David X. Calloway<br>dxc@xprt.net              | —                       | <p>Change:</p> <p>There are two ways to resolve this: Define insertion (<code>operator&lt;&lt;()</code>) and extraction (<code>operator&gt;&gt;()</code>) for <code>ci_strings</code> yourself, or tack on <code>“.c_str()”</code> to use <code>operator&lt;&lt;( const char* )</code>:</p> <p>To:</p> <p>There are two ways to resolve this: Define <code>operator&lt;&lt;()</code> and <code>operator&gt;&gt;()</code> for <code>ci_strings</code> yourself, or tack on <code>“.c_str()”</code> to use <code>operator&lt;&lt;( const char* )</code> if your application's strings don't have embedded nulls:</p>  |
| 15   | Enhancement | 2000.08.13     | Howard Hinnant<br>hinnant@metrowerks.com       | —                       | <p>In Item 5, the discussion of <code>fixed_vector</code>'s templated assignment operator shows how to make it satisfy the strong exception-safety guarantee. Unfortunately, because this discussion comes before the discussion of the various exception safety guarantees it might be taken to imply that <code>fixed_vector</code> isn't exception-safe at all, which isn't true — it does provide the basic guarantee. This is an artifact of the Items being reordered into sections: Item 5 (<i>GotW</i> #16) was originally written after Items 8 to 17 (<i>GotW</i> #8), and now the context needs to be pointed at better. Here's a quick 'fix.'</p> <p>In the final paragraph, change:</p> <p>Alas, it does. Did you notice that the templated assignment operator is not strongly exception-safe? Recall that it was defined as:</p> <p>To:</p> <p>Perhaps. Later in this book we'll distinguish between various exception safety guarantees (see Items 8 to 11, and page 38). Like the compiler-generated copy assignment operator, our templated assignment operator provides the basic guarantee, which can be perfectly fine. Just for a moment, though, let's explore what happens if we do want it to provide the strong guarantee, to make it <i>strongly</i> exception-safe. Recall that the templated assignment operator was defined as:</p> |
| 15   | Correction  | 2000.08.12     | Burkhard Kloss<br>bkloss@novallis2.demon.co.uk | —                       | <p>The example code using <code>std::copy()</code> tries to copy a range of six objects into a target that's only large enough to hold four objects. We should only copy four objects.</p> <p>Change:</p> <pre>copy( v.begin(), v.end(), w.begin() );</pre> <p>To:</p> <pre>copy( v.begin(), v.begin()+4, w.begin() );</pre>  |

| Page  | Severity    | First Reported |  | Corrected Printing # | Description  |
|-------|-------------|----------------|--|----------------------|--|
|       |             | Date           | By   |                      |  |
| 16-7  | Correction  | 1999.12.28     | Tim Butler<br>tim@indra.com                    | 2                    | The strongly exception-safe version now requires an explicitly written copy constructor and copy assignment operator, implemented like the templated versions. Add these functions.  |
|       |             | 2000.02.21     | Klaus Ahrens<br>ahrens@informatik.hu-berlin.de | 2                    | Also, the non-default constructor is missing a memory allocation.  |
|       |             | 2000.08.13     | Howard Hinnant<br>hinnant@metrowerks.com       | —                    | Also, the original fix in printing #2 had another bug — a memory leak if an exception occurs during the <code>copy()</code> — for which the simplest refix here is to wrap the <code>copy()</code> in a <code>try/catch</code> .<br><br>Change:<br><pre>template&lt;typename O, size_t osize&gt; fixed_vector( const fixed_vector&lt;O,osize&gt;&amp; other ) {     copy( other.begin(),           other.begin()+min(size,osize),           begin() ); } </pre> To:<br><pre>template&lt;typename O, size_t osize&gt; fixed_vector( const fixed_vector&lt;O,osize&gt;&amp; other ) : v_( new T[size] ) { try {copy(other.begin(), other.begin()+min(size,osize), begin());}   catch(...) { delete[] v_; throw; } } fixed_vector( const fixed_vector&lt;T,size&gt;&amp; other ) : v_( new T[size] ) { try {copy(other.begin(), other.begin()+min(size,osize), begin());}   catch(...) { delete[] v_; throw; } } </pre> And change:<br><pre>template&lt;typename O, size_t osize&gt; fixed_vector&lt;T,size&gt;&amp; operator=( const fixed_vector&lt;O,osize&gt;&amp; other ) {     fixed_vector&lt;T,size&gt; temp( other ); // does all the work     Swap( temp ); // this can't throw     return *this; } </pre> To:<br><pre>template&lt;typename O, size_t osize&gt; fixed_vector&lt;T,size&gt;&amp; operator=( const fixed_vector&lt;O,osize&gt;&amp; other ) {     fixed_vector&lt;T,size&gt; temp( other ); // does all the work     Swap( temp ); return *this; // this can't throw } fixed_vector&lt;T,size&gt;&amp; operator=( const fixed_vector&lt;T,size&gt;&amp; other ) {     fixed_vector&lt;T,size&gt; temp( other ); // does all the work     Swap( temp ); return *this; // this can't throw } </pre> |
| 21    | Enhancement | 2000.10.01     | Thomas Petillon<br>petillon@topic.fr           | —                    | In the mid-page code example, the illustrated approach is of course only correct if the list is passed by reference. To make this clearer:<br><br>Change:<br><pre>const string&amp; FindAddr( /* ... */ ) </pre> To:<br><pre>const string&amp; FindAddr( /* pass emps and name by reference */ ) </pre> And change:<br><pre>if( /* found */ ) </pre> To:<br><pre>if( i-&gt;name == name ) </pre>   |
| 31    | Correction  | 2000.08.13     | Howard Hinnant<br>hinnant@metrowerks.com       | —                    | At the end of the paragraph numbered “2.” change:<br>...must be unchanged.<br><br>To:<br>...must be destructible.  |
| 32,33 | Typo        | 2000.07.23     | hps  | —                    | Somehow the first presented version of <code>Stack</code> has a member function called <code>Size()</code> instead of <code>Count()</code> . For consistency with the later versions, not to mention Cargill's original article, it should be <code>Count()</code> .   |

| Page             | Severity    | First Reported |  | Corrected<br>Printing # | Description  |
|------------------|-------------|----------------|--|-------------------------|--|
|                  |             | Date           | By   |                         |  |
|                  |             |                |  |                         | In one place on page 32 and four places on page 33, change:<br>Size<br>To:<br>Count  |
| 37               | Enhancement | 2000.01.25     | Marc Briand<br>mbriand@mfi.com   | 2                       | In the Common Mistake box, the wording should make it clearer that I'm criticizing code that <i>cannot be made</i> exception-safe because of the underlying design, not just code that happens to be incidentally exception-unsafe and only needs a local fix.   |
|                  | Correction  | 2000.08.24     | Andrew Koenig<br>ark@research.att.com<br><br>Bill Wade<br>wwade@swbell.net | —                       | Also, as Andy Koenig pointed out to me, it is possible to write a copy assignment operator that is written in a such way that it <i>must</i> check for self-assignment and yet is strongly exception-safe (or better). Consider a copy assignment operator that is written in such a way that it must test for self-assignment to work properly, yet uses only nonthrowing operations such as builtin/pointer operations — clearly it meets not just the strong guarantee, but even the nothrow guarantee! (Andy's example was of a class that implements an intrusive linked list, where assignment consists of removing the object from its current list and adding it to the other object's list; the obvious implementation requires a self-assignment check, yet uses only nonthrowing pointer operations.)<br><br>In the Guideline, change:<br><i>"Exception-unsafe" and "poor design" go hand in hand. If a piece of code cannot be made exception-safe, that almost always is a signal of its poor design. Example 1: A function with two different responsibilities is difficult to make exception-safe. Example 2: A copy assignment operator that has to check for self-assignment cannot be exception-safe.</i><br>To:<br><i>"Exception-unsafe" and "poor design" go hand in hand. If a piece of code isn't exception-safe, that's generally okay and can simply be fixed. But if a piece of code cannot be made exception-safe because of its underlying design, that almost always is a signal of its poor design. Example 1: A function with two different responsibilities is difficult to make exception-safe. Example 2: A copy assignment operator that is written in such a way that it must check for self-assignment is probably not strongly exception-safe either.</i><br><br>In the next paragraph, change:<br>... cannot be exception-safe.<br>To:<br>... is probably not strongly exception-safe. |
| 38               | Enhancement | 2000.02.10     | Dave Abrahams<br>abrahams@mediaone.net                                     | —                       | URL moved, <a href="http://www.metabyte.com/~fbp/stl/eh_contract.html">http://www.metabyte.com/~fbp/stl/eh_contract.html</a> is now <a href="http://www.stlport.org/doc/exception_safety.html">http://www.stlport.org/doc/exception_safety.html</a>  |
| 42               | Correction  | 2000.08.13     | Howard Hinnant<br>hinnant@metrowerks.com                                   | —                       | The box implies that the helper functions <code>construct()</code> and <code>destroy()</code> are standard, when they aren't.<br><br>In the first paragraph, change:<br>...use three helper functions that are directly drawn (or derived in spirit) from the standard library:<br>To:<br>...use three helper functions, one of which ( <code>swap()</code> ) also appears in the standard library:<br><br>Delete the final paragraph:<br>To find out more about these standard functions, take a few minutes to examine how they're written in the standard library implementation you're using. It's a worthwhile and enlightening exercise.   |
| 42,<br>55,<br>56 | Enhancement | 2000.03.25     | hps  | —                       | This didn't make a difference in any example in the book, but it's a little odd: The two-parameter <code>destroy(FwdIter, FwdIter)</code> version is templated to take any generic iterator, and yet it calls the one-parameter <code>destroy(T*)</code> by passing it one of the iterators... which requires that <code>FwdIter</code> must be a plain old pointer! This needlessly loses some of the   |

| Page      | Severity    | First Reported |                                     | Corrected<br>Printing # | Description  |
|-----------|-------------|----------------|-------------------------------------|-------------------------|--|
|           |             | Date           | By                                  |                         |  |
|           |             |                |                                     |                         | <p>generality of templating on <code>FwdIter</code>. A simple change lets <code>FwdIter</code> be pretty much any iterator type, not just a pointer: In <code>destroy(FwdIter, FwdIter)</code>, change the call <code>destroy( first )</code> to <code>destroy( &amp;*first )</code>. This will work in all cases, unless <code>T</code> provides an <code>operator&amp;()</code> that does not return a pointer which should occur rarely if ever.</p> <p>On pages 42, 55, and 56, in three places change the two-parameter version of <code>destroy()</code> as above.</p> <p>Change:<br/> <code>destroy( first );</code></p> <p>To:<br/> <code>destroy( &amp;*first );</code></p> <p>See also GotW #68 at <a href="http://www.peerdirect.com/resources">www.peerdirect.com/resources</a>.</p> |
| 43        | Typo        | 1999.12.23     | Steve Vinoski<br>vinoski@iona.com   | 2                       | <p>In paragraph 2, change:<br/> <code>StackImpl&lt;T&gt;</code></p> <p>To:<br/> <code>StackImpl&lt;T&gt;</code></p>  |
| 46,<br>58 | Typo        | 2000.05.24     | Sam Lindley<br>sam@redsnapper.net   | —                       | <p>In the Guideline, I say “initialization is resource acquisition” instead of “resource acquisition is initialization.”</p> <p>Change:<br/> <i>“initialization is resource acquisition”</i></p> <p>To:<br/> <i>“resource acquisition is initialization”</i></p>   |
| 48        | Enhancement | 2000.06.16     | Stan Brown<br>brahms@mindspring.com | —                       | <p>To make it more obvious that <code>other</code> is passed by value and hence already a temporary object, change <code>other</code> to <code>temp</code> and add more explanation.</p> <p>Change:<br/> If you’re one of those folks who like terse code, you can write the <code>operator=()</code> canonical form more compactly as:<br/> <code>Stack&amp; operator=(Stack other)</code><br/> <pre>{   Swap( other );   return *this; }</pre></p> <p>To:<br/> If you’re one of those folks who like terse code, you can write the <code>operator=()</code> canonical form more compactly using pass-by-value to create the temporary:<br/> <code>Stack&amp; operator=(Stack temp)</code><br/> <pre>{   Swap( temp );   return *this; }</pre></p>  |
| 49        | Enhancement | 2000.02.21     | hps                                 | —                       | <p>In the paragraph following the bullets, it talks about ‘if we allowed iterators,’ but note that we do allow taking a reference into the container (via <code>Top()</code>) which is much the same thing.</p> <p>Change:<br/> If we were supporting iterators into this container, for instance, they would never be invalidated (by a possible internal grow operation) if the insertion is not completely successful.</p> <p>To:<br/> Any references returned from <code>Top()</code>, or iterators if we later chose to provide them, would never be invalidated (by a possible internal grow operation) if the insertion is not completely successful.</p>   |

| Page   | Severity    | First Reported |   | Corrected Printing # | Description  |
|--------|-------------|----------------|---|----------------------|--|
|        |             | Date           | By  |                      |  |
| 57, 58 | Format      | 2000.07.24     | hps   | —                    | In the two Guidelines, the word “overloaded” should not be in code font.   |
| 64     | Typo        | 1999.12.13     | Jon Kalb<br>kalb@libertysoft.com              | 2                    | In paragraph 2, “addtion” should be “addition.”  |
| 70     | Typo        | 2000.01.06     | Douglas Gilbert<br>dgilbert@724.com           | 2                    | In the last line, “Item 19” should be “Item 39.”   |
| 72     | Correction  | 2000.01.16     | Eric Nagler<br>epn@eric-nagler.com            | 2                    | In the second Guideline box, compound assignment operators need not be members.  |
|        |             | 2000.01.24     | hps   | 2                    | Also, these rules should be revised based on Scott Meyers' article “How Non-Member Functions Improve Encapsulation” ( <i>C/C++ Users Journal</i> , 18(2), February 2000) and <i>Exceptional C++</i> 's own arguments about nonmember functions in Items 31-34.   |
|        |             | 2000.08.24     | hps   | —                    | The initial fix was slightly wrong (a typo: it said “member” where it meant “nonmember”). I also forgot about the requirement that operators <code>new</code> , <code>new[ ]</code> , <code>delete</code> , and <code>delete[ ]</code> be static members. What is shown below should now be correct.<br><br>Change:<br><ul style="list-style-type: none"> <li>– <i>Unary operators are members.</i></li> <li>– <code>= ( ) [ ]</code> and <code>-&gt;</code> must be members.</li> <li>– <i>The assignment operators (<code>+=</code> <code>-=</code> <code>/=</code> <code>*=</code> and so forth) must be members.</i></li> <li>– <i>All other binary operators are nonmembers.</i></li> </ul> To:<br><i>The standard requires that operators <code>= ( ) [ ]</code> and <code>-&gt;</code> must be members, and class-specific operators <code>new</code>, <code>new[ ]</code>, <code>delete</code>, and <code>delete[ ]</code> must be static members. For all other functions:</i> <ul style="list-style-type: none"> <li><i>if the function is <code>operator&gt;&gt;</code> or <code>operator&lt;&lt;</code> for stream I/O,</i></li> <li><i>or if it needs type conversions on its leftmost argument,</i></li> <li><i>or if it can be implemented using the class's public interface alone,</i></li> <li><i>make it a nonmember (and friend if needed in the first two cases)</i></li> <li><i>if it needs to behave virtually,</i></li> <li><i>add a virtual member function to provide the virtual behavior,</i></li> <li><i>and implement it in terms of that</i></li> </ul> <i>else</i><br><i>make it a member.</i> |
| 82, 90 | Enhancement | 2000.09.07     | Thomas Petillon<br>petillon@topic.fr          | —                    | For consistency, once on page 82 and twice on page 90, “aggregation” should be “containment.”  |
| 84, 86 | Enhancement | 2000.08.30     | Thomas Petillon<br>petillon@topic.fr          | —                    | (For more details see the comment for pages 153-154.) In most of the book I demonstrate the Pimpl idiom using <code>struct</code> for both the declaration and the definition of the Pimpl class. On pages 84 and 86 I don't, so for consistency, once each in the first line of page 84 and in the middle of page 86:<br><br>Change:<br><pre>class GenericTableAlgorithmImpl* pimpl_; // MYOB</pre> To:<br><pre>struct GenericTableAlgorithmImpl* pimpl_; // MYOB</pre>   |
| 84, 87 | Typo        | 2000.01.12     | Steve Vinoski<br>vinoski@iona.com             | 2                    | In the example code comment “override Filter() and ProcessRow() do implement a specific operation,” “do” should be “to.”   |
| 92, 95 | Enhancement | 2000.02.10     | M. Thomas Groszko<br>tom.groszko@all-mmii.com | —                    | URL moved, <a href="http://www.oma.com">www.oma.com</a> is now <a href="http://www.objectmentor.com">www.objectmentor.com</a> .  |
| 100    | Typo        | 2000.09.07     | Thomas Petillon<br>petillon@topic.fr          | —                    | The first line “using namespace std;” is redundant and shouldn't be there.<br><br>Remove the first line:<br><pre>using namespace std;</pre>  |
| 103    | Typo        | 1999.12.27     | Kjell Swedin<br>kjells@wrq.com                | 2                    | In the last line, “Lokos96” should be “Lakos 96.”  |

| Page | Severity    | First Reported |   | Corrected Printing # | Description  |
|------|-------------|----------------|---|----------------------|--|
|      |             | Date           | By  |                      |  |
| 106  | Enhancement | 2000.09.07     | Thomas Petillon<br>petillon@topic.fr              | —                    | The sense should be understood, but to be consistent with page 102, in the second paragraph of Item 28:<br><br>Change:<br>...so that existing code that uses <i>X</i> is unaffected.<br><br>To:<br>...so that existing code that uses <i>X</i> is unaffected beyond requiring a simple recompilation.  |
| 107  | Enhancement | 2000.02.10     | M. Thomas Groszko<br>tom.groszko@ait-mmii.com     | —                    | URL moved, <a href="http://www.oma.com">www.oma.com</a> is now <a href="http://www.objectmentor.com">www.objectmentor.com</a> .  |
| 107  | Typo        | 2000.01.12     | Steve Vinoski<br>vinoski@iona.com                 | 2                    | In the first Solution paragraph, I refer to a future Item that is actually earlier in the book.<br><br>Change:<br>I'll save the whole lecture for a later Item, but my bottom line is simply that...<br><br>To:<br>See Item 24 for the whole exhausting lecture; the bottom line is simply that...   |
| 108  | Correction  | 2000.04.30     | Brian Danilko<br>bdanilko@formalsolutions.com.au  | —                    | A forward declaration is still needed for class <i>B</i> , because <i>B</i> is still mentioned in some function declarations.<br><br>In the first paragraph, delete the text:<br>and in order to get rid of the <code>b.h</code> header entirely,<br><br>In the code, before the line <code>class C;</code> insert a new line:<br><code>class B;</code>  |
| 110  | Typo        | 2000.09.01     | Tetsuroh Asahata<br>asahata@jp.ibm.com            | —                    | In the second paragraph, "at at time" should be "at a time."   |
| 110  | Typo        | 2000.10.06     | hps   | —                    | In Option 1, change:<br>(rather than <code>#include</code> the class's actual declaration,<br><br>To:<br>(rather than <code>#include</code> the class's actual definition,   |
| 110  | Enhancement | 2000.12.05     | John McGuinness<br>John_McGuinness@Mastercard.com | —                    | In Option 2, change:<br><i>Option 2 (Score: 10 / 10): Put all private members into XImpl.</i><br><br>To:<br><i>Option 2 (Score: 10 / 10): Put all nonvirtual private members into XImpl.</i><br><br>Change the following paragraph:<br>There are some caveats, the first of which is the reason for my "almost" above.<br><br>To:<br>There are some caveats.<br><br>In the following bullet, change the paragraph:<br>Making a virtual function private is usually not a good idea, anyway. The point of a virtual function is to allow a derived class to redefine it, and a common redefinition technique is to call the base class's version (not possible, if it's private) for most of the functionality.<br><br>To:<br>Virtual functions should normally be private, except that they have to be protected if a derived class's version needs to call the base class's version (for example, for a virtual <code>DoWrite()</code> persistence function). |
| 113  | Typo        | 2000.03.21     | Klaus Ahrens<br>ahrens@informatik.hu-berlin.de    | —                    | In the paragraph following the output 1 and 8, "x2" should be "x."<br><br>Change:<br>...inside each <i>X2</i> object...<br><br>To:<br>...inside each <i>X</i> object...  |

| Page       | Severity    | First Reported |                                      | Corrected Printing # | Description  |
|------------|-------------|----------------|--------------------------------------|----------------------|--|
|            |             | Date           | By                                   |                      |  |
| 118        | Typo        | 2000.09.07     | Thomas Petillon<br>petillon@topic.fr | —                    | Footnote 10 should not exist. Delete it.   |
| 124        | Typo        | 2000.01.12     | Steve Vinoski<br>vinoski@iona.com    | 2                    | In the third-to-last line, “an” should be “and.”   |
| 127        | Enhancement | 2000.09.07     | hps                                  | —                    | <p>At the top of the page, I present two options. Technically there’s one more option: A using declaration for <code>std::operator&lt;&lt;()</code>.</p> <p>Change:</p> <p>...have to write either “<code>std::operator&lt;&lt;( std::cout, hello );</code>” which is exceedingly ugly, or “<code>using namespace std;</code>” which dumps all the names in <code>std</code> into the current namespace and thus eliminates much of the advantage of having namespaces in the first place.</p> <p>To:</p> <p>...have to write either “<code>std::operator&lt;&lt;( std::cout, hello );</code>” which is exceedingly ugly, or “<code>using std::operator&lt;&lt;;</code>” which is annoying and quickly becomes tedious if there are many operators, or “<code>using namespace std;</code>” which dumps all the names in <code>std</code> into the current namespace and thus eliminates much of the advantage of having namespaces in the first place.</p> |
| 127        | Typo        | 2000.08.12     | hps                                  | —                    | <p>In the footnote, “article” should be “Item.”</p> <p>Change:</p> <p>...later in this article.</p> <p>To:</p> <p>...later in this Item.</p>   |
| 132        | Typo        | 2000.01.12     | Steve Vinoski<br>vinoski@iona.com    | 2                    | <p>In the paragraph starting “Finally,” change:</p> <p>...is a member function of <code>a</code>:</p> <p>To:</p> <p>...is a member function of <code>A</code>:</p>   |
| 133        | Typo        | 2000.10.01     | Thomas Petillon<br>petillon@topic.fr | —                    | <p>At the bottom of the page, change:</p> <p><code>// Example 1: Will this compile?</code></p> <p>To:</p> <p><code>// Example 2: Will this compile?</code></p>   |
| 141        | Enhancement | 1999.12.08     | hps                                  | 2                    | <p>Needs to stress more strongly that “heap” and “free store” are commonly used terms, not words from the standard.</p> <p>At the end of the final sentence, add:</p> <p>; in particular, “heap” and “free store” are common and convenient shorthands for distinguishing between two kinds of dynamically allocated memory</p>  |
| 149        | Typo        | 1999.12.08     | hps                                  | 2                    | <p>In the code example, change:</p> <p><code>new (shared) T; // if T::T() throws, memory is leaked</code></p> <p>To:</p> <p><code>new (shared) Y; // if Y::Y() throws, memory is leaked</code></p>   |
| 151, 157-8 | Correction  | 2000.01.11     | Douglas Gilbert<br>dgilbert@724.com  | 2                    | <p>At the top of page 151, toward the end of the code example, add “<code>/*...*/</code>” after “<code>private:</code>”.</p> <p>At the bottom of page 157, add “<code>/*...*/</code>” after “<code>private:</code>”.</p> <p>At the top of page 158, add the following new paragraph:</p> <p>Possible issue: One of the <code>/*...*/</code> areas (whether <code>public</code>, <code>protected</code>, or <code>private</code>) had better include at least declarations for copy construction and copy assignment.</p>   |
| 153, 154   | Enhancement | 2000.08.23     | Thomas Petillon<br>petillon@topic.fr | —                    | <p>It’s perfectly legal and standards-conforming to forward-declare a <code>class</code> as a <code>struct</code> and vice versa. In most of the book I’ve tended to avoid doing that, though. Why? Only because some compilers are buggy and still don’t get this right — e.g., by name-mangling a <code>class</code> and a <code>struct</code></p>   |



| Page | Severity    | First Reported |                                       | Corrected<br>Printing # | Description   |
|------|-------------|----------------|---------------------------------------|-------------------------|---|
|      |             | Date           | By                                    |                         |   |
|      |             |                |                                       |                         | <p>differently, which will cause the linker to fail to match them up. Such compiler bugs really are bugs and are wrong, but they're common enough that we might as well avoid the issue by not relying on this standard feature. Sigh.</p> <p>Specifically, in most of the book I demonstrate the Pimpl idiom using <code>struct</code> for both the declaration and the definition of the Pimpl class. On pages 153 and 154 I don't, so for consistency, once each in Example 4(a) and 4(b):</p> <p>Change:</p> <pre>class C::CImpl { /* ... */ };</pre> <p>To:</p> <pre>struct C::CImpl { /* ... */ };</pre>  |
| 154  | Enhancement | 1999.12.08     | hps                                   | 2                       | <p>In the first paragraph following Example 4(b), after the first sentence add: Better still, it means <code>C::C()</code> has to do less work to detect and recover from constructor failures because <code>pimpl_</code> is always automatically cleaned up.</p>  |
| 158  | Enhancement | 1999.12.08     | hps                                   | —                       | <p><b>ADD NEW MATERIAL:</b> Before "The <code>const auto_ptr</code> idiom" add the new section "auto_ptr and Exception Safety," included later in this errata document.</p>   |
| 160  | Typo        | 2000.05.14     | hps                                   | —                       | <p>In the second guideline, the word "to" is missing.</p> <p>Change:</p> <p>It's all right use a ...</p> <p>To:</p> <p>It's all right to use a ...</p>  |
| 160  | Correction  | 2000.08.24     | Andrew Koenig<br>ark@research.att.com | —                       | <p>(See the discussion for the corresponding erratum for page 37.)</p> <p>In the second paragraph, change:</p> <p>If <code>T::operator=( )</code> is exception-safe, it doesn't need to test for self-assignment. Period. Because we should always write exception-safe code, we should never perform the self-assignment test, right?</p> <p>To:</p> <p>If <code>T::operator=( )</code> is written using the create-a-temporary-and-swap idiom (see page 47), it will be both strongly exception-safe and not <i>have</i> to test for self-assignment. Period. Because we should normally prefer to write copy assignment this way, we shouldn't need to perform the self-assignment test, right?</p> <p>In the following Guideline, change:</p> <p><i>... an exception-safe copy assignment operator is automatically safe for self-assignment.</i></p> <p>To:</p> <p><i>... a copy assignment operator that uses the create-a-temporary-and-swap idiom is automatically both strongly exception-safe and safe for self-assignment.</i></p> |
| 161  | Enhancement | 2000.06.22     | hps                                   | —                       | <p>In the footnote, at the end of the final paragraph ("Yikes..."), append:</p> <p>Similarly, the following code is also not valid C++, in that it's semantically legal (a conforming compiler must accept it) but has undefined behavior (a conforming compiler may legitimately emit code that will reformat your hard drive). If the code were valid, it would also make the test meaningful:</p> <pre>T t = t; // invalid, but it would make the test meaningful</pre>  |
| 164  | Typo        | 2000.09.15     | Thomas Petillon<br>petillon@topic.fr  | —                       | <p>In the last paragraph, change:</p> <p>Here, the slicing issue is that <code>t.f()</code> replaces...</p> <p>To:</p> <p>Here, the slicing issue is that <code>t.DestroyAndReconstruct()</code> replaces...</p>  |
| 168  | Typo        | 2000.05.24     | Sam Lindley<br>sam@redsnapper.net     | —                       | <p>In the last line, I say "initialization is resource acquisition" instead of "resource acquisition is initialization."</p>  |

| Page  | Severity    | First Reported |  | Corrected Printing # | Description   |
|-------|-------------|----------------|--|----------------------|---|
|       |             | Date           | By   |                      |   |
|       |             |                |  |                      | Change:<br>"initializat ion is resource acquisition"<br>To:<br>"resource acquisition is initialization"   |
| 169   | Correction  | 2000.08.24     | Andrew Koenig<br>ark@research.att.com<br><br>Bill Wade<br>wwade@swbell.net | —                    | (See the discussion for the corresponding erratum for page 37.)<br><br>At the bottom of the page, change:<br>Any copy assignment that <i>must</i> check for self-assignment is not exception-safe.<br><br>To:<br>Any copy assignment that is written in such a way that <i>it must</i> check for self-assignment is probably not strongly exception-safe.<br><br>In the following Guideline, change:<br>... an exception-safe copy assignment operator is automatically safe for self-assignment.<br><br>To:<br>... an copy assignment operator that uses the create-a-temporary-and-swap idiom is automatically both strongly exception-safe and safe for self-assignment. |
| 172   | Enhancement | 2000.08.12     | hps  | —                    | C++ Report no longer exists, so remove it.  |
| 174   | Correction  | 2000.12.12     | Mark Handy<br>mhandy@neonsoft.com  | —                    | After the code example "T t(u);", change:<br>This is <i>direct initialization</i> . The variable t is initialized directly from the value of u by calling T::T(u).<br><br>To:<br>Assuming u is not the name of a type, this is <i>direct initialization</i> . The variable t is initialized directly from the value of u by calling T::T(u). (If u is a type name, this is a declaration even if there is also a variable named u in scope; see above.)   |
| 176   | Typo        | 2000.01.18     | Douglas Gilbert<br>dgilbert@724.com  | 2                    | In the code example at the bottom of the page, in the comment "not the same as f(int&)," "f(int&)" should be "g(int&)."   |
| 176   | Enhancement | 2000.08.21     | Robert Dick<br>dickrp@EE.Princeton.EDU                                     | —                    | In the Guideline, change:<br>Avoid declaring <code>const</code> pass-by-value function parameters.<br><br>To:<br>Avoid <code>const</code> pass-by-value parameters in function declarations. Still make the parameter <code>const</code> in the same function's definition if it won't be modified.   |
| 179   | Typo        | 1999.11.26     | hps  | 2                    | Vestigial plural.<br><br>Change:<br>(If, in looking for the "bonus" part, you said something about these two functions being uncompileable—sorry, they're quite legal C++. You were probably thinking of putting the <code>const</code> to the left of the <code>&amp;</code> or <code>*</code> , which would have made the function body illegal.)<br><br>To:<br>(If, in looking for the "bonus" part, you said something about this function being uncompileable—sorry, it's quite legal C++. You were probably thinking of putting the <code>const</code> to the left of the <code>*</code> , which would have made the function body illegal.)                          |
| 180-2 | Format      | 1999.12.29     | hps  | 2                    | Move page 182 to page 180 (so that 180/181 become 181/182) to put the box closer to the text it accompanies.  |
| 181-3 | Format      | 1999.12.29     | hps  | —                    | Restore the originally intended vertical whitespace to the Item 44 question code to make it more readable.  |
| 185   | Correction  | 1999.12.28     | Chris Uzdavinis<br>chris@aldesk.com  | 2                    | The commentary for pa3 code line doesn't take into account possible friendship.<br><br>Change:  |

| Page                  | Severity    | First Reported |   | Corrected<br>Printing # | Description  |
|-----------------------|-------------|----------------|---|-------------------------|--|
|                       |             | Date           | By                                      |                         |  |
|                       |             |                |   |                         | <p>Error: Because b1 IS-NOT-AN A (because B is not publicly derived from A; its derivation is private), this is illegal.</p> <p>To:<br/>Probable error: Because b1 IS-NOT-AN A (because B is not publicly derived from A; its derivation is private), this is illegal unless g() is a friend of B.</p> |
| 188                   | Typo        | 2000.08.13     | Philip Brabbin<br>pabrabbin@hotmail.com | —                       | <p>In Option 2, the <code>#define</code> directive is backwards.</p> <p>Change:<br/><code>#define int bool</code></p> <p>To:<br/><code>#define bool int</code></p>   |
| 195                   | Typo        | 2000.08.12     | hps                                     | —                       | <p>In the last line, change:<br/>...but it run correctly.</p> <p>To:<br/>...but it will run correctly.</p>   |
| 196                   | Typo        | 2000.01.01     | George Reilly<br>george@reilly.org      | 2                       | <p>At the bottom of the page, the <code>Resize()</code> function contains a spurious <code>memset()</code> call that is incorrect and was not in the original question.</p> <p>Delete:<br/><code>memset( buffer_, ' ', newSize );</code></p>   |
| 197                   | Typo        | 2000.10.01     | Thomas Petillon<br>petillon@topic.fr    | —                       | <p>In the expansion of the return statement, change:<br/><code>", y = " ) ,</code></p> <p>To:<br/><code>", used = " ) ,</code></p>   |
| 201                   | Enhancement | 2000.08.12     | hps                                     | —                       | <code>C++ Report</code> no longer exists, so remove it.  |
| 203                   | Typo        | 2000.07.17     | hps                                     | —                       | Nathan's last name is Myers, not Meyers. In the Meyers97 reference, change "Meyers" to "Myers" in both places.   |
| <a href="#">Index</a> | Enhancement | 2000.07.01     | Scott Meyers<br>smeyers@arsteia.com     | —                       | <a href="#">REPLACE INDEX: A more thorough index is included later in this errata document. It replaces the index originally included in the first two printings.</a>  |

## auto\_ptr and Exception Safety

Finally, `auto_ptr` is sometimes essential to writing exception-safe code. Consider the following function:

```
// Exception-safe?
//
String f()
{
    String result;
    result = "some value";
    cout << "some output";
    return result;
}
```

This function has two visible side effects: It emits some output, and it returns a `String`. A detailed examination of exception safety is beyond the scope of this Item,<sup>1</sup> but the goal we want to achieve is the strong exception-safety guarantee, which boils down to ensuring that the function acts atomically—even if there are exceptions, either all side effects happen or none of them do.

Although the above code comes pretty close to achieving the strong exception-safety guarantee, there's still one minor quibble, as illustrated by the following client code:

```
String theName;
theName = f();
```

The `String` copy constructor is invoked because the result is returned by value, and the copy assignment operator is invoked to copy the result into `theName`. If either copy fails, then `f()` has completed all of its work and all of its side effects (good), but the result has been irretrievably lost (oops).

Can we do better, and perhaps avoid the problem by avoiding the copy? For example, we could let the function take a non-const `String` reference parameter and place the return value in that:

```
// Better?
//
void f( String& result )
{
    cout << "some output";
    result = "some value";
}
```

This may look better, but it isn't, because the assignment to `result` might still fail which leaves us with one side effect complete and the other incomplete. Bottom line, this attempt doesn't really buy us much.

One way to solve the problem is to return a pointer to a dynamically allocated `String`, but the best solution is to go a step farther and return the pointer in an `auto_ptr`:

```
// Correct (finally!)
//
auto_ptr<String> f()
{
    auto_ptr<String> result = new String;

    *result = "some value";
    cout << "some output";
}
```

---

<sup>1</sup> See Items 8 to 19.

```
    return result;
    // rely on transfer of
    // ownership; this can't throw
}
```

This does the trick, since we have effectively hidden all of the work to construct the second side effect (the return value) while ensuring that it can be safely returned to the caller using only nonthrowing operations after the first side effect has completed (the printing of the message). We know that, once the `cout` is complete, the returned value will make it successfully into the hands of the caller, and be correctly cleaned up in all cases: If the caller accepts the returned value, the act of accepting a copy of the `auto_ptr` causes the caller to take ownership; and if the caller does not accept the returned value, say by ignoring the return value, the allocated `String` will be automatically cleaned up as the temporary `auto_ptr` holding it is destroyed. The price for this extra safety? As often happens when implementing strong exception safety, the strong safety comes at the (usually minor) cost of some efficiency—here, the extra dynamic memory allocation. But, when it comes to trading off efficiency for correctness, we usually ought to prefer the latter!

Make a habit of using smart pointers like `auto_ptr` in your daily work. `auto_ptr` neatly solves common problems and will make your code safer and more robust, especially when it comes to preventing resource leaks and ensuring strong exception safety. Because it's standard, it's portable across libraries and platforms, and so it will be right there with you wherever you take your code.

## Updated Index

This updated index applies to all printings of *Exceptional C++*, and is the index included in the book from the third printing onward.

- #define,
  - see: *macros*
- #include,
  - see: *Pimpl idiom*
- #pragma, 113
  
- A**
- AAssert, 192
- Abrahams, Dave, 25, 38, 59
- abstract class,
  - see: *class, abstract*
- abstraction, 97-98
- accumulate, 136-137
  - example use, 134
- Adler, Darin, 136
- aggregation,
  - see: *containment*
- AInvariant, 193
- AINVARIANT\_GUARD, 193-194
- Alexandrescu, Andrei, xiii
- algorithms,
  - auto\_ptr and, 156-157
  - standard library,
    - see: algorithms by name (e.g., *copy*)
- alignment, 113, 117-118
- allocation,
  - see: *memory and resource management*
- allocator, 59
- ambiguity,
  - see: *function overloading; name lookup*
- Annotated C++ Reference Manual (ARM),
  - alluded to, 187
- Array, 193-194
- arrays,
  - misuse of, 148
  - polymorphism and, 147
  - prefer vector or deque instead, 147-148
- assert, 195
  - example use, 30, 192-194
- assign, 14-15
- assignment
  - avoiding by constructing in place, 53
  - copy assignment, 9-17
    - not a template, 11-13
  - copy construction interaction, 165-172
  - iterator ranges and, 14-15
  - self-assignment,
    - exception safety and, 32, 37, 47, 160, 169-171
      - need to check for, 32, 37, 159-161, 169-171
    - swap and, 47, 170-171
    - templated, 11-17
    - to self,
      - see: *assignment, self-assignment*
- Austern, Matt, 25, 59
- auto, 174
- auto\_ptr, 150-158
  - see also: *sources; sinks*
  - algorithms and, 156-157
  - const auto\_ptr idiom, 158
  - containers and, 156-157
  - example use, 66
  - members, 153-154, 157-158
  - ownership, 154-155, 157
  - passing, 154-157
  - Pimpl idiom and, 153-154
  - returning can improve exception safety, 66-67
  - usefulness of, 151-153
  - wrapping pointer data members, 153-154, 157-158
  
- B**
- back\_inserter,
  - example use, 1-2
- bad\_alloc, 145-150, 197
- bad\_cast, 186
- base class,
  - see: *class, base*
- basic\_ostream, 101-102
  - example use, 9
- basic\_string, 5
  - see also: *string(s)*
  - c\_str vs. implicit conversion to char\*, 162-163
- BasicProtocol, 80-82
- bool, 187-190
- Bridge pattern,
  - see: *design patterns, Bridge*
  
- C**
- C,
  - object-oriented programming in, 124-125
- C With Classes, 181-182
- C++ Report, 25
- caching precomputed values, 18
- calloc, 143
  - relationship with new, 143
- Cargill, Tom, 25-26, 30, 32
- case-insensitive comparison
  - see: *string(s), case-insensitive comparison*
- casts,
  - see also: *const\_cast, dynamic\_cast, reinterpret\_cast, static\_cast*
  - C vs. C++ casts, 183
  - void\* and, 184
- cat, abuse of, 38
- catch,
  - see: *exception safety*
- Chang, Juana,
  - PeerDirect Dessert Society and, v
- char\_traits, 5
  - see also: *string(s), comparison*
- Cheers, Mark, v
- ci\_char\_traits, 4-9
- ci\_string, 4-9
- cin,
  - example use, 1-2
- Clamage, Steve, xiii
- class(es),
  - see also: *name hiding*
  - abstract, 90
  - base, 77
  - design considerations and guidelines, 69-98
  - empty base,
    - see also: *empty base class optimization*
  - interface of,
    - see: *Interface Principle*
  - member vs. nonmember functions, 125
    - see also: *Interface Principle*
  - namespaces and, 136-140
  - one class, one responsibility,
    - see: *cohesion*
  - Pimpl idiom and,
    - see: *Pimpl idiom*
  - private members, hiding,
    - see: *Pimpl idiom*
  - virtual base,
    - see: *inheritance, virtual*
- code paths,
  - see: *control flow*
- code reuse,
  - see: *reuse*
- cohesion, 67, 85, 94
  - see also: *coupling*
- Colvin, Greg, 25, 59, 150
- communications protocol example, 80-82
- comp.lang.c++.moderated newsgroup, ix-xii, 25
- comparison, case-insensitive
  - see: *string(s), case-insensitive*

*comparison*  
 compilation dependencies, 87  
   see also: *Pimpl idiom*  
 compilation firewalls,  
   see: *Pimpl idiom*  
 complex, 79  
   example use, 76  
 Complex, 69-75  
 composition,  
   see: *containment*  
 conditions,  
   short-circuit evaluation and,  
     see: *control flow*  
 const, 175-181  
   see also: *mutable*  
   auto\_ptr idiom, 158  
   cast,  
     see: *const\_cast*  
   correctness, 175-181  
   iterator,  
     see: *iterator(s)*, *const\_iterator*  
   member functions, 177-181  
   return value, 72-73  
 const correctness,  
   see: *const*, *correctness*  
 const\_cast, 176-187  
   undefined behavior and, 179  
   to work around absence of mutable,  
     178  
   to work around const-incorrect third-  
     party interfaces, 181-182  
 const\_iterator,  
   see: *iterator(s)*, *const\_iterator*  
 construct, 41-42  
   used, 46, 48, 51  
 constructor,  
   see also: *initialization*  
   conversion by, 19, 70  
   see also: *conversions*, *implicit*  
   copy constructor, 9-17, 53-54  
     copy assignment interaction, 165-  
       172  
     elision by compiler, 190-191  
     not a template, 11-13  
   default, 15, 27-28, 53-54  
   exceptions and, 28, 58, 62  
   explicit, 19, 70, 162-163  
   failure of, 28  
   initialization list, 196  
   iterator ranges and, 14-15  
   templated, 11-17  
 containers,  
   see also: *list*; *vector*; etc.  
   auto\_ptr and, 16-157  
   destructible, 38, 59  
   homogeneous, 13  
 containment,  
   generic, 94-95  
   inheritance vs., 82, 89-96, 107-108  
 control flow, 60-68  
   exceptions and,

    see: *exceptions*, *control flow and*  
     short-circuit evaluation and, 61-62  
 conversions,  
   implicit, 19, 70, 162-163, 189  
   exceptions and, 62  
   explicit, 19, 70  
 Coplien, Jim, 109  
 copy,  
   example use, 1-2, 11-12, 30, 193  
   exception safety and, 15-17, 197  
 copy assignment,  
   see: *assignment*, *copy assignment*  
 copy construction,  
   see: *constructor*, *copy constructor*  
 coupling, 88, 92  
   see also: *cohesion*; *Pimpl idiom*  
**D**  
 declaration,  
   definition vs., 85  
   forward, 85, 100-102  
 decoupling,  
   see: *coupling*; *Pimpl idiom*  
 default constructor  
   see: *constructor*, *default*  
 deallocate, 59  
 default parameters, 70, 78  
   example use, 75  
 #define,  
   see: *macros*  
 definition vs. declaration,  
   see: *declaration*, *definition vs.*  
 delegation,  
   see: *containment*  
 delete, 27, 29, 144-150  
   array delete[], 29, 57-58  
   example use, 31  
   placement, 148-149  
   relationship with free, 142-143  
   should never throw, 29, 57-58  
   virtual destructor and,  
     see: *virtual destructor*  
 dependencies,  
   see: *coupling*; *Pimpl idiom*; *Interface*  
   *Principle*  
 deque,  
   arrays vs., 147-148  
   exception guarantees and, 59-60  
 design patterns, 84-85  
   Bridge pattern, 84  
   Pimpl idiom vs., 84  
   Singleton, alluded to, 115  
   Strategy, 87  
   Template Method pattern, 84  
   public virtual functions vs., 84  
 destroy, 41-42, 55-56  
   used, 49, 52  
 destroy-and-reconstruct idiom, 163-172  
   exception safety and, 168  
 destructor, 29  
   operator delete and, 147

  should never throw, 28-29, 55-58  
   virtual,  
     see: *virtual destructor*  
 Dewhurst, Steve, xiii  
 dynamic type,  
   see: *type*, *dynamic*  
 dynamic\_cast, 181-187  
   inheritance and, 185-186  
 dynamically allocated memory,  
   see: *memory and resource*  
   *management*  
**E**  
 Effective C++,  
   see: *not this book*  
   see also: *this book*  
 Einstein, Albert, 107  
 Employee, 17-23  
 empty base class optimization, 91  
 encapsulation, 97-98  
 end(), dereferencing,  
   see: *iterators*, *dereferencing* *end()*  
 enum, 188  
 EvaluateSalaryAndReturnName, 60-  
   68  
 evaluation order of function arguments,  
   197  
 exception handling, 97-98  
 exception neutral, 26,  
 exception safety, 25-68  
   affects class design, 17, 26, 35  
   assignment and, 15-17, 31  
   casting and, 186  
   destroy-and-reconstruct idiom and,  
     164, 168  
   dynamically allocated resources can  
     improve, 66-67  
   encapsulation and, 40  
   guarantees, 26, 38  
     basic guarantee, 38, 59  
     nothrow guarantee, 16, 38, 59  
     destructors and, 29, 55-58  
     swap and, 44, 47, 59  
   strong guarantee, 16, 38, 59  
   iterator invalidation and, 38  
   multiple side effects and, 64-68  
   not always necessary, 67  
   performance overhead and, 67  
   history of, 25  
   object lifetime and, 163-165  
   overhead and, 60  
   placement new and delete and, 148-  
     150  
   return by value and, 34-37, 66-68  
   side effects and, 34-37, 64-68  
   standard library and, 26, 59-60  
   swap and, 16, 43-44, 59, 170-171  
   throw() specification, 53  
   try/catch and, 39, 50  
 exception specifications, 54  
 Exceptional C++,

- see: *this book*
  - see also: *not this book*
  - exceptions,
    - control flow and, 61-63
    - multiple, 56
  - execution flow,
    - see: *control flow*
  - explicit, 19
  - extractor, using, 2
- F**
- false,
    - see: *bool*
  - Fast Pimpl idiom, 111-118
    - see also: *Pimpl idiom*
  - FastArenaObject, 115-118
  - find, 23
  - find\_if, 23
  - FindAddr, 17-23
  - firewall, compilation,
    - see: *Pimpl idiom*
  - FixedAllocator, 115-118
  - fixed\_vector, 9-17
  - flow of execution,
    - see: *control flow*
  - Ford Escort, 151
  - forward declaration,
    - see: *declaration, forward*
  - forwarding function, 190-192
  - free,
    - relationship with `delete`, 142-143
  - free store, 142
  - French,
    - gratuitous use of, 90, 102
  - friend, friendship, 88, 123
  - Fulcher, Margot,
    - Superwoman and, v
  - function arguments,
    - evaluation order of, 197-199
  - function overloading, 120
    - see also: *name lookup*
  - functors,
    - alluded to, 23
- G**
- garbage collection,
    - alluded to, 98
  - Generic Liskov Substitution Principle, 8
  - generic containment/delegation, 94-95
  - generic programming, 1-17, 88, 97-98
    - see also: *standard library, templates*
  - GenericTableAlgorithm, 83-88
  - GenericTableAlgorithmImpl, 86
  - Gibbons, Bill, 150
  - global data, 142
  - GLSP,
    - see: *Generic Liskov Substitution Principle*
  - GotW,
    - see: *Guru of the Week*
  - GTAClient, 86
  - Guru of the Week (GotW), ix-xii, 25, 150, 158, 201-202
- H**
- handle/body idiom,
    - see: *Pimpl idiom*
  - HAS-A, 107
    - see: *containment*
  - header files,
    - see: *Pimpl idiom*
  - heap, 142
  - Henney, Kevlin, 10
  - hiding names,
    - see: *name hiding*
  - Horstmann, Cay, xiii
  - Hyslop, Jim, xiii, 161
- I**
- implicit conversions
    - see: *conversions, implicit*
  - #include,
    - see: *Pimpl idiom*
  - increment operator
    - see: *operators, ++*
  - infinite loop,
    - see: *loop, infinite*
  - inheritance, 97-98
    - deep hierarchies of, 81
    - dynamic\_cast and, 185-186
    - empty base class optimization, 91
    - from char\_traits, 7-8
    - IS-A,
      - see: *Liskov Substitution Principle*
    - IS-ALMOST-A, 95-96
    - Liskov Substitution Principle (LSP),
      - see: *Liskov Substitution Principle*
    - not for reuse, 81-82
    - overuse of, 88-96, 107-108
    - polymorphism and, 7-8
    - private, 44, 52-53
    - protected, 92
    - public, 80-82, 95-96
      - see also: *Liskov Substitution Principle*
    - static members and, 8
    - virtual, 91
    - WORKS-LIKE-A,
      - see: *Liskov Substitution Principle*
  - initialization,
    - copy initialization, 173-174
    - default initialization, 173-174
    - direct initialization, 173-174
    - of base classes, 196
    - of global data, 192-199
    - list,
      - see: *constructor, initialization list*
    - resource acquisition and,
      - see: *resource acquisition is initialization*
    - static and, 192-199
  - inline, 20, 191-192
  - Interface Principle, 122-133
    - dependencies and, 131-133
  - iosfwd, 101
  - iostream, 100-102
  - IS-A,
    - see: *Liskov Substitution Principle*
  - IS-IMPLEMENTABLE-IN-TERMS-OF, 94-95
  - IS-IMPLEMENTED-IN-TERMS-OF, 81-82, 89-96
    - private inheritance vs. containment, 44, 52-53, 89-96
  - istream\_iterator, 1-2
  - iterator(s), 1-3
    - algorithms and, 2
    - assignment and, 14-15
    - common mistakes, 1-3
    - const\_iterator, 178
      - example use, 22
    - construction and, 14-15
    - dereferencing `end()`, 2
    - exceptions and, 59
    - lifetime,
      - see: *iterator(s), validity*
    - modifying,
      - why `--end()` may be illegal, 2-3
    - ranges, 2-3
    - validity, 2-3
      - exception safety and, 38
- J**
- Jagger, Jon, 10
  - Jones, Morgan,
    - rituals of EMACS worship and, v
- K**
- Karabegovic, Justin, v
  - Kehoe, Brendan, xiii
  - Koenig, Andrew, 120, 162
  - Koenig name lookup, 119-121, 125-130
- L**
- Lafferty, Debbie, xiii
  - Lajoie, Josée, xi
  - Lang, Marina, xiii
  - layering,
    - see: *containment*
  - less, 161
  - lifetime,
    - of iterators,
      - see: *iterator(s), validity*
    - of objects,
      - see: *object(s), lifetime*
    - of references, 21
  - Lippman, Stan, xi
  - Liskov, Barbara, 8
  - Liskov Substitution Principle, 8, 81-82, 95-96, 107-108
    - IS-ALMOST-A vs., 95-96
    - protected inheritance and, 92
  - list, 36
    - example use, 17-18, 22, 99, 103



Loi, Duk, v  
 Long, Ian, v  
 loop, infinite,  
   see: *infinite loop*  
 LSP,  
   see: *Liskov Substitution Principle*  
 Lukov, Violetta, v

**M**

Machiavelli, Niccolo,  
 alluded to, 159-161  
 macros,  
   evils of, 74, 188  
 main,  
   does not return `void`, 76-77  
   return and, 77  
   standard signatures, 77  
 malloc, 111, 114, 117  
   relationship with `new`, 142-143  
 managing dependencies,  
   see: *coupling, Pimpl idiom*  
 Mancl, Dennis, xiii  
 map,  
   pointers and, 161  
 max\_align, 118  
 member functions,  
   const, 177-181  
   nonmember vs., 125  
   see also: *Interface Principle*  
   templated,  
   see: *templates, member functions*  
 memory and resource management, 27,  
 144-150  
   avoiding leaks, 27  
   dynamically allocated resources can  
   improve exception safety, 66-67  
   encapsulating for better exception  
   safety, 40-52  
   fixed-size allocators, 114  
 Meyers, Scott, ix, xi, xiii, 25, 29, 147,  
 171  
 modules,  
   alluded to, 97-98  
 Moo, Barbara, 162  
 multiparadigm language, 97-98  
 Murphy, Edward A., Jr.  
   alluded to, 159-160  
 mutable, 178-182, 184  
   workaround using `const_cast`, 178  
 Myers Example, 128-130  
 Myers, Nathan, 8, 91, 128  
 MyList, 89-96  
 MySet, MySet1, MySet2, 89-96  
 MySet3, 94-95

**N**

name hiding, 77, 133-140  
   base classes and, 134-135  
   explicit scope resolution and, 135  
 name lookup, 119-140  
 namespaces, 119-140

  see also: *name hiding; using*  
   class design and, 136-140  
   indirect interactions between, 121, 130  
 new, 27-28, 111, 114, 117, 144-150  
   array `new[]`, 27, 57-58  
   default constructor and, 41  
   example use, 27-28, 30  
   placement, 42  
   destroy-and-reconstruct idiom, 163-172  
   exception from, 148-150  
   relationship with `malloc`, 142-143  
 new-style casts,  
   see: *casts, C vs. C++ casts*  
 NewCopy, 30-33  
 Nguyen, Kim,  
   tunnels and, v  
 not this book,  
   see: *Effective C++*

**O**

object(s),  
   destructible, 38  
   identity, 159-161  
   lifetime, 163-172  
   exception safety and, 163-164, 168  
   slicing, 164, 167-170  
   temporary  
     see: *temporary objects*  
 object-oriented programming, 97-98  
 C and, 124-125  
   not just about inheritance,  
     see: *inheritance, overuse of*  
   without classes, 124-125  
 Occam, William of, 87  
 Occam's Razor, 87  
 one class, one responsibility,  
   see: *cohesion*  
 opaque pointer,  
   see: *Pimpl idiom*  
 operator `delete`, 27-28, 59, 142-150  
   example use, 41  
   virtual destructor and, 147  
 operator `new`, 27-28, 142-150  
   example use, 41  
 operators,  
   +, 9, 71-73  
     example use, 133-134, 175  
   ++, 18-20, 73-74  
   +&, 71  
   =, 72, 165-172  
   &, 160, 163, 165  
   !=, 160  
   (), 72, 87  
   [], 72  
   ->, 72  
   <<, 9, 73, 130-133  
     example use, 100  
     exceptions and, 62, 64  
     virtual `Print()` and, 130-133  
   >>, 9  
     example use, 1-2

assignment,  
   see: *assignment*  
 chaining, 72-73  
 conversion,  
   see: *conversions*  
 delete,  
   see: *delete; operator delete*  
 member vs. nonmember, 72-73  
 new,  
   see: *new; operator new*  
 implementing related, 71-72  
 preincrement,  
   see: *operators, ++*  
 postincrement,  
   see: *operators, ++*  
 ostream, 100-102  
   example use, 100, 130  
*Overload* magazine, 10  
 overload resolution, 187-190  
   see also: *function overloading; name  
 lookup*  
 overriding vs. overloading, 77-79  
   see also: *virtual functions*

**P**

Palmer, Larry,  
   Against All Odds and, v  
 pass by reference, 190-191  
 pass by value,  
   see also: *temporary objects*  
   const and, 176  
 patterns,  
   see: *design patterns*  
 PeerDirect, v, xi-xiii, 103, 110, 201  
 Pimpl idiom, 84-85, 99-118  
   see also: *Fast Pimpl idiom*  
   auto\_ptr and, 153-154  
   back pointers and, 111  
   Bridge pattern vs., 84  
   overhead, 111-118  
   performance, 111-118  
   virtual functions and, 110  
 pivot,  
   during sorting, 157  
 placement delete,  
   see: *delete, placement*  
 placement new,  
   see: *new, placement*  
 pointer,  
   opaque,  
     see: *Pimpl idiom*  
 Polygon, 175-181  
 polymorphism, 97-98  
   see also: *Liskov Substitution Principle*  
   arrays and, 147  
   exception specifications and, 54  
   virtual destructor and, 8, 77  
   virtual functions and, 8, 82  
 Pop Goes the Weasel,  
   reference to, 34  
 postincrement operator

see: *operators*, ++  
 #pragma, 113  
 precomputing values, 18, 23  
 pregnant,  
   a little bit, 96  
 preincrement operator  
   see: *operators*, ++  
 private inheritance,  
   see: *inheritance*, *private*  
 programming, generic  
   see: *generic programming*  
 protected,  
   members, 53, 82  
 public inheritance,  
   see: *inheritance*, *public*

**Q**

queue, 94

**R**

realloc, 143  
   relationship with *new*, 143  
 recomputing values,  
   see: *precomputing values*  
 Rectangle example, 95  
 references,  
   see also: *temporary objects*, *pass-by-value and validity*, 21  
 reinterpret\_cast, 181-187  
   example use, 112, 117-118  
 reserved names, 74  
 resource acquisition is initialization, 46, 168  
 resource management,  
   see: *memory and resource management*  
 return,  
   by reference, 20-21  
   by value, 20-21, 34-37  
     exception safety and, 66-68  
   main and, 77  
 reuse,  
   analyzing reusability, 53-54  
   benefits of, 22-23  
   inheritance not for, 81-82, 96  
   standard library and, 22-23, 70  
 Rumsby, Steve, 150

**S**

scope resolution, 135  
 self-assignment,  
   see: *assignment*, *self-assignment*  
 Shakespeare, William,  
   gratuitous quotes from, 163  
 SharedMemory, 145-150  
 Shelley, Doug,  
   boom in global coffee industry and, v  
 short-circuit evaluation,  
   see: *control flow*

sink functions, 154-157  
 slicing,  
   see: *object(s)*, *slicing*  
 smart pointers,  
   see: *auto\_ptr*  
 sort, 157  
   alluded to, 23  
 source functions, 154-157  
 Square example, 95  
 Stack, 26-54  
   copy constructor, 30-31, 46, 51  
   copy assignment operator, 30-32, 47-48, 51  
   Count, 33, 48, 51  
   default constructor, 27-28, 45-46, 51  
   destructor, 29  
     eliminated, 46  
   NewCopy and, 30-31  
   Pop, 34-37, 49, 52  
     division of responsibilities with Top, 36-37  
   Push, 33-34, 48-49, 51  
   requirements on contained type, 39, 53-54  
   Top, 36-37, 49, 51  
 stack, 142  
   stack, 36, 94  
 StackImpl, 40-44  
   Swap, 40, 43-44  
   used, 45-54  
 standard library,  
   exception safety and,  
     see: *exception safety*, *standard library and reusing code from*, 22-23  
 static,  
   operators *new* and *delete* should be, 146  
   return by reference and, 21  
 static data, 142  
 static type,  
   see: *type*, *static*  
 static\_cast, 181-187  
   example use, 41, 192, 195  
 Strategy pattern,  
   see: *design patterns*, *Strategy*  
 streams,  
   see also: *operators*, >>; *ostream*  
   exception safety and, 64-65, 68  
 strcmp,  
   see: *string*, *comparison*  
 string,  
   see: *basic\_string*; *string(s)*  
 string(s),  
   case-insensitive comparison, 4-9  
   comparison, 4-9  
     done in object or function, 6-7  
   c\_str vs. implicit conversion to char\*, 162-163  
 Stroustrup, Bjarne, xi, xiii, 162, 181-182  
 Sumner, Jeff, 103  
   dazzling code magery and, v  
 swap, 42, 59

see also: *exception-safety*, *swap* and *elegant copy assignment* and, 47-48

**T**

template(s), 97-98  
   see also: *generic programming*  
   member functions, 9-17  
     see also: *assignment*, *templated*; *constructor*, *templated*  
   requirements on template parameter types, 39  
   templated assignment operator, 11-13  
   templated constructor, 11-13  
 Template Method pattern,  
   see: *design patterns*, *Template Method*  
 temporary objects, 17-23, 71  
   elision by compiler, 190-191  
   exceptions and, 62-63  
   modifying, 2-3  
   of builtin type, 2-3  
   pass by value and, 18, 71  
   recomputation and, 18  
   return-by-value and, 20-21  
 terminate, 28  
 this != other test, 159-161  
   see also: *assignment*, *self-assignment*  
 this book,  
   see: *Exceptional C++*  
 throw,  
   see: *exception safety*  
 toupper,  
   example use, 5-6  
 traits, 4-9  
 true,  
   see: *bool*  
 try,  
   see: *exception safety*  
 type,  
   dynamic, 79  
   static, 79  
 typedef, 187-188  
 typeid,  
   example use, 192-194

**U**

underscores,  
   see: *reserved names*  
 USES-A,  
   see: *containment*; *HAS-A*  
 using declarations and directives,  
   example use, 75, 89  
   forwarding function vs., 92  
   private and, 78  
   to avoid name hiding, 135

**V**

vector,  
   arrays vs., 147-148  
   example use, 1-2, 175  
   exception guarantees and, 59-60

iterator,  
  can be `T*`, 2-3  
  invalidation by `insert()`, 3  
  is random-access, 3  
virtual base class,  
  see: *inheritance*, *virtual*  
virtual destructor, 7-8, 77, 82  
  example use, 80, 83  
  operator `delete` and, 147  
  slicing and, 167  
virtual functions, 7-8, 53, 75-79, 90

avoid public, 84  
  default parameters and, 78  
  exception specifications and, 54  
  Pimpl idiom and, 110  
virtual inheritance,  
  see: *inheritance*, *virtual*  
void,  
  main and,  
    see: *main*, *does not return void*  
void\*,  
  casts and, 184

**W**  
`wchar_t`,  
  alluded to, 187  
West, Declan, v  
Wilson, Eric,  
  “in the beginning” and, v  
Wizard of Oz, The,  
  reference to, 194  
WORKS-LIKE-A,  
  see: *Liskov Substitution Principle*

— end of document —