Errata List for *Exceptional C++*


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 with a subject line containing the words “XC++ Errata.”

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

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.
<table>
<thead>
<tr>
<th>Page</th>
<th>Severity</th>
<th>First Reported</th>
<th>Corrected Printing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xii</td>
<td>Enhancement</td>
<td>2000.04.14 Michel Michaud</td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

Immediately before the subhead “How We Got Here: GotW and PeerDirect,” add the following new paragraph:

This book includes many guidelines, in which the following words usually carry a specific meaning:

- **always** = This is absolutely necessary. Never fail to do this.
- **prefer** = This is usually the right way. Do it another way only when a situation specifically warrants it.
- **consider** = This may or may not apply, but it’s something to think about.
- **avoid** = 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.
- **never** = This is extremely bad. Don’t even think about it. Career limiting move.

| 9    | Enhancement | 2000.10.31 David X. Calloway |  | Change: There are two ways to resolve this: Define insertion (operator<<()) and extraction (operator>>() for ci_strings yourself, or tack on ".c_str()" to use operator<<(const char*): |

To:

There are two ways to resolve this: Define operator<<() and operator>>() for ci_strings yourself, or tack on ".c_str()" to use operator<<(const char*) if your application’s strings don’t have embedded nulls:

| 15   | Enhancement | 2000.08.13 Howard Hinnant |  | In Item 5, the discussion of fixed_vector’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 fixed_vector 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 (GotW #16) was originally written after Items 8 to 17 (GotW #8), and now the context needs to be pointed at better. Here’s a quick ‘fix.’ |

In the final paragraph, change:

Alas, it does. Did you notice that the templated assignment operator is not strongly exception-safe? Recall that it was defined as:

To:

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 strongly exception-safe. Recall that the templated assignment operator was defined as:

| 15   | Correction  | 2000.08.12 Burkhard Kloss |  | The example code using std::copy() 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. |

Change:

```cpp
copy( v.begin(), v.end(), w.begin() );
```

To:

```cpp
copy( v.begin(), v.begin()+4, w.begin() );
```
## Errata for Exceptional C++

Updated 2000.12.12

<table>
<thead>
<tr>
<th>Page</th>
<th>Severity</th>
<th>Date</th>
<th>By</th>
<th>Corrected Printing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-7</td>
<td>Correction</td>
<td>1999.12.28</td>
<td>Tim Butler <a href="mailto:tim@indra.com">tim@indra.com</a></td>
<td>2</td>
<td>The strongly exception-safe version now requires an explicitly written copy constructor and copy assignment operator, implemented like the templated versions. Add these functions. Also, the non-default constructor is missing a memory allocation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000.02.21</td>
<td>Klaus Ahrens <a href="mailto:aahrens@informatik.hu-berlin.de">aahrens@informatik.hu-berlin.de</a></td>
<td>2</td>
<td>Also, the original fix in printing #2 had another bug — a memory leak if an exception occurs during the copy() — for which the simplest reflex here is to wrap the copy() in a try/catch.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000.08.13</td>
<td>Howard Hinnant <a href="mailto:hinnant@metrowerks.com">hinnant@metrowerks.com</a></td>
<td>—</td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>template&lt;typename O, size_t osize&gt; fixed_vector( const fixed_vector&lt;O,osize&gt;&amp; other ) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>copy( other.begin() );</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>other.begin()+min(size,osize),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>begin() );</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>template&lt;typename O, size_t osize&gt; fixed_vector( const fixed_vector&lt;O,osize&gt;&amp; other ) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v_( new T[size] ) ( try {copy(other.begin()..., other.begin()+min(size,osize), begin());}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>catch(...) { delete[] v_; throw; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fixed_vector( const fixed_vector&lt;T,size&gt;&amp; other ) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v_( new T[size] ) ( try {copy(other.begin()..., other.begin()+min(size,osize), begin());}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>catch(...) { delete[] v_; throw; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>And change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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 ) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fixed_vector&lt;T,size&gt; temp( other ); // does all the work</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Swap( temp ); // this can't throw</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>return *this;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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 ) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fixed_vector&lt;T,size&gt; temp( other ); // does all the work</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Swap( temp ); return *this; // this can't throw</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>And change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>template&lt;typename O, size_t osize&gt; fixed_vector&lt;T,size&gt;&amp; operator=( const fixed_vector&lt;T,size&gt;&amp; other ) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fixed_vector&lt;T,size&gt; temp( other ); // does all the work</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Swap( temp ); return *this; // this can't throw</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>21</td>
<td>Enhancement</td>
<td>2000.10.01</td>
<td>Thomas Petillon <a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
<td>—</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>const string&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FindAddr( /* ... */ )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>const string&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FindAddr( /* pass emps and name by reference */ )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>And change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if( /* found */ )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if( i-&gt;name == name )</td>
</tr>
<tr>
<td>31</td>
<td>Correction</td>
<td>2000.08.13</td>
<td>Howard Hinnant <a href="mailto:hinnant@metrowerks.com">hinnant@metrowerks.com</a></td>
<td>—</td>
<td>At the end of the paragraph numbered “2.” change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…must be unchanged.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…must be destructible.</td>
</tr>
<tr>
<td>32,33</td>
<td>Typo</td>
<td>2000.07.23</td>
<td>hps</td>
<td>—</td>
<td>Somehow the first presented version of stack has a member function called Size() instead of Count(). For consistency with the later versions, not to mention Cargill’s original article, it should be Count().</td>
</tr>
<tr>
<td>Page</td>
<td>Severity</td>
<td>First Reported</td>
<td>Corrected Printing</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>----------------</td>
<td>--------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Date</td>
<td>By</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>Enhancement</td>
<td>2000.01.25</td>
<td>Marc Briand</td>
<td>In one place on page 32 and four places on page 33, change:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:mbriand@mil.com">mbriand@mil.com</a></td>
<td>Size</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>To:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Count</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Correction</td>
<td>2000.08.24</td>
<td>Andrew Koenig</td>
<td>In the Common Mistake box, the wording should make it clearer that I’m criticizing code that cannot be made exception-safe because of the underlying design, not just code that happens to be incidentally exception-unsafe and only needs a local fix.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:ark@research.att.com">ark@research.att.com</a></td>
<td>Also, as Andy Koenig pointed out to me, it is possible to write a copy assignment operator that is written in such a way that it must 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 built-in/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.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bill Wade</td>
<td>In the Guideline, change:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:wrwade@swbell.net">wrwade@swbell.net</a></td>
<td>“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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>“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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In the next paragraph, change:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>… cannot be exception-safe.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>… is probably not strongly exception-safe.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:abrahams@mediaone.net">abrahams@mediaone.net</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Correction</td>
<td>2000.08.13</td>
<td>Howard Hinnant</td>
<td>The box implies that the helper functions construct() and destroy() are standard, when they aren’t.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:hinnant@metroworks.com">hinnant@metroworks.com</a></td>
<td>In the first paragraph, change:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>… use three helper functions that are directly drawn (or derived in spirit) from the standard library:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>… use three helper functions, one of which (swap()) also appears in the standard library:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Delete the final paragraph:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>42, 55, 56</td>
<td>Enhancement</td>
<td>2000.03.25</td>
<td>hps</td>
<td>This didn’t make a difference in any example in the book, but it’s a little odd: The two-parameter destroy(FwdIter,FwdIter) version is templated to take any generic iterator, and yet it calls the one-parameter destroy(T*) by passing it one of the iterators… which requires that FwdIter must be a plain old pointer! This needlessly loses some of the</td>
<td></td>
</tr>
</tbody>
</table>
Errata for *Exceptional C++*

Updated 2000.12.12

<table>
<thead>
<tr>
<th>Page</th>
<th>Severity</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>Typo</td>
<td>1999.12.23</td>
<td>Steve Vinoski</td>
<td>In paragraph 2, change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:vinoski@iona.com">vinoski@iona.com</a></td>
<td>StampImpl&lt;T&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>StackImpl&lt;T&gt;</td>
</tr>
<tr>
<td>46,</td>
<td>Typo</td>
<td>2000.05.24</td>
<td>Sam Lindley</td>
<td>In the Guideline, I say “initialization is resource acquisition” instead of</td>
</tr>
<tr>
<td>58</td>
<td></td>
<td></td>
<td><a href="mailto:sam@redsnapper.net">sam@redsnapper.net</a></td>
<td>“resource acquisition is initialization.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>“initialization is resource acquisition”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>“resource acquisition is initialization”</td>
</tr>
<tr>
<td>48</td>
<td>Enhancement</td>
<td>2000.06.16</td>
<td>Stan Brown</td>
<td>To make it more obvious that other is passed by value and hence already a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="mailto:brahms@mindspring.com">brahms@mindspring.com</a></td>
<td>temporary object, change other to temp and add more explanation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If you’re one of those folks who like terse code, you can write the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>operator=() canonical form more compactly as:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stack&amp; operator=(Stack other)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Swap( other );</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>return *this;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If you’re one of those folks who like terse code, you can write the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>operator=() canonical form more compactly using pass-by-value to create the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>temporary:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stack&amp; operator=(Stack temp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Swap( temp );</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>return *this;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>49</td>
<td>Enhancement</td>
<td>2000.02.21</td>
<td>hps</td>
<td>In the paragraph following the bullets, it talks about ‘if we allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>iterators,’ but note that we do allow taking a reference into the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>container (via Top()) which is much the same thing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If we were supporting iterators into this container, for instance, they</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>would never be invalidated (by a possible internal grow operation) if the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>insertion is not completely successful.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Any references returned from Top(), or iterators if we later chose to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>provide them, would never be invalidated (by a possible internal grow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>operation) if the insertion is not completely successful.</td>
</tr>
</tbody>
</table>

Generality of templating on `FwdIter`. A simple change lets `FwdIter` be pretty much any iterator type, not just a pointer: In `destroy(FwdIter,FwdIter)`, change the call `destroy( first )` to `destroy( &*first )`. This will work in all cases, unless `T` provides an `operator&()` that does not return a pointer which should occur rarely if ever.

On pages 42, 55, and 56, in three places change the two-parameter version of `destroy()` as above.

Change:

```
    destroy( first );
```

To:

```
    destroy( &*first );
```

See also GotW #68 at www.peerdirect.com/resources.

---

**49**  Enhancement  2000.02.21  hps  —  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 `Top()`) which is much the same thing.

Change:

```
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.
```

To:

```
Any references returned from `Top()`, 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.
```
<table>
<thead>
<tr>
<th>Page</th>
<th>Severity</th>
<th>First Reported</th>
<th>Corrected Printing #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>57, 58</td>
<td>Format</td>
<td>2000.07.24</td>
<td>hps</td>
<td>In the two Guidelines, the word “overloaded” should not be in code font.</td>
</tr>
<tr>
<td>64</td>
<td>Typo</td>
<td>1999.12.13</td>
<td>Jon Kalb <a href="mailto:kalb@libertysoft.com">kalb@libertysoft.com</a></td>
<td>2</td>
</tr>
<tr>
<td>70</td>
<td>Typo</td>
<td>2000.01.06</td>
<td>Douglas Gilbert <a href="mailto:dgilbert@724.com">dgilbert@724.com</a></td>
<td>2</td>
</tr>
<tr>
<td>72</td>
<td>Correction</td>
<td>2000.01.16</td>
<td>Eric Nagler <a href="mailto:epn@eric-nagler.com">epn@eric-nagler.com</a></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000.01.24</td>
<td>hps</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000.08.24</td>
<td>hps</td>
<td>—</td>
</tr>
<tr>
<td>82, 90</td>
<td>Enhancement</td>
<td>2000.09.07</td>
<td>Thomas Petillon <a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
<td>—</td>
</tr>
<tr>
<td>84, 86</td>
<td>Enhancement</td>
<td>2000.08.30</td>
<td>Thomas Petillon <a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000.01.12</td>
<td>Steve Vinoski <a href="mailto:vinoski@iona.com">vinoski@iona.com</a></td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>Typo</td>
<td>2000.09.07</td>
<td>Thomas Petillon <a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
<td>—</td>
</tr>
<tr>
<td>103</td>
<td>Typo</td>
<td>1999.12.27</td>
<td>Kjell Swedin <a href="mailto:kjells@wrq.com">kjells@wrq.com</a></td>
<td>2</td>
</tr>
<tr>
<td>Page</td>
<td>Severity</td>
<td>First Reported</td>
<td>Corrected Printing #</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>----------------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>106</td>
<td>Enhancement</td>
<td>2000.09.07 Thomas Petillon</td>
<td>—</td>
<td>The sense should be understood, but to be consistent with page 102, in the second paragraph of Item 28:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…so that existing code that uses X is unaffected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…so that existing code that uses X is unaffected beyond requiring a simple recompilation.</td>
</tr>
<tr>
<td>107</td>
<td>Typo</td>
<td>2000.01.12 Steve Vinoski</td>
<td>2</td>
<td>In the first Solution paragraph, I refer to a future Item that is actually earlier in the book.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I’ll save the whole lecture for a later Item, but my bottom line is simply that…</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See Item 24 for the whole exhausting lecture; the bottom line is simply that…</td>
</tr>
<tr>
<td>108</td>
<td>Correction</td>
<td>2000.04.30 Brian Danilko</td>
<td>—</td>
<td>A forward declaration is still needed for class B, because B is still mentioned in some function declarations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In the first paragraph, delete the text:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>and in order to get rid of the b.h header entirely,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In the code, before the line “class C;” insert a new line:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>class B;</td>
</tr>
<tr>
<td>110</td>
<td>Typo</td>
<td>2000.09.01 Tetsuroh Asahata</td>
<td>—</td>
<td>In the second paragraph, “at at time” should be “at a time.”</td>
</tr>
<tr>
<td>110</td>
<td>Typo</td>
<td>2000.10.06 hps</td>
<td>—</td>
<td>In Option 1, change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(rather than #include the class’s actual declaration,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(rather than #include the class’s actual definition,</td>
</tr>
<tr>
<td>110</td>
<td>Enhancement</td>
<td>2000.12.05 John McGuinness</td>
<td>—</td>
<td>In Option 2, change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Option 2 (Score: 10 / 10): Put all private members into XImpl.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Option 2 (Score: 10 / 10): Put all nonvirtual private members into XImpl.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Change the following paragraph:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>There are some caveats, the first of which is the reason for my “almost” above.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>There are some caveats.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>In the following bullet, change the paragraph:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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 DoWrite() persistence function).</td>
</tr>
<tr>
<td>113</td>
<td>Typo</td>
<td>2000.03.21 Klaus Ahrens</td>
<td>—</td>
<td>In the paragraph following the output 1 and 8, “X2” should be “X.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…inside each X2 object…</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…inside each X object…</td>
</tr>
<tr>
<td>Page</td>
<td>Severity</td>
<td>First Reported</td>
<td>Date</td>
<td>By</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>----------------</td>
<td>----------</td>
<td>----------------</td>
</tr>
<tr>
<td>118</td>
<td>Typo</td>
<td>2000.09.07</td>
<td>Thomas Petillon</td>
<td><a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
</tr>
<tr>
<td>124</td>
<td>Typo</td>
<td>2000.01.12</td>
<td>Steve Vinoski</td>
<td><a href="mailto:vinoski@iona.com">vinoski@iona.com</a></td>
</tr>
<tr>
<td>127</td>
<td>Enhancement</td>
<td>2000.09.07</td>
<td>hps</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>Typo</td>
<td>2000.08.12</td>
<td>hps</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>Typo</td>
<td>2000.01.12</td>
<td>Steve Vinoski</td>
<td><a href="mailto:vinoski@iona.com">vinoski@iona.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>Typo</td>
<td>2000.10.01</td>
<td>Thomas Petillon</td>
<td><a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>Enhancement</td>
<td>1999.12.08</td>
<td>hps</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>Typo</td>
<td>1999.12.08</td>
<td>hps</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>151</td>
<td>Correction</td>
<td>2000.01.11</td>
<td>Douglas Gilbert</td>
<td><a href="mailto:dgilbert@724.com">dgilbert@724.com</a></td>
</tr>
<tr>
<td>157-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>Enhancement</td>
<td>2000.08.23</td>
<td>Thomas Petillon</td>
<td><a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
</tr>
<tr>
<td>Page</td>
<td>Severity</td>
<td>First Reported</td>
<td>Corrected Printing #</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>----------------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>154</td>
<td>Enhancement</td>
<td>1999.12.08</td>
<td>hps</td>
<td>2</td>
</tr>
<tr>
<td>160</td>
<td>Typo</td>
<td>2000.05.14</td>
<td>hps</td>
<td>—</td>
</tr>
<tr>
<td>160</td>
<td>Correction</td>
<td>2000.08.24</td>
<td>Andrew Koenig <a href="mailto:ark@research.att.com">ark@research.att.com</a></td>
<td>—</td>
</tr>
<tr>
<td>161</td>
<td>Enhancement</td>
<td>2000.06.22</td>
<td>hps</td>
<td>—</td>
</tr>
<tr>
<td>164</td>
<td>Typo</td>
<td>2000.09.15</td>
<td>Thomas Petillon <a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
<td>—</td>
</tr>
<tr>
<td>168</td>
<td>Typo</td>
<td>2000.05.24</td>
<td>Sam Lindley <a href="mailto:sam@redsnapper.net">sam@redsnapper.net</a></td>
<td>—</td>
</tr>
<tr>
<td>Page</td>
<td>Severity</td>
<td>First Reported</td>
<td>Corrected Printing #</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>----------------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 169  | Correction | 2000.08.24 | — | Change:  
“initialization is resource acquisition”  
To:  
“resource acquisition is initialization”  
(See the discussion for the corresponding erratum for page 37.)  
At the bottom of the page, change:  
Any copy assignment that *must* check for self-assignment is not exception-safe.  
To:  
Any copy assignment that is written in such a way that it *must* check for self-assignment is probably not strongly exception-safe.  
In the following Guideline, change:  
... an exception-safe copy assignment operator is automatically safe for self-assignment.  
To:  
... 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 | — | After the code example "T t(u);", change:  
This is direct initialization. The variable t is initialized directly from the value of u by calling T::T(u).  
To:  
Assuming u is not the name of a type, this is direct initialization. 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 | 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 | — | In the Guideline, change:  
Avoid declaring const pass-by-value function parameters.  
To:  
Avoid const pass-by-value parameters in function declarations. Still make the parameter const in the same function’s definition if it won’t be modified. |
| 179  | Typo | 1999.11.26 | hps | 2 | Vestigial plural.  
Change:  
(If, in looking for the “bonus” part, you said something about these two functions being un compilable—sorry, they’re quite legal C++. You were probably thinking of putting the const to the left of the & or *, which would have made the function body illegal.)  
To:  
(If, in looking for the “bonus” part, you said something about this function being un compilable—sorry, it’s quite legal C++. You were probably thinking of putting the const to the left of the *, 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 | 2 | The commentary for pa3 code line doesn’t take into account possible friendship.  
Change:  

<table>
<thead>
<tr>
<th>Page</th>
<th>Severity</th>
<th>First Reported</th>
<th>Corrected Printing #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>188</td>
<td>Typo</td>
<td>2000.08.13 Philip Brabbin</td>
<td>—</td>
<td>In Option 2, the #define directive is backwards.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="mailto:pabrabbin@hotmail.com">pabrabbin@hotmail.com</a></td>
<td></td>
<td>Change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>Typo</td>
<td>2000.08.12 hps</td>
<td>—</td>
<td>In the last line, change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…but it run correctly.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…but it will run correctly.</td>
</tr>
<tr>
<td>196</td>
<td>Typo</td>
<td>2000.01.01 George Reilly</td>
<td>2</td>
<td>At the bottom of the page, the Resize() function contains a spurious memset() call that is incorrect and was not in the original question.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="mailto:george@reilly.org">george@reilly.org</a></td>
<td></td>
<td>Delete:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>197</td>
<td>Typo</td>
<td>2000.10.01 Thomas Petillon</td>
<td>—</td>
<td>In the expansion of the return statement, change:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="mailto:petillon@topic.fr">petillon@topic.fr</a></td>
<td></td>
<td>&quot;*, y = * ) ,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>To:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;*, used = * ) ,</td>
</tr>
<tr>
<td>201</td>
<td>Enhancement</td>
<td>2000.08.12 hps</td>
<td>—</td>
<td>C++ Report no longer exists, so remove it.</td>
</tr>
<tr>
<td>203</td>
<td>Typo</td>
<td>2000.07.17 hps</td>
<td>—</td>
<td>Nathan’s last name is Myers, not Meyers. In the Meyers97 reference, change “Meyers” to “Myers” in both places.</td>
</tr>
</tbody>
</table>
| Index | Enhancement | 2000.07.01 Scott Meyers | — | REPLACE INDEX: A more thorough index is included later in this errata document. It replaces the index originally included in the first two printings.
auto_ptr and Exception Safety

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

```cpp
// 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, 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:

```cpp
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:

```cpp
// 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:

```cpp
// Correct (finally!)
//
auto_ptr<String> f()
{
    auto_ptr<String> result = new String;
    *result = "some value";
    cout << "some output";
}
```

---

1 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 \texttt{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 \texttt{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 \texttt{String} will be automatically cleaned up as the temporary \texttt{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 \texttt{auto_ptr} in your daily work. \texttt{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
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)
cStr 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
compositional dependencies, 87
see also: \textit{Pimpl idiom}

compilation dependencies, 87
see also: \textit{Pimpl idiom}

compilation firewalls, see: \textit{Pimpl idiom}

complex, 79
example use, 76

Complex, 69-75
composition, see: \textit{containment}
conditions, short-circuit evaluation and, see: \textit{control flow}

const, 175-181
see also: \textit{mutable}

auto\_ptr idiom, 158
cast, see: \textit{const\_cast}
correctness, 175-181

iterator, see: \textit{iterator(s)}, \textit{const\_iterator}

member functions, 177-181
return value, 72-73

const correctness, see: \textit{const}, \textit{correctness}

\textit{const\_cast}, 176-187
undefined behavior and, 179
to work around absence of \textit{mutable}, 178
to work around \textit{const\_incorrect} third-party interfaces, 181-182

\textit{const\_iterator}, see: \textit{iterator(s)}, \textit{const\_iterator}

\textit{construct}, 41-42
used, 46, 48, 51

\textit{constructor}, see also: \textit{initialization}
conversion by, 19, 70
see also: \textit{conversions}, \textit{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: \textit{list}, \textit{vector}; etc.

\textit{auto\_ptr} and, 16-157
destructible, 38, 59
homogeneous, 13

containment, generic, 94-95
inference vs., 82, 89-96, 107-108
control flow, 60-68
exceptions and,

\textit{should never throw}, 28-29, 55-58
virtual, see: \textit{virtual destructor}

Dewhurst, Steve, xiii
dynamic type, see: \textit{type}, \textit{dynamic}
dynamic\_cast, 181-187
inheritance and, 185-186
dynamically allocated memory, see: \textit{memory and resource management}

E
Effective C++, see: \textit{not this book}
see also: \textit{this book}

Einstein, Albert, 107
Employee, 17-23
empty base class optimization, 91
encapsulation, 97-98
end(), dereferencing, see: \textit{iterators}, \textit{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
\textit{swap} and, 44, 47, 59
strong guarantee, 16, 38, 59

\textit{iterator} invalidation and, 44, 47, 59

\textit{multiple side effects} and, 64-68
not always necessary, 67

\textit{performance overhead} and, 67

\textit{history of}, 25

\textit{object lifetime} and, 163-165

\textit{overhead} and, 60

placement \textit{new} and \textit{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

\textit{swap} and, 16, 43-44, 59, 170-171

\textit{throw()} specification, 53

\textit{try/catch} and, 39, 50

exception specifications, 54

Exceptional C++,
Errata for Exceptional C++
Updated 2000.12.12  page 16

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 hiding
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, Kevin, 10
hiding names,
see: name hiding
Hyslop, Jim, xiii

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
iosfwed, 101
iosstream, 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
cost_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
Machachielli, 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
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

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++
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

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
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(a)
string(s),
case-insensitive comparison, 4-9
comparison, 4-9
done in object or function, 6-7
c_str vs. implicit conversion tochar*,
162-163
Stroustrup, Bjarne, xi, xiii, 162, 181-182
Sunner, 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
eletion by compiler, 190-191
exceptions and, 62-63
modifying, 2-3
of built-in type, 2-3
pass by value and, 18, 71
recomputation and, 18
return-by-value and, 20-21
terminate, 28
test != other, 159-161
see also: assignment, self-assignment
this book,
see: Exceptional C++
throw,
see: exception safety
toupper,
example use, 5-6
traits, 4-9
ture,
see: bool
try,
see: exception safety
type,
der
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
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
strncpy,
see: string, comparison
string,
see: basic_string, string(a)
string(s),
case-insensitive comparison, 4-9
comparison, 4-9
done in object or function, 6-7
c_str vs. implicit conversion tochar*,
162-163
Stroustrup, Bjarne, xi, xiii, 162, 181-182
Sunner, 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
eletion by compiler, 190-191
exceptions and, 62-63
modifying, 2-3
of built-in type, 2-3
pass by value and, 18, 71
recomputation and, 18
return-by-value and, 20-21
terminate, 28
test != other, 159-161
see also: assignment, self-assignment
this book,
see: Exceptional C++
throw,
see: exception safety
toupper,
example use, 5-6
traits, 4-9
ture,
see: bool
try,
see: exception safety
type,
der
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
element 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 —