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 with a subject line containing the words "XC++ Errata."

Severity Category	# Entries
Format (change to page layout or text formatting only)	3
Typo (correction of simple typographical errors, cut-and-paste errors, and dyslexic mistakes)	28
Enhancement (addition of new or clarifying material)	22
Correction (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.

		First Reported			
Page	**Severity**	**Date**	**By**	Corrected Printing #	**Description**
xii	Enhancement	2000.04.14	Michel Michaud micm19@mail2.cstjean.qc.ca	—	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. 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 dxc@xprt.net	—	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_string`s 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 hinnant@metrowerks.com	—	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
			bkloss@novalis2.demon.co.uk
				—	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:
					`copy( v.begin(), v.end(), w.begin() );`
					
					To:
					`copy( v.begin(), v.begin()+4, w.begin() );`

<table>
<tr><th rowspan="2">Page</th><th rowspan="2">Severity</th><th colspan="2">First Reported</th><th rowspan="2">Corrected Printing #</th><th rowspan="2">Description</th></tr>
<tr><th>Date</th><th>By</th></tr>
<tr><td>16-7</td><td>Correction</td><td>1999.12.28</td><td>Tim Butler
tim@indra.com</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.</td></tr>
<tr><td></td><td></td><td>2000.02.21</td><td>Klaus Ahrens
ahrens@informatik.hu-berlin.de</td><td>2</td><td>Also, the non-default constructor is missing a memory allocation.</td></tr>
<tr><td></td><td></td><td>2000.08.13</td><td>Howard Hinnant
hinnant@metrowerks.com</td><td>—</td><td>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</code>/<code>catch</code>.

Change:
<pre>
template<typename O, size_t osize>
fixed_vector(const fixed_vector<O,osize>& other)
{
  copy(other.begin(),
        other.begin()+min(size,osize),
        begin());
}
</pre>
To:
<pre>
template<typename O, size_t osize>
fixed_vector(const fixed_vector<O,osize>& other)
  : v_(new T[size])
  { try {copy(other.begin(), other.begin()+min(size,osize), begin());}
    catch(...) { delete[] v_; throw; } }
fixed_vector(const fixed_vector<T,size>& other)
  : v_(new T[size])
  { try {copy(other.begin(), other.begin()+min(size,osize), begin());}
    catch(...) { delete[] v_; throw; } }
</pre>
And change:
<pre>
template<typename O, size_t osize>
fixed_vector<T,size>&
operator=(const fixed_vector<O,osize>& other)
{
  fixed_vector<T,size> temp(other); // does all the work
  Swap(temp);   // this can't throw
  return *this;
}
</pre>
To:
<pre>
template<typename O, size_t osize>
fixed_vector<T,size>& operator=(const fixed_vector<O,osize>& other) {
  fixed_vector<T,size> temp(other); // does all the work
  Swap(temp); return *this;   // this can't throw
}
fixed_vector<T,size>& operator=(const fixed_vector<T,size>& other) {
  fixed_vector<T,size> temp(other); // does all the work
  Swap(temp); return *this;   // this can't throw
}
</pre></td></tr>
<tr><td>21</td><td>Enhancement</td><td>2000.10.01</td><td>Thomas Petillon
petillon@topic.fr</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:

Change:
<pre>
const string&
FindAddr(/* ... */)
</pre>
To:
<pre>
const string&
FindAddr(/* pass emps and name by reference */)
</pre>
And change:
<pre>
if(/* found */)
</pre>
To:
<pre>
if(i->name == name)
</pre></td></tr>
<tr><td>31</td><td>Correction</td><td>2000.08.13</td><td>Howard Hinnant
hinnant@metrowerks.com</td><td>—</td><td>At the end of the paragraph numbered “2.” change:
…must be unchanged.

To:
…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 <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>.</td></tr>
</table>

Page	Severity	First Reported Date	First Reported By	Corrected Printing #	Description
					In one place on page 32 and four places on page 33, change: `Size` To: `Count`
37	Enhancement	2000.01.25	Marc Briand mbriand@mfi.com	2	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.
	Correction	2000.08.24	Andrew Koenig ark@research.att.com Bill Wade wrwade@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 *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 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.) In the Guideline, change: *"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.* To: *"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.* In the next paragraph, change: … cannot be exception-safe. To: … is probably not strongly exception-safe.
38	Enhancement	2000.02.10	Dave Abrahams abrahams@mediaone.net	—	URL moved, http://www.metabyte.com/~fbp/stl/eh_contract.html is now http://www.stlport.org/doc/exception_saf ety.html
42	Correction	2000.08.13	Howard Hinnant hinnant@metrowerks.com	—	The box implies that the helper functions `construct()` and `destroy()` are standard, when they aren't. In the first paragraph, change: …use three helper functions that are directly drawn (or derived in spirit) from the standard library: To: …use three helper functions, one of which (`swap()`) also appears in the standard library: Delete the final paragraph: 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, 55, 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 `destroy(FwdIter,FwdIter)` version is templatized 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

		First Reported			
Page	**Severity**	**Date**	**By**	Corrected Printing #	**Description**

					generality of templatizing 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.

43	Typo	1999.12.23	Steve Vinoski vinoski@iona.com	2	In paragraph 2, change: StampImpl<T> To: StackImpl<T>
46, 58	Typo	2000.05.24	Sam Lindley sam@redsnapper.net	—	In the Guideline, I say "initialization is resource acquisition" instead of "resource acquisition is initialization." Change: *"initialization is resource acquisition"* To: *"resource acquisition is initialization"*

48	Enhancement	2000.06.16	Stan Brown
			brahms@mindspring.com
				—	To make it more obvious that other is passed by value and hence already a temporary object, change other to temp and add more explanation.
					
					Change:
					If you're one of those folks who like terse code, you can write the operator=() canonical form more compactly as:
					Stack& operator=(Stack other)
					{
					Swap(other);
					return *this;
					}
					
					To:
					If you're one of those folks who like terse code, you can write the operator=() canonical form more compactly using pass-by-value to create the temporary:
					Stack& operator=(Stack temp)
					{
					Swap(temp);
					return *this;
					}

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.

		First Reported			
Page	**Severity**	**Date**	**By**	Corrected Printing #	**Description**
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 kalb@libertysoft.com	2	In paragraph 2, "addtion" should be "addition."
70	Typo	2000.01.06	Douglas Gilbert dgilbert@724.com	2	In the last line, "Item 19" should be "Item 39."
72	Correction	2000.01.16	Eric Nagler 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" (*C/C++ Users Journal*, 18(2), February 2000) and *Exceptional C++*'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 `new`, `new[]`, `delete`, and `delete[]` be static members. What is shown below should now be correct. Change: – *Unary operators are members.* – `= () []` *and* `->` *must be members.* – *The assignment operators (*`+= -= /= *=` *and so forth) must be members.* – *All other binary operators are nonmembers.* To: *The standard requires that operators* `= () []` *and* `->` *must be members, and class-specific operators* `new`, `new[]`, `delete`, *and* `delete[]` *must be static members. For all other functions:* *if the function is* `operator>>` *or* `operator<<` *for stream I/O,* *or if it needs type conversions on its leftmost argument,* *or if it can be implemented using the class's public interface alone,* *make it a nonmember (and friend if needed in the first two cases)* *if it needs to behave virtually,* *add a virtual member function to provide the virtual behavior,* *and implement it in terms of that* *else* *make it a member.*
82, 90	Enhancement	2000.09.07	Thomas Petillon 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
			petillon@topic.fr
				—	(For more details see the comment for pages 153-154.) In most of the book I demonstrate the Pimpl idiom using `struct` 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:
					
					Change:
					`class GenericTableAlgorithmImpl* pimpl_; // MYOB`
					
					To:
					`struct GenericTableAlgorithmImpl* pimpl_; // MYOB`

84, 87	Typo	2000.01.12	Steve Vinoski 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 tom.groszko@ait-mmii.com	—	URL moved, www.oma.com is now www.objectmentor.com.
100	Typo	2000.09.07	Thomas Petillon petillon@topic.fr	—	The first line "using namespace std;" is redundant and shouldn't be there. Remove the first line: `using namespace std;`
103	Typo	1999.12.27	Kjell Swedin kjells@wrq.com	2	In the last line, "Lokos96" should be "Lakos 96."

Page	Severity	First Reported: Date	First Reported: By	Corrected Printing #	Description
106	Enhancement	2000.09.07	Thomas Petillon petillon@topic.fr	—	The sense should be understood, but to be consistent with page 102, in the second paragraph of Item 28: Change: …so that existing code that uses `X` is unaffected. To: …so that existing code that uses `X` is unaffected beyond requiring a simple recompilation.
107	Enhancement	2000.02.10	M. Thomas Groszko tom.groszko@ait-mmii.com	—	URL moved, www.oma.com is now www.objectmentor.com.
107	Typo	2000.01.12	Steve Vinoski vinoski@iona.com	2	In the first Solution paragraph, I refer to a future Item that is actually earlier in the book. Change: I'll save the whole lecture for a later Item, but my bottom line is simply that… To: See Item 24 for the whole exhausting lecture; the bottom line is simply that…
108	Correction	2000.04.30	Brian Danilko bdanilko@formalsolutions.com.au	—	A forward declaration is still needed for class `B`, because `B` is still mentioned in some function declarations. In the first paragraph, delete the text: and in order to get rid of the `b.h` header entirely, In the code, before the line "`class C;`" insert a new line: `class B;`
110	Typo	2000.09.01	Tetsuroh Asahata 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: (rather than `#include` the class's actual declaration, To: (rather than `#include` the class's actual definition,
110	Enhancement	2000.12.05	John McGuinness John_McGuinness @Mastercard.com	—	In Option 2, change: *Option 2 (Score: 10 / 10): Put all private members into* `XImpl`*.* To: *Option 2 (Score: 10 / 10): Put all nonvirtual private members into* `XImpl`*.* Change the following paragraph: There are some caveats, the first of which is the reason for my "almost" above. To: There are some caveats. In the following bullet, change the paragraph: 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. To: 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).
113	Typo	2000.03.21	Klaus Ahrens ahrens@informatik.hu-berlin.de	—	In the paragraph following the output `1` and `8`, "`X2`" should be "`X`." Change: …inside each `X2` object… To: …inside each `X` object…

Page	Severity	First Reported: Date	First Reported: By	Corrected Printing #	Description
118	Typo	2000.09.07	Thomas Petillon petillon@topic.fr	—	Footnote 10 should not exist. Delete it.
124	Typo	2000.01.12	Steve Vinoski vinoski@iona.com	2	In the third-to-last line, “an” should be “and.”
127	Enhancement	2000.09.07	hps	—	At the top of the page, I present two options. Technically there’s one more option: A using declaration for `std::operator<<()`. Change: …have to write either “`std::operator<<( std::cout, hello );`” which is exceedingly ugly, or “`using namespace std;`” which dumps all the names in std into the current namespace and thus eliminates much of the advantage of having namespaces in the first place. To: …have to write either “`std::operator<<( std::cout, hello );`” which is exceedingly ugly, or “`using std::operator<<;`” which is annoying and quickly becomes tedious if there are many operators, or “`using namespace std;`” which dumps all the names in std into the current namespace and thus eliminates much of the advantage of having namespaces in the first place.
127	Typo	2000.08.12	hps	—	In the footnote, “article” should be “Item.” Change: …later in this article. To: …later in this Item.
132	Typo	2000.01.12	Steve Vinoski vinoski@iona.com	2	In the paragraph starting “Finally,” change: …is a member function of `a`: To: …is a member function of A:

133	Typo	2000.10.01	Thomas Petillon
			petillon@topic.fr
				—	At the bottom of the page, change:
					`// Example 1: Will this compile?`
					To:
					`// Example 2: Will this compile?`

141	Enhancement	1999.12.08	hps	2	Needs to stress more strongly that “heap” and “free store” are commonly used terms, not words from the standard. At the end of the final sentence, add: ; in particular, “heap” and “free store” are common and convenient shorthands for distinguishing between two kinds of dynamically allocated memory

149	Typo	1999.12.08	hps	2	In the code example, change:
					`new (shared) T; // if T::T() throws, memory is leaked`
					To:
					`new (shared) Y; // if Y::Y() throws, memory is leaked`

151, 157-8	Correction	2000.01.11	Douglas Gilbert dgilbert@724.com	2	At the top of page 151, toward the end of the code example, add “`/*...*/`” after “`private:`”. At the bottom of page 157, add “`/*...*/`” after “`private:`”. At the top of page 158, add the following new paragraph: Possible issue: One of the `/*...*/` areas (whether `public`, `protected`, or `private`) had better include at least declarations for copy construction and copy assignment.
153, 154	Enhancement	2000.08.23	Thomas Petillon petillon@topic.fr	—	It’s perfectly legal and standards-conforming to forward-declare a `class` as a `struct` 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 `class` and a `struct`

		First Reported			
Page	**Severity**	**Date**	**By**	Corrected Printing #	**Description**

					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.
					
					Specifically, in most of the book I demonstrate the Pimpl idiom using `struct` 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):
					
					Change:
					`class C::CImpl { /* ... */ };`
					
					To:
					`struct C::CImpl { /* ... */ };`

154	Enhancement	1999.12.08	hps	2	In the first paragraph following Example 4(b), after the first sentence add: Better still, it means `C::C()` has to do less work to detect and recover from constructor failures because `pimpl_` is always automatically cleaned up.
158	Enhancement	1999.12.08	hps	—	ADD NEW MATERIAL: Before "The const auto_ptr Idiom" add the new section "auto_ptr and Exception Safety," included later in this errata document.
160	Typo	2000.05.14	hps	—	In the second guideline, the word "to" is missing. Change: It's all right use a … To: It's all right to use a …
160	Correction	2000.08.24	Andrew Koenig ark@research.att.com	—	(See the discussion for the corresponding erratum for page 37.) In the second paragraph, change: If `T::operator=()` 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? To: If `T::operator=()` is written using the create-a-temporary-and-swap idiom (see page 47), it will be both strongly exception-safe and not *have* 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? 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.*
161	Enhancement	2000.06.22	hps	—	In the footnote, at the end of the final paragraph ("Yikes…"), append: 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: `T t = t; // invalid, but it would make the test meaningful`
164	Typo	2000.09.15	Thomas Petillon petillon@topic.fr	—	In the last paragraph, change: Here, the slicing issue is that `t.f()` replaces… To: Here, the slicing issue is that `t.DestroyAndReconstruct()` replaces…
168	Typo	2000.05.24	Sam Lindley sam@redsnapper.net	—	In the last line, I say "initialization is resource acquisition" instead of "resource acquisition is initialization."

Page	Severity	First Reported: Date	First Reported: By	Corrected Printing #	Description
					Change: “initializat ion is resource acquisition” To: “resource acquisition is initialization”
169	Correction	2000.08.24	Andrew Koenig ark@research.att.com Bill Wade wrwade@swbell.net	—	(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 mhandy@neonsoft.com	—	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 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 dickrp@EE.Princeton.EDU	—	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 uncompilable—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 uncompilable—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 chris@atdesk.com	2	The commentary for pa3 code line doesn’t take into account possible friendship. Change:

Page	Severity	First Reported: Date	First Reported: By	Corrected Printing #	Description
					Error: Because `b1` IS-NOT-AN `A` (because `B` is not publicly derived from `A`; its derivation is private), this is illegal. To: 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`.

188	Typo	2000.08.13	Philip Brabbin
			pabrabbin@hotmail.com
				—	In Option 2, the `#define` directive is backwards.
					
					Change:
					`#define int   bool`
					
					To:
					`#define bool  int`

195	Typo	2000.08.12	hps	—	In the last line, change: …but it run correctly. To: …but it will run correctly.
196	Typo	2000.01.01	George Reilly george@reilly.org	2	At the bottom of the page, the `Resize()` function contains a spurious `memset()` call that is incorrect and was not in the original question. Delete: `memset( buffer_, ' ', newSize );`

197	Typo	2000.10.01	Thomas Petillon
			petillon@topic.fr
				—	In the expansion of the return statement, change:
					`", y = " ) ,`
					
					To:
					`", used = " ) ,`

201	Enhancement	2000.08.12	hps	—	*C++ Report* 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.
Index	Enhancement	2000.07.01	Scott Meyers smeyers@aristeia.com	—	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:

```
// 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,[1] 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”;
```

[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 `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.

D

E

— end of document —