Scott Meyers

Modification History and Errata List for *Effective Modern C++*

Last Updated 7 October 2019
by Scott Meyers

What follows are my notes on what I've changed in *Effective Modern C++* since its official publication in November 2014 (i.e., excluding "Early Release" and "Rough Cuts" versions) and what I believe may need to be changed in the future. Most of the changes (or prospective changes) are cosmetic and don't affect the technical content of the book. Entries that *do* affect technical material are preceded with an exclamation mark ("!") and are displayed in red.

Each entry includes the following information:

- **Date Reported:** The date the problem was brought to my attention.
- **Who:** The initials of the person who caused me to become aware of the problem. The initials "sdm" refer to me. The mapping from other initials to full names is at the bottom of this page.
- **Platform:** The format(s) in which the problem occurs, i.e., PDF, ePub, Kindle (mobi), or Safari Books Online. PDF includes print books. As a general rule, content problems affect all platforms, but formatting issues are platform-specific.
- **Location:** Where in the book the erratum exists. This is typically either the PDF page number(s) or the Item number.
- **What:** A description of the problem and the solution.
- **Date Fixed:** The date on which the problem was fixed. If no date is shown, it means it hasn't yet been addressed.

I am always interested in bug reports and suggestions for improvements to *Effective Modern C++*. To submit comments, send email to emc++@aristeia.com.

The following issues have been addressed in published versions of the book:

DATE REPORTED	WHO	PLATFORM	LOCATION	WHAT	DATE FIXED
1/13/15	sdm	Digital	Metadata	Digital TOCs should included a link to the book's copyright page.	4/ 2/15
12/ 7/14	kxh	ePub on Marvin	Metadata	Author is shown as "(Unknown author)".	3/31/15
12/16/14	rxk sdm	All	All	When words in code font are broken across lines, they're not hyphenated.	5/19/15
11/18/14	sdm	Kindle	All	Font size in code displays should be increased to match size in the rest of the book.	1/20/15
11/18/14	lxs	Kindle	All	Inter-paragraph spacing should be increased.	1/20/15
11/28/14 *1/ 7/15*	kxa bxy	Kindle on Nexus	All	Highlighted code text (i.e., red and/or italicized) is invisible. Wide and narrow ellipses are swapped. (Problems reported on Nexus 4, 5, 7, and 10.)	4/14/15
11/18/14	sdm	PDF	vii	Item title for Item 8 is incorrectly indented.	1/20/15
4/17/15	sdm	PDF	xiv	Twice in first para, bad line break between day and month in a date.	5/ 8/15
4/17/15	sdm	PDF	xiv	Near end of first para, sentence-ending period should precede the parenthesis, not follow it.	5/ 8/15
4/17/15	sdm	PDF	xiv	Add tyk for performing an extraordinarily thorough post-publication review of the book.	1/25/16
1/ 9/16	sdm	PDF	xv 310 315	Ashley Morgan Williams ==> Ashley Morgan Long	1/25/16
11/15/14	kxv	PDF	xv	"reviwing" ==> "reviewing"	1/20/15
11/21/14	sdm	Safari	5	In last line of first code display, final parenthesis in	3/31/15

		Online		comment wraps to next line.	
!	1/30/15	txk	PDF	12	The third paragraph (beginning with "These examples all show lvalue reference parameters, but type deduction works exactly the same way for rvalue reference parameters") should be removed. When the type of param in the template f on page 11 is changed to an rvalue reference (i.e., to have type "T&&"), it becomes a universal reference, and then the rules for Case 2 (on pages 13-14) apply.	3/24/15

	11/22/14	sdm	PDF	21-22	In November 2014, N3922 was adopted for draft C++17. With this change to the language, it is no longer the case that the following two statements do the same thing:
					
					`auto x3 = { 27 };`
					`auto x4{ 27 };`
					
					Under N3922, x4 is of type int, not std::initializer_list<int>. Some compilers have already implemented this prospective language change. I added a footnote to this effect.
						3/25/15

	3/18/16	gxk	PDF	21	In the footnote added to resolve an earlier erratum, remove the cross reference to Item 42.	1/ 5/17
	12/19/16	sdm	PDF	21	In footnote, "some compilers" => "many compilers".	1/ 5/17
	3/26/15	sdm	PDF	24	In first line, "Occasionally" should be followed by a comma.	3/26/15
	2/ 9/15	tyk	PDF	24 28	Clarify that in authAndAccess, we're interested only in supporting containers taking *numeric* index values. (This excludes std::map and std::unordered_map, which take indices of arbitrary types).	1/25/16
	12/13/14	rxm	PDF	25	I should mention that for functions with deduced return types, if there is more than one return statement, all return statements must yield the same deduced type.	3/25/15
	3/27/15	sdm	PDF	26-27	Bad page break between the comment and the code it applies to.	3/30/15
	1/22/16	sdm	PDF	28	Improper indentation of "-> decltype(...)".	1/25/16
	2/ 2/15	mxw	PDF	28	Reworded last para on page to reflect that not all names are lvalues. Also added weasel wording regarding "lvalue expressions more complicated than names" to try to avoid contradicting 7.1.6.2/4 bullet 1 without getting sucked into the details.	3/25/15
	2/17/15	mam	PDF	31	When attempting to instantiate an undefined template, Intel C++ 15.0.2 on Linux issues a diagnostic that fails to mention the type being used for the instantiation.	3/24/15
	1/ 6/16	sdm	PDF	31	Too much blank space above first code example.	1/25/16
	12/ 1/14	daa	PDF	33	"the type that all three compilers report for param are" ==> "the type that all three compilers report for param is"	1/20/15
	11/21/14	sdm	PDF	34	"neither are IDEs or templates" ==> "neither are IDEs nor templates"	1/20/15
	11/17/14	txk	PDF	34	Boost's web site is boost.org, not boost.com.	1/20/15
	11/23/14	bbs	PDF	37-38	In the dwim functions, it would be more ideomatic to use a for loop instead of a while loop.	3/25/15
	2/ 8/15	mxm	PDF	38	In comment next to initialization of x3, "x's value is well-defined" ==> "x3's value is well-defined"	3/24/15
	1/ 6/17	sxa	PDF	41	In last para, xref to Items 2 and 6 should include Item 3, too.	5/18/18
	2/ 8/17	sdm	PDF	41	At beginning of 4th line of second para, "const" should be italicized, i.e., it should start with "<*const* std::string, int>". (This error exists only in the PDF and printed versions of the book and only in the 8th-10th printings. Kindle and epub versions of the book don't have this mistake, nor do earlier versions in PDF and print.)	1/ 5/17
	1/ 6/17	sxa	PDF	45	At end of 2nd para, "Pantheon" => "pantheon".	5/18/18

	1/25/15	txk	PDF	47	In comment of penultimate code block, "impliclitly" ==> "implicitly".	3/24/15
	11/23/14	bbs	PDF	47-48	In last para, I should make explicit my assumption that the container isn't empty.	3/30/15
!	11/23/14	bbs	PDF	48	Twice on page, "d * c.size()" ==> "d * (c.size() - 1)".	1/20/15
!	3/26/15	lxz	PDF	48	The 1/20/15 fix for the code on this page resolved the problem (present in early printings of the book) that when d==1.0, index would be c.size(), hence out of bounds. Unfortunately, it also resulted in index being c.size()-1 (the last element of the container) only when d==1.0.	4/ 3/15
	2/18/15	vhg	PDF	51	In second paragraph, I say that narrowing conversions using braced initializers won't compile, but the Standard requires only that compilers issue some kind of diagnostic message. A warning fulfills this requirement, and some compilers issue only warnings for at least some narrowing conversions using braced initializers.	3/24/15
	12/19/16	sdm	PDF	54-55	First line of page 55 should be moved to page 54 to keep the comment together.	1/ 5/17
!	11/20/16	cxl	PDF	55 314	Comment on last line on page is incorrect: the constructor for w5 is called with a one-element std::initializer_list, not an empty one. For details, consult this blog post.	1/ 5/17
	2/12/15	sdm	PDF	58	Formatting of Item title is incorrect: too much space after the colon.	3/24/15
	12/21/14	tjb	PDF	61	lockAndCall implementations should include this line at the beginning: `using MuxGuard = std::lock_guard<MuxType>;`	1/20/15
	4/21/16	gxp	PDF	71	2nd para suggests that it's possible to specify the underlying type only for unscoped enums. Reword.	1/ 5/17
	2/23/15	mxk	PDF	77	In next-to-last para, wchar_t, char16_t, and char32_t are keywords, hence should not be std::-qualified.	3/24/15
	7/16/16	axd	PDF	77	Penultimate para fails to mention volatile-only overloads (i.e., volatile void*, volatile char*).	1/ 5/17
	11/17/15	rxh bxc sdm	PDF	81	Of the three compilers I use for my testing (gcc, MSVC, Clang), only one (gcc) issues no warnings when full warnings are enabled (i.e. -Wall -Wsuggest-override).	1/25/16
	11/12/16	hxb	PDF	81-82	Derived::mf4 should be declared virtual.	1/ 5/17
	11/23/14	bbs	PDF	83-85	I should mention that if a member function is reference-qualified, any other overload of the same member function (i.e., matching name and parameter types) must also be reference-qualified.	3/26/15
	12/25/14 1/27/15	mxs tyk	PDF	85	A better way to have the rvalue reference overload of the "data" member function return an rvalue is to have it return an rvalue reference. That would avoid the creation of a temporary object for the return value, and it would be consistent with the by-reference return of the original "data" interface near the top of page 84.	3/26/15
	11/12/16	hxb	PDF	86 88	Missing space after first comma in page's first call to std::find.	1/ 5/17
	1/22/16	sdm	PDF	88	Missing spaces around "->" in return type for cbegin at bottom of page.	1/25/16
	1/22/16	sdm	PDF	89	In first para, incorrect font for final "r" in "const_iterator".	1/25/16
	1/ 4/15	jxz	All	PDF pg 95	At end of sentence before code display, space between "seems" and "appropriate" is too wide. In ePub version, there are two spaces, but in PDF and Mobi versions, there's a single overly-wide space.	1/20/15
	1/18/15	bbs	PDF	99-100	In pow, exp should be declared unsigned, because the algorithm works only for non-negative exponents.	1/25/16
!	12/24/14	akb	PDF	105	Neither std::mutex nor std::atomics are move-only. They	3/25/15

		106 108	are neither copyable nor movable, so classes containing them are rendered both uncopyable and unmovable.		
	11/14/16 sdm PDF	105	In antipenultimate line on page (i.e., 3rd-to-last line— I just wanted to use the word "antipenultimate" :-}), "i.e," => "i.e.,".		1/ 5/17
	1/ 4/15 jxz PDF	106	It would be better to use std::hypot to compute the distance from the origin, because (1) std::hypot does exactly what's needed and is part of the standard library and (2) as this blog post explains, it handles potential overflow values much better than the code shown in the book.		3/25/15
	12/12/14 bxr PDF	107	"CacheValid" ==> "cacheValid".		1/20/15
!	12/12/14 bxr PDF	107	The text above the code display near the middle of the page says that reversing the order of the assignments solves the problem, but that's incorrect. Multiple threads could still perform the expensive calculations before cacheValid is set.		1/20/15
	8/31/16 txb PDF	112	In antipenultimate line (see above) of paragraph above first code display, "i.e," => "i.e.,".		1/ 5/17
	3/25/15 gxs PDF	115	At end of second bullet in "Things to Remember," "and a destructor" ==> "or a destructor".		3/25/15
	1/24/17 djm PDF	115	Reword last sentence of bullet at top of page to improve clarity. Also undo 3/25/15 change suggested by gxs, i.e., at end of second bullet in "Things to Remember," "or a destructor" ==> "and a destructor".		5/18/18
	3/ 6/17 djm PDF	115	In last sentence of third bullet in box, "with an explicitly declared destructor" ==> "with an explicitly declared copy operation or destructor".		5/18/18
	6/ 2/15 lxz PDF	118	In fourth para, "it's not possible" ==> "it wasn't possible". (In C++11, std::auto_ptrs can be stored in containers.)		10/21/15
	8/31/16 ryk PDF	118	"Item 17" in footer should be Item 18.		1/ 5/17
	6/19/16 axd PDF	123	The book's claim that stateless function object deleters incur no size penalty is true in practice, but it's not guaranteed by the Standard. Reword. Be caseful to avoid implying that stateless function objects *in general* have no size.		1/ 5/17
	6/23/15 dgw ePub	119 124 127 135 138	In iBooks, the beginnings or endings of some sentences are missing. (This is a systemic problem. The list of affected pages is probably not complete.)		6/30/15
	6/ 2/15 lxz PDF	128	Text near end of the first paragraph implies that the control block may not contain a weak count, but, as I note on page 144 (in Item 21), the weak count is always present.		1/25/16
	12/15/14 sdm PDF	129	"on page 115" ==> "on page 117".		1/20/15
	1/18/15 bbs PDF	133	In first para, both copy- and move assignment can call for reference count manipulations, because the target of a move assignment must typically decrement the reference count of the control block it currently points to.		1/25/16
	6/10/17 tgm PDF	136	In line 7 of 2nd para, the final "s" in "std::weak_ptrs" is in wrong font.		5/18/18
	6/ 2/15 lxz PDF	139	Text in the second paragraph implies that std::make_unique supports custom deleters, but that is not the case.		1/25/16
	12/19/14 sdm PDF	151	Last line on page is indented too far.		1/20/15
	1/29/15 tyk PDF	152-153	std::unique_ptr's move operations are noexcept, so Widget's should be, too. Compiler-generated versions would be noexcept automatically (see below), but user-declared versions must be explicitly declared noexcept.		1/25/16
	1/ 8/16 sdm PDF	153	In first para, clarify that compilers must be able to generate code to destroy pImpl in the event of an exception, even if the move constructor is noexcept. (The source of this requirement is 12.6.2/10 in C++14, though		1/25/16

				there's a proposal to relax it.)	
!	12/18/14	ahp	PDF	154	The copy operations shown on this page unconditionally dereference pImpl, but an earlier move operation could have left pImpl null. The code should check for nullness and, if present, deal with it in a reasonable fashion.	3/26/15
	12/18/14	ahp	PDF	155	Using compiler-generated copy and move operations for Widget containing a pImpl of type std::shared_ptr would replace the value semantics of the class (i.e., deep copy) with reference semantics (i.e., shallow copy).	3/26/15
	11/12/16	hxb	PDF	158	"Item 22" in footer should be Item 23.	1/ 5/17
	1/ 4/15	jxz	PDF	160	Near top of page, declaration of Annotation constructor is missing terminating semicolon.	1/20/15
	1/ 6/16	nxj	PDF	163	Add "Things to Remember" bullet about how declaring objects const disables move semantics for them.	1/25/16
	12/16/14	rxm	PDF	168	In second line on page, "args is zero or more" ==> "params is zero or more".	1/20/15
	1/24/17	djm	PDF	168	In last sentence before box, "general relativity" => "Einstein's theory of relativity".	5/18/18
	2/11/15	mjs	PDF	171	The last sentence of the second paragraph is misleading, because for setters that are passed the "wrong" type argument, the performance gap between (a) overloading on lvalues and rvalues and (b) pass-by-universal-reference is determined by several factors: - Cost of constructing a temporary object of the "right" type from the argument; - Cost of moving the temporary into the data member; - Cost of assigning the argument to the data member (which may incur the costs of a temporary construction and destruction). I eliminated the sentence.	3/25/15
	1/30/15	tyk	PDF	171	Page should contain a cross reference to Item 41, which has a fuller discussion of the tradeoffs between the use of overloading and passing by universal reference.	1/25/16
	7/ 7/15	mys	PDF	179	In 2nd-to-last para, the overload taking a universal reference would deduce T to be short&, not short, because logAndAdd is being called with an lvalue (nameIdx).	1/25/16
	12/15/14	sdm	PDF	184	"on page 175" ==> "on page 177".	1/20/15
	12/15/14	sdm	PDF	185	"on page 177" ==> "on page 178".	1/20/15
	4/15/15	sdm	PDF	186	In first para, bad line break between "Item" and "26".	5/ 8/15
	4/15/15	sdm	PDF	187	In first para, bad line break between "Item" and "9".	5/ 8/15
	12/15/14	sdm	PDF	188	"on page 178" ==> "on page 180".	1/20/15
	12/11/14	kxv	PDF	192	"on page 206"==> "on page 183". Also, page number should be a link.	1/20/15
	2/22/15	vhg	PDF	192	Final paragraph says that std::is_base_of<T,T>::value is true, but that's the case only for class types. When T is a built in type such as int, std::is_base_of<T,T>::value is false.	3/24/15
	4/ 6/15	sdm	PDF	200	In first sentence, bad line break between "Item" and "9".	5/ 8/15
	12/19/14	ahp	PDF	210	This section also applies to declaration-only integral static constexpr data members (i.e., to constexpr data members as well as const data members).	3/24/15
	11/12/16	hxb	PDF	216	"Item 30" in footer should be Item 31.	1/ 5/17
	12/17/14	rxm	PDF	217	In first comment block, reference to Item 2 should be to Item 5.	1/20/15
	6/28/15	bvw	PDF	226 229	C++11 code examples on these pages use std::make_unique, but std::make_unique is a C++14 feature, not C++11	1/25/16
	2/10/15	sxt	PDF	Item 33	Inside the lambda, the call to func should be eliminated, because it plays no role in any of the examples.	3/30/15
	11/14/16	sdm	PDF	230	In second-to-last line preceding the last code display, "i.e," => "i.e.,".	1/ 5/17

2/10/15 sxt PDF 231-232 For consistency with the rest of the Item, the lambda parameter on page 231 should be named x, not param, and in the variadic version on page 232, it should be named xs, not params. 3/30/15

3/27/15 sdm PDF 232-233 Bad page break between the comment and the code it applies to. 3/30/15

! 11/30/14 rxb PDF 234 235 236 The revised versions of setSoundB still invoke steady_clock::now() when std::bind is called instead of when the bind object is called. A proper fix requires nesting a third call to std::bind inside the second one, e.g., as follows for the C++14 version of the code: 1/20/15

```
auto setSoundB =
  std::bind(setAlarm,
            std::bind(std::plus<>(),
                      std::bind(steady_clock::now),
                      1h),
            _1,
            30s);
```

! 5/ 5/15 mam PDF 235 With the code at the top of the page, calls to setSoundB won't compile, because std::plus<steady_clock::time_point>'s operator() declares two parameters of type steady_clock::time_point, but we're passing a time_point and a duration.

The fix is to essentially implement C++14's generic std::plus ourselves. Its templatized operator() function accepts arguments of any type. So: 1/25/16

```
struct genericAdder {
  template<typename T1, typename T2>
  auto operator()(T1&& param1, T2&& param2)
    -> decltype(std::forward<T1>(param1) + std::forward<T2>(param2))
  {
    return std::forward<T1>(param1) + std::forward<T2>(param2);
  }
};

auto setSoundB =
  std::bind(setAlarm,
            std::bind(genericAdder(),
                      std::bind(steady_clock::now),
                      hours(1)),
            _1,
            seconds(30));
```

3/ 9/16 jxm PDF 235 The lambda used to initialize setSoundL is missing the using directive for std::literals. 1/ 5/17

! 2/23/15 vhg PDF 239 Inside operator() for the closure boundPW, pw is const, because operator() is a const member function. However, PolyWidget::operator() is not const, so the call "pw(param)" attempts to invoke a non-const member function on a const object, and that's invalid. 3/25/15

3/23/15 sdm PDF 241 In first paragraph, "compiler-writers" ==> "compiler writers". 3/24/15

2/ 8/15 tyk PDF 241 Clarify that std::async isn't synonymous with "task-based programming," it's just an example of it. 1/25/16

1/ 6/15 oxb PDF 247 In paragraph above final code block, "std::launch::deferred" ==> "std::future_status::deferred" 1/20/15

5/23/15 pxz Kindle ~8526 The end of the paragraph beginning with "If f runs concurrently with the thread calling std::async (i.e., if the launch policy chosen for f is std::launch::async)" is not displayed. This problem was introduced in the 2015-05-08 release, i.e., it did not exist in prior releases. 6/30/15

1/ 1/15 fxf PDF 251 In second comment block, reference to Item 2 should be to Item 5. 1/20/15

7/26/16 mtb PDF 251 Clarify that terminating "program execution" means all the threads, i.e., the entire process. 1/ 5/17

11/19/14 dxj PDF 257 "isappropriate" ==> "is appropriate" 12/12/14 (print and epub)
1/20/15 (PDF, mobi, and Safari)

	4/ 9/15	sdm	PDF	269	Footnote should replace blog post reference with [this followup blog post]().	1/25/16
!	4/13/15	lxp	PDF	270	Italicized comment opposite first "…" is incorrect. ThreadRAII isn't being used in this example, so an exception in this region would cause joinable std::threads to be destroyed, hence would lead to program termination.	1/25/16
	3/ 1/16	jsb	PDF	271	2nd para of Item 40 says that "operations on [std::atomics] behave as if they were inside a mutex-protected critical section." For std::atomics using sequential consistency, this is true, but for std::atomics using weaker consistency models, it's only partially true. Regardless of the consistency model, std::atomics appear to other threads to change values atomically, but std::atomics using weaker models permit some instruction reorderings that would not be permitted if mutexes were used. (As the footnote on page 274 implies, *Effective Modern C++* assumes the use of sequential consistency throughout the book.)	1/ 5/17
	11/12/16	hxb	PDF	272	In 4th line from bottom "e.g," => "e.g.,".	1/ 5/17
	6/19/16	axd	PDF	273	Penultimate para suggests that compilers can detect data races and use the leeway from the resulting undefined behavior to generate code to do anything they like. This is misleading. Code generation occurs during compilation, but data races occur at runtime. Reword.	1/ 5/17
	7/17/15	sdm	PDF	274	In 2nd line of footnote, "that use the syntax" ==> "that uses the syntax".	10/21/15
	12/15/14	sdm	PDF	281	"Recall on page 2" ==> "Recall from page 4".	1/20/15
	6/19/16	axd	PDF	278	2nd para says "the code must be written like this", but that's not true. Instead of invoking std::atomic<int>::store on y, assignment can be used: `y = x.load();  // same as y.store(x.load());`	1/ 5/17
	1/11/15	pxv	PDF	283	In final paragraph, "addName will be copy constructed" ==> "newName will be copy constructed".	1/20/15
	1/23/15	kxv	PDF	289 290	Near middle of page, saying "changeTo's use of assignment to copy the parameter newPwd probably causes that function's pass-by-value strategy to explode in cost" is misleading. It suggests that the problem is that changeTo uses assignment, but that's how changeTo has to be implemented. The real problem is that changeTo uses pass by value for its parameter. At the bottom of the page, I show how using pass by reference-to-const eliminates the problem.	3/25/15
	1/23/15	kxv	PDF	292	In the "Things to Remember" box, the second bullet point is unclear and should be reworded.	3/25/15
	11/24/14	sdm	PDF	299	Line 6 is unnecessarily indented.	1/20/15
	11/25/14	sdm	PDF	300	In the text following first code fragment, the constraint on the pointer passed to the std::regex constructor is incorrect. It's that the pointer be non-null, not that it point to a valid regular expression. (Null pointers yield undefined behavior. Pointers pointing to invalid regular expressions yield exceptions.)	3/24/15
	8/ 3/15	sdm	PDF	303	Allusion to "Adventure" should refer to page 292, not page 295.	10/21/15
	9/ 7/15	kyh	PDF	308	In second column, entries for std::basic_ios::basic_ios and std::basic_ios::operator= should refer only to page 75, not both 75 and 160.	10/21/15
	2/19/18	sdm	PDF	310	In column 2, eliminate fourth line from bottom of page.	5/18/18
	6/19/16	sdm	PDF	313	std::is_constructible should refer to page 196, not 195.	1/ 5/17
	1/ 8/15	bvw	PDF	315	Bart Vandewoestyn ==> Bart Vandewoestyne	1/20/15

The following changes have been proposed, but have not yet been published:

	DATE REPORTED	WHO	PLATFORM	LOCATION	WHAT	
!	9/27/19	kst	PDF	21	C++98 permits scalar types to be initialized using the "*type var = { value };*" syntax, so the initialization of	

				x3 near the top of the page is valid in C++98. What C++11 did was extend this initialization syntax to non-scalar types.
12/24/18	dxk	PDF	77 79	Text on these pages refers to deleting instantiations or declaring them private, but it's actually the template *specializations* that are deleted or declared private, not the template instantiations.
4/ 9/19	fxr	PDF	96	In middle para on page, the statement that functions in the C++ Standard Library inherited from C are not declared noexcept is true, but 17.6.5.12/3 of the C++11 and C++14 Standards makes clear that implementations should treat them as if they were.
3/12/19	gpk	PDF	194	For consistency with the original code on page 180, remove the "…" from both constructor bodies on this page.
5/10/19	dxk	PDF	214	The second "Things to Remember" bullet should include declaration-only integral constexpr data members.
2/20/19	dxk	PDF	314	In first column, add page 294 to entry for std::vector::emplace_back.

What follows are interesting comments about the material in *Effective Modern C++*, but I don't expect to be able to modify the book to take these comments into account until (if ever) I write a second edition.

DATE REPORTED	WHO	PLATFORM	LOCATION	WHAT
1/21/15	sdm	PDF	All	The leading within paragraphs is sometimes inconsistent. This is because the code font is slightly taller than the non-code font, so lines containing text in code font get a bit more leading than those without. Reducing the size of the code font 5% fixes that problem, but it leads to a surprisingly large number of revised page breaks throughout the book, so we've decided to live with the inconsistent leading.
1/12/15	sdm	Kindle	All	The book's TOC is at the end of the book, rather than at the beginning. This is O'Reilly policy for digital books, because E-readers have their own intrabook navigation systems.
11/18/14	sdm	Kindle	All	On some platforms, spacing above or below code displays is excessive, (e.g., on Kindle app on iPad, there's too much space above code displays, whereas on Kindle for PC, there's too much space after code displays). I'm told "there are known inconsistencies within the Kindle device/app family, and there is no way to account for each variation." Not that that has prevented us from trying.
11/21/14	bxm	Kindle	All	Search functionality is...counterintuitive. Examples: Text to Search For — Kindle Hits — Actual Occurrences std::array — 0 — 27 std:: array — 25 — 0 std :: array — 0 — 0 #include <string> — 0 — 5 #include < string> — 0 — 0 #include < string > — 5 — 0 = delete — 2 — 12 = delete; — 10 — 10 If you're aware of a good description of how Kindle search works, please let me know. I'd like to link to it.
1/13/15	sdm	Kindle on PC	All	The book's digital TOC lacks entries for individual Items, but this is apparently a limitation of Kindle on PC. On other Kindle platforms, I'm told that the digital TOC does link to individual Items.
3/29/15	scp	Kindle on HTC One (M8)	All	Emphasized words (presumably italicized) are invisible. I'm told this is a problem with the Kindle app when using the default font (Caecelia). Choosing a different font should resolve the issue.
! 10/ 5/14	txn	All	Item 1	Case 1 applies only when *ParamType* is a non-universal reference. It doesn't apply when *ParamType* is a pointer type, because parameters of pointer type are treated like by-value parameters. (The pointer is passed by value.) So the three cases are really: Case 1: *ParamType* is a non-universal reference. Case 2: *ParamType* is a universal reference. Case 3: *ParamType* is a non-reference.

This is a simpler way of classifying template type deduction behavior: two major cases (reference parameters and non-reference parameters) and two sub-cases for one of the major cases (universal reference parameters and non-universal reference parameters).

12/18/14 csc PDF 24

In the final line of the code example, if v were a *const* vector, decltype(v[0]) would be *const* int&, because std::vector::operator[] is overloaded on const, and the const version has a reference-to-const return type.

7/ 5/17 rxz PDF 27, 87, 166, 192

Technically, parameters bind to arguments, but in some places in the book (as noted by the page numbers at left), I refer to arguments as binding to parameters (or, in some cases, as expressions as binding to variables). I believe the current wordings are pretty clear, and I'm concerned that trying to rewrite the explanations to be technically correct might make things worse, so I'm leaving as is the places where I refer to things binding to names instead of names binding to things.

3/17/15 fdb PDF 27-28

Except for std::vector<bool>, *container<T>*::operator[] for all standard library containers returns T&, which is an lvalue. Given an rvalue container rc, then, indexing into it (e.g., rc[i]) yields an lvalue, even though rc itself is an rvalue. Because authAndAccess with a decltype(auto) return type returns exactly what *container<T>*::operator[] does, an implication is that in the code defining s on page 27, s will be copy constructed, even though the element initializing it is in an rvalue std::deque<string>.

authAndAccess could be revised to return rvalues for rvalue container arguments, but the crux of the issue in this case is that operator[] for rvalue containers yields an lvalue. Given C++11's support for overloading on lvalue and rvalue objects (as described in Item 12 of EMC++), it'd be interesting to know why operator[] for an rvalue object doesn't return an rvalue reference (which, for function return types, is an rvalue).

1/26/15 tyk All Item 3

The behavior of these two ways of declaring a function template may not be the same:

```
template<typename T>
auto f()->decltype(expr)    // #1 -- explicit return type
{
  ...
  return expr;
}

template<typename T>
decltype(auto) f()          // #2 -- deduced return type
{
  ...
  return expr;
}
```

In #1, if *expr* is invalid, SFINAE will cause f to be removed from the candidate overload set, but in #2, if *expr* is invalid, f will be ill-formed, and compilation will fail. The same would be true for an auto return type. This means that it's not always possible to replace an explicit return type with a deduced return type.

(If the sentence with "SFINAE" makes no sense to you, you should probably not be writing templates with auto or decltype(auto) return types.)

6/19/16 axd All 38

The text suggests that it's always a good idea to initialize variables, but there are cases where it makes sense to leave variables uninitialized (e.g. as described here).

11/22/14 tyk All Item 6

A noteworthy difference between the explicity typed initializer idiom (ETII)'s

```
auto var = static_cast<type>(expr);
```

and this auto-free alternative,

```
type var = expr;
```

is that explicit conversions are permitted in the ETII version, but not in the auto-free one. However, the ETII version is equivalent to these auto-free versions:

```
type var(expr);
type var{expr};
```

In effect, the ETII performs direct initialization, not copy initialization, and as such is arguably less safe (due to support for more conversions) than an auto-free variable declaration using

copy initialization syntax.

11/23/14 bbs PDF 46

A way to confirm that an expression (such as a function call) returns the type you expect is to use static_assert. On page 45, for example, the type of the object sum could be asserted to be Matrix:

```
static_assert(std::is_same<decltype(sum), Matrix>::value,
              "sum isn't a Matrix!");
```

Typically, you want to strip reference and cv-qualifiers before doing the comparison, so you can use std::decay (as on pages 190-191), but, as I note on page 191-192, std::decay will also turn arrays and functions into pointers, and you may not want that. In that case, you'll want to use something more like R. Martinho Fernandes' Unqualified type trait.

4/23/15 nwp PDF 59

The fact that nullptr isn't a pointer type means that it can't be passed to templates partially specialized for pointers:

```
template<typename T>
void f(T*);            // partial specialization for pointers

f(nullptr);            // error!
```

10/24/14 dxn PDF Item 9

Just as C++14 standardizes the convention of using a "_t" suffix for nested types, C++17 is likely to standardize the convention of using a "_v" suffix for values (i.e., a nested "value" member).

1/25/15 tyk All Item 9

Another advantage of alias templates over typedefs nested inside templatized structs is that type deduction is applicable to parameters of alias template types, but not to parameters of typedef-inside-a-struct types:

```
template<typename T>                              // alias template from
using MyAllocList = std::list<T, MyAlloc<T>>;     // Item 9 (PDF page 64)

template<typename T>                              // f1 is a function
void f1(MyAllocList<T>& list);                    // template

MyAllocList<Widget> lw;                           // also from Item 9

f1(lw);                                           // fine, T in f1
                                                  // deduced as Widget


template<typename T>                              // struct with nested typedef
struct MyAllocList {                              // from Item 9 (PDF page 64)
  typedef std::list<T, MyAlloc<T>> type;
};

template<typename T>                              // f2 is a function
void f2(typename MyAllocList<T>::type& list);     // template

MyAllocList<Widget>::type lw;                     // also from Item 9

f2(lw);                                           // error! can't
                                                  // deduce T in f2
```

11/23/14 bbs PDF 71-73

In C++14, std::tuples can be indexed by the type of a field instead of its index, as long as the type is unique. That has only limited applicability to the example in the book, because the first and second fields are of the same type (std::string), but it's still worth knowing about.

6/19/16 axd All 71-73

Another approach to creating enums to identify std::tuple fields is to nest an anonymous enum inside a struct, e.g.:

```
struct UserInfoFields
{
    enum { name, email, reputation };
};

auto val = std::get<UserInfoFields::name>(uInfo);
```

2/12/15 hxh All Item 11

Move operations should probably never be deleted. If you want neither moving nor copying to be available, deleting the copy operations suffices. (That will prevent generation of the move operations.) If you want only copying to be supported, declaring the copy operations (possibly via "=default") will again prevent the move operations from being generated.

If you try to permit copying but disable moving by declaring the

copy operations but deleting the move operations, you will prevent the copying of rvalues, because the deleted move operations will be a better match during overload resolution than the "fallback" copy operations.

3/29/15 dxg All Item 11

dxg writes: "I had a class with default constructor declared as private without implementation. I decided to do a simple refactor and made the function public and deleted. After that clang found an unused field in the class! While the function had a declaration compiler had no any information about its body, so it can't prove that the field has been unsued (the constructor could be defined anywhere with the usage of the field)."

1/25/15 tyk All 83-85

One application of reference-qualified member functions is to prevent modification of rvalues (e.g., objects returned from functions). Such modification is not permitted for rvalues of built-in types, so this would be a way for user-defined types to mimic that behavior.

The approach is to restrict use of modifying member functions to lvalues by "&"-qualifying them, e.g.:

```
class Widget {
public:
  Widget& operator=(const Widget&) &;   // copy-assign only lvalues
  Widget& operator=(Widget&&) &;        // move-assign only lvalues

  void setName(const std::string&) &;   // setter is only for lvalues
};
```

12/18/14 ahp PDF 88-89
1/26/15 tyk

The Standard requires that std::cbegin be implemented by calling std::begin (as shown in the book), but in C++14, the code could be made a smidge more generic by using the following implementation, which, unlike the code in the book, would work with container-like objects supporting non-member begin instead of member begin:

```
template <class C>
decltype(auto) cbegin(const C& container)
{
  using std::begin;
  return begin(container);
}
```

Achieving the same thing in C++11 is trickier, because we have to make std::begin visible during evaluation of the return type declaration without making it visible outside the cbegin function we're declaring. The following works:

```
namespace cbegin_access_impl {         // helper namespace
  using std::begin;

  template<class C>
  auto cbegin(const C& container)->decltype(begin(container))
  {
    return begin(container);
  }
}

template<class C>
auto cbegin(const C& container)->decltype(cbegin_access_impl::begin(container))
{
  return cbegin_access_impl::begin(container);
}
```

1/24/15 sdm PDF 92

In the third sentence of the fourth paragraph, the word "sporting" is correct; it's not a typo for "supporting." To "sport" something can mean to wear it as an adornment, so in the sentence in the book, I'm talking about functions that offer (i.e., that "sport") the strong exception safety guarantee.

12/19/14 ahp All Item 16

For cached values that are constant, use of a std::once_flag and std::call_once is probably preferable to use of a mutex or a std::atomic. However, if a cached value can change over time (e.g., in the examples in Item 16, if the coefficients of a Polynomial can be changed at runtime (thus changing its roots) or if the value returned by Widget::magicValue changes each day at midnight), the approaches shown in the Item are likely to be more appropriate.

11/17/15 jxl All 105-6
2/ 9/17 abt

Though introduction of the std::mutex causes Polynomial and Point to become uncopyable and unmovable, copyability and movability can be restored by defining the copy and move operations manually. abt writes that one way to approach the matter would be to "create a new mutex for the object being created and then copy everything else. Likewise, custom copy/move assignment operators would be required and would copy/move all items except for the mutex."

11/17/15 jxl PDF 106-7

If magicValue is called often enough and the cost of mutex acquisition is high enough compared to the cost of using std::atomics, it could be an overall performance improvement to use std::atomics for cacheValid and cachedValue, even though cachedValue might be computed more than once. That's because the cost savings on each access could more than compensate for the excess costs associated with multiple magicValue computations.

11/17/15 jxl PDF 108

If more than one variable or memory location requires synchronization, but all can be packed into the amount of memory that can be manipulated truly atomically by the underlying hardware (typically one or two words), an alternative to use of a std::mutex is a std::atomic<PackedType>, where PackedType is a user-defined type containing the variables to be manipulated as a unit.

1/11/16 sdm All Item 17

Compiler-generated special functions are implicitly noexcept unless their compiler-generated implementations call non-noexcept functions. Destructors and move operations rarely call functions that can throw, so implicitly-generated destructors and move operations are typically noexcept. (As noted in Item 14, even user-declared destructors are usually implicitly noexcept.)

1/18/15 bbs PDF 117

Another disadvantage of raw pointers is that deleting incomplete types through them, while often eliciting a warning, nevertheless compiles. Trying to do the same thing through a smart pointer is an error, and the code typically fails to compile.

8/31/16 ryk PDF 117

Two more disadvantages of raw pointers:

- If they're not explicitly initialized, their initial value may not be defined. std::unique_ptr and std::shared_ptr are initialized to null by default.
- As data members, they typically require that the containing class implement the special member functions. With smart pointer data members, this is generally unnecessary.

12/13/14 rxm All Item 18

The makeInvestment interface is unrealistic, because it implies that all derived object types can be created from the same types of arguments. This is especially apparent in the sample implementation code, where are arguments are perfect-forwarded to all derived class constructors.

12/15/14 sdm PDF 122

The encapsulation offered by the "auto"-returning version of makeInvestment is diluted by the fact that in order to call it, its body must be visible. In practice, this means that functions with deduced return types must often be defined in header files.

On the plus side, the "auto"-returning version of makeInvestment prevents the name "delInvmt" from being visible outside the function.

1/28/15 tyk PDF 122-123

A drawback of giving makeInvestment an "auto" return type is that it complicates things for callers who need to know that type (e.g., to declare a container holding objects returned from makeInvestment). tyk argues that there's no point in trying to hide a factory function's return type.

1/18/15 bbs PDF 124

A situation where you might choose to use std::unique_ptr<T[]> instead of a standard container is when the size of the array isn't known during compilation (thus ruling out std::array) and where you want to minimize memory usage. The footprint of std::unique_ptr<T[]> is slightly smaller than that of a std::vector, std::deque, or std::string.

12/20/14 tyk All Item 21
6/23/15 whb

Make functions can't be used when the constructor to be called isn't public. (Declaring the make functions friends may not work, because make functions may internally call other functions that actually perform the constructor invocation.) An example of this is in Item 19 (on page 132 of the PDF version of the book), where Widget declares a static factory function instead of using a make function.

For workarounds, see this StackOverflow discussion.

3/23/15 etc All Item 21

Make functions can't be used to create smart pointers to abstract base classes, e.g., that are initialized by cloning functions. For example:

```
class Base {
public:
  virtual Base* clone() const = 0;
};

class Derived: public Base {
public:
  virtual Derived* clone() const override;
```

```
  };

  Derived d;

  auto pb1 = std::make_shared<Base>(d.clone());  // error!
  auto pb2 = std::shared_ptr<Base>(d.clone());   // fine
```

5/ 8/18 hxp All 140-141 In C++17, argument evaluation order rules have been revised such that no resource leak can occur.

1/29/15 tyk PDF 152 In second para, I say that the compiler-generated move operations do "exactly what's desired: perform a move on the underlying std::unique_ptr". Such moves are shallow, but for a class with value semantics like Widget, deep move operations would be at least as reasonable, and they'd avoid the need to handle null pImpl pointers in Widget member functions. However, they'd also require allocating and initializing a Widget::Impl object for moved-from objects, even if the next operation to be performed on the Widget was destruction.

1/19/15 bbs All Item 22 The Pimpl idiom reduces build times, but the indirection introduced by the pPimpl pointer may increase runtime, and the pointer itself increases total memory usage.

3/10/16 mtl PDF 154 The shown Widget::operator= works correctly if a Widget is assigned to itself, but such assignment would incur the cost of copying an Impl object, and that could be expensive. If that cost is sufficiently high, or if self-assignment is expected to be common, it could be worthwhile to add the customary self-assignment check to the top of operator=:

```
  Widget& Widget::operator=(const Widget& rhs)
  {
    if (this == &rhs) return *this;

    ...                                 // implementation as in book
  }
```

However, this kind of check has costs of its own. The function is a little longer, and a new branch is added.

2/11/15 sdm PDF 171 In last paragraph, I say that the number of overloads grows *geometrically* instead of *exponentially*, because, per the Wikipedia page, "geometric" is appropriate for discrete domains, while "exponential" applies to continuous domains.

12/19/14 ahp All Item 26 The fact that template functions may outcompete non-template functions with the same name (i.e., overloads) is not unique to templates taking universal references or even to C++11. The same thing can happen in C++98 with a template taking a const T& parameter, because such a parameter can bind to all types and can accept both lvalues and rvalues. In my (Scott's) experience, however, this rarely leads to trouble in practice, probably because there are no move semantics in C++98.

3/ 1/18 ext PDF 187-8 An alternative to having logAndAddImpl at the top of page 188 call logAndAdd is to have it call directly to the version of logAndAddImpl taking a non-integral type, i.e.,

```
  void logAndAddImpl(int idx, std::true_type)
  {
    logAndAddImpl(nameFromIdx(idx), std::false_type());
  }
```

This approach has the advantages that (1) it avoids the call to logAndAdd that merely forwards to logAndAddImpl and (2) it may simplify the declaration order among the logAndAdd template and the logAndAddImpl template specializations.

1/21/15 sdm All 195-6 Several people have suggested moving std::is_constructible from a static_assert inside the Person constructor to the std::enable_if/std::enable_if_t outside it. This would change the behavior of the code. The enable_if controls whether the constructor should be considered during overload resolution. The static_assert controls whether the function, once it's been chosen by overload resolution, has received a valid argument type. If std::is_constructible were part of the enable_if clause and a caller passed an argument of an invalid type, the error message would say something like "No function found for an argument of type [*type of argument*]." With std::is_constructible inside the static_assert, the error message would say "Parameter n can't be used to construct a std::string."

2/ 3/16 shc PDF 196 To get the message specified in the static_assert to appear before the errors resulting from invalid template instantiation, the

static_assert can be moved into a lambda whose invocation precedes the normal initializer in a comma expression in the member initializer list:

```
class Person {
public:
  template<                                  // as before
    typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
      !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n)
  : name((
          []{ static_assert(std::is_constructible<std::string, T>::value,
                            "Parameter n can't be used to construct a std::string"); }(),
          std::forward<T>(n)
         ))
  {
    …                    // the usual ctor work goes here
  }

  …                      // remainder of Person class (as before)

};
```

The code is a bit tricky, because you have to surround the comma expression with parentheses in order to avoid an incorrect parse (as explained here).

Note that it's not possible to use the static_assert in the comma expression directly (i.e., to omit the lambda), because only expressions can be used in a member initializer, and static_assert is a declaration, not an expression.

12/31/14 axk All Item 28
2/ 1/15 tyk

If a template taking a parameter of type cv T& (where cv is const or volatile or both) is explicitly instantiated with a reference type, cv refers to the reference in the explicitly-specified reference type and is ignored. Reference-collapsing then applies as usual:

```
template<typename T>
void func(const T& param);

int x;
func<int&>(x)  // inside func, param's type is
               // int&, not const int&
```

For details, consult this StackOverflow question.

The phenomenon of cv qualifiers disappearing when references to references arise is not limited to function template parameters. In general, if references to references arise during compilation (e.g., in parameter type construction, typedef construction, or type creation via decltyp expressions), any cv-qualifiers that apply to the referred-to type are discarded before reference collapsing occurs. Here's a modified example from the end of Item 28 (on page 202 of the printed and PDF versions of the book):

```
template<typename T>
class Widget {
public:
  typedef const T&& RvalueRefToConstT;  // compared to example
  …                                     // in book, const has
};                                      // been added

Widget<int&> w;               // w::RvalueRefToConstT is int&
```

3/12/19 gpk All 209

Another workaround for the inability to perfect forward braced initializers is to overload the forwarding function such that a new overload takes a std::initializer_list:

```
template<typename T>                        // as in book
void fwd(T&& param) { … }

template<typename T>                        // new overload
void fwd(std::initializer_list<T> ilist)
{
  f(ilist);
}

fwd( {1,2,3} );                             // fine, calls new overload
```

gpk notes that this approach seems to violate Item 26 ("Avoid overloading on universal references"), but in this case, the overly greedy behavior of universal reference parameters isn't a problem. In fact, when it comes to braced initializer arguments, the problem in some sense that they're not greedy enough.

12/19/14 ahp All Item 30

A workaround for the inability to perfect-forward const/constexpr data members and bitfields is to use the member to create a temporary expression with the same value. For example, instead of "foo(my_static_const)", pass "foo(+my_static_const)", or instead of "foo(my->bit_field)", pass "foo(my->bit_field+0)". This must be applied at each call site and may change the type of the expression (chars and shorts will typically be promoted to ints, for example), but it may be acceptable in some contexts.

If a static member is in a library that is header-only, this approach keeps it header-only, and if the class is part of a library not under the programmer's control, this solution avoids providing a definition for a symbol that the user didn't declare.

8/10/15 nwp PDF 216

Because each lambda gives rise to a unique closure class, two textually identical lambdas create different types. This can lead to code duplication if the closure type's operator() isn't inlined. The problem can be avoided by storing the lambda's closure in an auto variable and using that variable in all locations where the lambda would otherwise be employed:

```
f1([x, y, z]{ /* complex code */ });         // each call creates a
f2([x, y, z]{ /* same code as above */ });   // new closure class with
                                             // a new operator()
                                             // function

auto func = [x, y, z]{ /* same code as above*/ };

f1(func);                                    // both calls use the
f2(func);                                    // same closure class's
                                             // operator() function
```

12/19/14 ahp PDF 223

Just as "[=]" could mislead readers into thinking that a lambda was self-contained, "[]" could have the same effect; a reader might think that "no capture" implies "self-contained." The behavior of both "[=]" and "[]" lambdas can be dependent on objects with static storage duration.

6/15/14 tyk All Item 34

Lambdas can be much more verbose than std::bind. This is especially true for a bound "pass through" function, i.e., a bound function whose arguments are perfect forwarded and whose return value is decltype(auto)'d to the caller (i.e., where "perfect returning" is used). For example, given

```
class Widget {
public:
  const std::vector<std::string>& update(std::string newVal);
  ...
};
```

and a Widget w we know we want to invoke update on, compare the following equivalent ways to express the binding of w and update:

```
// assume "using namespace std::placeholders;" has already been done
auto boundWithBind = std::bind(&Widget::update, w, _1);

auto boundWithLambda =
 [w](auto&& newVal)->decltype(auto)
 { return w.update(std::forward<decltype(newVal)>(newVal)); };
```

2/ 8/15 tyk All Item 35

"Task-based programming" is a design approach based on (1) defining units of work ("tasks") to be executed, (2) specifying the data and timing dependencies among the tasks, and (3) relying on the runtime system to execute the tasks such that the dependencies are respected. std::async is the best tool in the C++14 Standard Library for such programming, but it doesn't fully enable task-based programming, as this blog post explains.

6/19/16 axd All 244

The book notes that task-based programming significantly reduces the likelihood of an out-of-threads exception, but that doesn't change the validity of page 243's statement that well-written software must deal with such exceptions. Task-based programming therefore has no fundamental advantage over thread-based programming as regards the need for software to address out-of-thread exceptions.

12/18/17 pxj All 244 246

Rigorously speaking, the std::launch::async launch policy doesn't guarantee that std::async will run the passed-in function on a different thread, it merely guarantees that program execution will

proceed *as if* the passed-in function ran on a different thread. This flexibility might permit implementations to, for example, run the passed-in function on an existing-but-not-currently-used thread in a thread pool, or to optimize out execution of the passed-in function entirely (e.g., if they could establish that eliminating execution of the function would have no effect on the observable behavior of the program).

2/ 8/15 tyk PDF 251-256

The program will be terminated if an exception arises in the lambda expression inside doWork, i.e., if the filtering function or std::vector::push_back throws. This can be fixed by wrapping the lambda in a std::packaged_task, but if control in doWork never reaches *performComputation* (e.g., due to *conditionsAreSatisfied* returning false or to an exception arising while t's priority is being set), the use of ThreadRAII::DtorAction::join on page 256 would cause doWork to block waiting for the lambda to finish, even though its result (the data in goodVals) would never be used.

Having the lambda move-capture filter and return goodVals would make it possible to change the ThreadRAII DtorAction to ThreadRAII::detach, thus eliminating the need to block. Rewriting doWork to incorporate these ideas yields:

```
bool doWork(std::function<bool(int)> filter,
            int maxVal = tenMillion)
{
  std::packaged_task<std::vector<int>()>
     filterTask([filter=std::move(filter), maxVal]
                {
                  std::vector<int> goodVals;
                  for (auto i = 0; i <= maxVal; ++i)
                    { if (filter(i)) goodVals.push_back(i); }
                  return goodVals;
                });

  auto goodValsFuture = filterTask.get_future();

  ThreadRAII t(std::thread(std::move(filterTask)),
               ThreadRAII::DtorAction::detach);

  auto tnh = t.get().native_handle();
  …

  if (conditionsAreSatisfied()) {
    auto goodVals = goodValsFuture.get();
    performComputation(goodVals);
    return true;
  }

  return false;
}
```

A noteworthy aspect of this implementation is that if an exception occurs in the lambda, it will be propagated to doWork only if *conditionsAreSatisfied* returns true. If *conditionsAreSatisfied* returns false, the exception will be ignored.

10/27/15 kzh PDF 254

std::thread::join and std::thread::detach can emit exceptions, and that means that ThreadRAII's destructor can throw. If this happens during stack unwinding due to another exception, program execution either terminates or yields undefined behavior.

My advice on how to approach this problem is described in Item 8 of *Effective C++, Third Edition*: add a function to ThreadRAII that clients can call if they need to be able to handle exceptions that may arise when the destructor action is performed.

1/19/15 bbs All Item 38

The fact that the final future referring to a shared state created via a call to std::async and where the launch policy is std::launch::async blocks until the asynchronously running task completes means that "fire and forget" std::async calls that ignore std::async's return value may block until the asynchronously running task finishes. That is, such calls may run synchronously. That's because the std::future returned from std::async will be destroyed at the end of the statement calling std::async, so if the launch policy is std::launch::async, that std::future's destructor will block until the invoked task finishes running. (The Standardization Committee considered changing this behavior, but they ultimately decided to keep it in the interest of backwards compatibility.)

2/ 7/15 tyk All Item 40

A secondary use for volatile is to prevent compilers from optimizing away code that has no effect on observable program behavior. If you try to time an empty loop to calcuate the loop overhead (e.g., try to time "for (int i = 0; i < 10'000'000; ++i);"), compilers will typically optimize the loop away and report that no time was

spent there. To prevent the optimization, the loop variable can be declared volatile: "for (**volatile** int i = 0; i < 10'000'000; ++i);".

1/30/15 tyk All Item 41 Another behavioral difference between (1) overloading for lvalues and rvalues and (2) a template taking a universal reference (uref) is that the lvalue overload declares its parameter const, while the uref approach doesn't. This means that functions invoked on the lvalue overload's parameter will always be the const versions, while functions invoked on the uref version's parameter will be the const versions only if the argument passed in is const. In other words, non-const lvalue arguments may yield different behavior in the overloading design vis-a-vis the uref design.

1/30/15 tyk All Item 41 A universal reference parameter can bind to virtually any kind of argument. (Exceptions are covered in Item 30.) An implication is that metaprogramming techniques based on testing the validity of function calls with certain kinds of arguments are rendered largely useless, because virtually all calls to functions taking universal references are considered valid. That's not the case with functions taking parameters of specific types, because only argument types compatible with the parameter types yield valid calls. This difference can be a reason to overload for lvalues and rvalues of specific types instead of writing a template taking universal references.

Who's who:

txn = Tyson Nottingham
dxn = Daniel Nielsen
kxv = Kostas Vlahavas
txk = Takatoshi Kondo
lxs = Lewis Stiller
dxj = Dainis Jonitis
tyk = Tomasz Kamiński
kxa = Ken Aarts
rxb = Ryan Brown
daa = Daniel Alonso Alemany
bxr = Barry Revzin
rxm = Robert McCabe
rxk = Rex Kerr
kxh = Kjell Hedstrom
csc = Curtis S. Cooper
bbs = Bartek Szurgot
ahp = Adam Peterson
hxh = Howard Hinnant
sxt = Semen Trygubenko
lxz = Leor Zolman
scp = Siu Ching Pong
etc = Eitan Frachtenberg
pxz = Piotr Zygielo
whb = Will Bickford
kyh = Kisung Han
rxh = Rein Halbersma
jxl = Johannes Laire
shc = 蘇浩禎 Su Hao-Chen
jxm = József Mihalicza
gxk = Gerhard Kreuzer
axd = Alex Dumov
ryk = Robin Kuzmin
hxb = Harri Berglund
djm = Declan Moran
sxa = Stephane Aubry
rxz = Ryan Zischke
ext = Evgeny Tarasov
dxk = Daniel Khoshnoudirad
fxr = Fraser Ross
tjb = Tyler Brock
akb = Agustín K-ballo Bergé
mxs = Minkoo Seo
axk = Andrey Khalyavin
fxf = Fabrice Ferino
jxz = Jay Zipnick
oxb = Oliver Bruns
bvw = Bart Vandewoestyne
bxy = Benjamin Yates
bxm = Bernhard Merkle
pxv = Petr Valasek
mxw = Marcel Wid
mxm = Miroslaw Michalski
mjs = Matthias J. Sax
mam = Mark A. McLaughlin
vhg = Vlad Gheorghiu
mxk = Mitsuru Kariya
gxs = Grebënkin Sergey
fdb = Ferdinand deBoevere
dxg = Denis Gladkiy
lxp = Lucas Panian
nwp = Nathan W. Panike
dgw = Don Goodman-Wilson
mys = Marek Scholle
kzh = Kalle Huttunen
bxc = Ben Craig
nxj = Nicolai Josuttis
jsb = Jesper Storm Bache
mtl = Matthew Limber
gxp = Gennaro Prota
mtb = Mitch T. Besser
txb = Tim Buchowski
cxl = Calum Laing
abt - Allen Taylor
tgm = Tomasz Grzegorz Markiewicz
pxj = Péter Joó
hxp = Helge Penne
gpk = Greg Klein
kst = Keith S. Thompson

From here you may wish to visit the Amazon Page for *Effective Modern C++*.