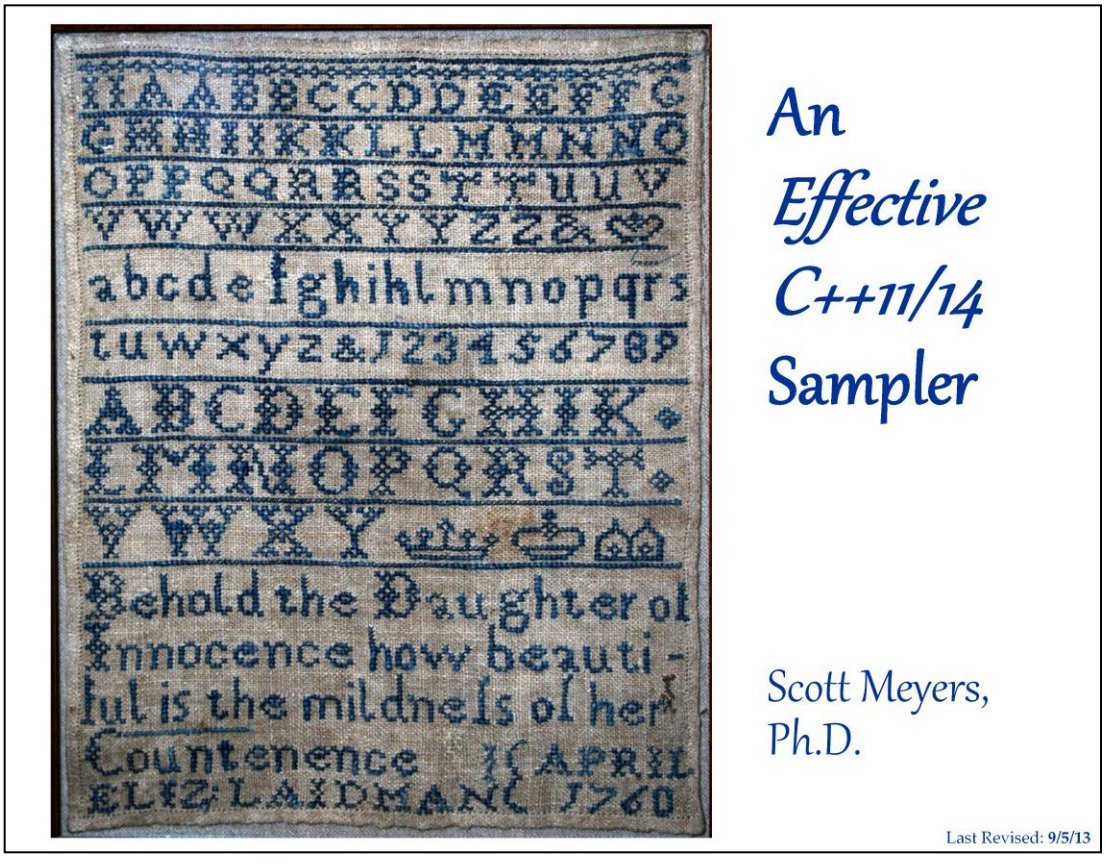
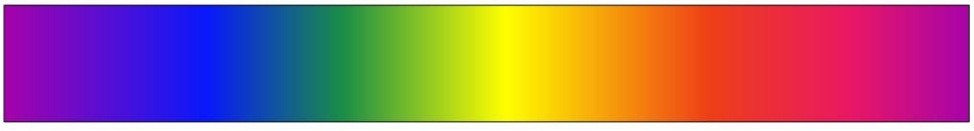


Image: http://upload.wikimedia.org/wikipedia/commons/b/b1/Sampler_by_Elizabeth_Laidman%2C_1760.jpg. Wikipedia caption: "Cross-stitch alphabet sampler worked by Elizabeth Laidman, 1760."

Effective C++11/14

- All new material vis-à-vis my other books.
- Usual guideline-based schtick.
- Target readership:
  - Good knowledge of C++98.
  - Basic knowledge of C++11.
  - No knowledge of C++14.
- Projected completion 2Q 2014.

http://siliconangle.com/blog/2010/04/23/facebook-insights-not-scary-and-not-effective-yet/



Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Understand `std::move` and `std::forward`

What they don't do:

- `std::move` doesn't move.
- `std::forward` doesn't forward.
- Neither generates code.

They're simply casts:

- `std::move` *unconditionally* casts to an rvalue.
- `std::forward` *conditionally* casts to an rvalue.

std::move

Essentially-conforming implementation:

```
template<typename T>                          // in namespace std
typename remove_reference<T>::type&&
move(T&& param)
{
  using ReturnType = typename remove_reference<T>::type&&;

  return static_cast<ReturnType>(param);
}
```

Clearly just a cast.

- Think of it as `rvalue_cast`.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

std::move

Aside: C++14 enables two implementation simplifications:

```
template<typename T>                              // deduce
decltype(auto) move(T&& param)                    // return type
{
  using ReturnType = remove_reference_t<T>&&;     // reduce noise
  return static_cast<ReturnType>(param);
}
```


Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

std::move

Consider:

```
struct SomeDataStructure {
  std::string name;
  …
};
SomeDataStructure sds;


void processAndAdd(const std::string s)  // "Want speed? Pass
{                                        // by value", "Pass
                                         // sink args by value"
  …                                      // read-only ops on s
  sds.name = std::move(s);               // "move" s into sds
}
```

Compiles, runs, probably passes unit tests.

- Move a *const* `std::string`?

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

std::move

Two `std::string` assignment operators:

```
class string {                              // really
public:                                     // basic_string<char…>
  …
  string& operator=(const string& rhs);     // copy assignment
  string& operator=(string&& rhs);          // move assignment
  …
};
```

Ergo:

```
void processAndAdd(const std::string s)
{
  …
  sds.name = std::move(s);                  // copies s!
}
```

Essentially same as why "moves" on legacy types copy (but compile!).

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

`std::move`

Lessons:

- Don't declare objects `const` if you want to move from them.
- Using `std::move` doesn't guarantee anything will be moved.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

std::forward

Consider:

```
void process(Widget& lvalParam);       // process lvalues
void process(Widget&& rvalParam);      // process rvalues

template<typename T>
void logAndProcess(T&& param)
{
  makeLogEntry("Calling 'process'",
               std::chrono::system_clock::now());
  process(std::forward<T>(param));
}

Widget w;

logAndProcess(w);                      // call with lvalue
logAndProcess(std::move(w));           // call with rvalue
```

- `param` is an lvalue.
- `std::forward` casts `param` to rvalue iff rvalue was passed in.
  - ➡ A conditional cast.

std::forward

In `std::forward`, `T` encodes lvalueness/rvalueness.

Conceptual `std::forward` implementation:

```
template<typename T>                    // in namespace std
T&& forward(T&& param)
{
  if (is_lvalue_reference<T>::value) { // if T indicates lvalue
    return param;                       // do nothing
  } else {                              // else
    return move(param);                 // cast to rvalue
  }
}
```

Simply a conditional cast.

Declared return type misleading:

- For lvalues, returns `T&`.

Real implementations more sophisticated, faster, will compile.

Guideline

Understand `std::move` and `std::forward`.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Declare Functions noexcept Whenever Possible

`noexcept` the new `throw()`:

```
void someFunc() noexcept;    // C++11
void someFunc() throw();     // C++98 and C++11 (deprecated)
```

Semantics similar, but not identical.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Violated Exception Specifications

When exception thrown:

```
void someFunc() throw() { /* code that throws */ }
```

1. Unwind stack to `someFunc`'s caller.
2. Call `std::unexpected` (or its replacement).
   - Typically calls `std::terminate` (or its replacement).

```
void someFunc() noexcept { /* code that throws */ }
```

- *Maybe* unwind stack.

1. Call `std::terminate` (or its replacement).

Scott Meyers, Software Development Consultant
http://aristeia.com/

Slide 14

For a violated `noexcept`, whether (and, if so, how far) the stack is unwound is implementation-defined.

Stack Unwinding and Optimization

Not unwinding stack permits more optimizations:

```
void someFunc() noexcept;  // more optimization opportunities
void someFunc() throw();   // fewer optimization opportunities
```

Stack Unwinding and Optimization

Analysis not limited to `noexcept` vs. `throw()`:

```
void someFunc() noexcept;  // more optimization options
void someFunc() throw();   // fewer optimization options
void someFunc();           // fewer optimization options
```

When `noexcept` applies, use it.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Move Functions

Especially important for move functions:

- Move constructor, move assignment operator

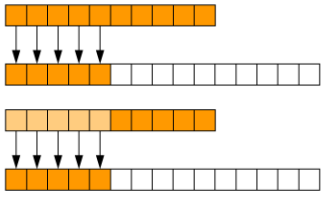
Some code may replace copying *only with non-throwing moves*.

- Functions with strong exception-safety guarantee.
  - E.g., many `std::vector` functions:
    - `reserve`
    - `resize`
    - `push_back`
    - …

Exception Safety of `std::vector::push_back`

If exception thrown migrating elements from old to new storage:

- Copying: original storage unchanged.
  - ➡ Strong guarantee.
- Moving: original storage modified.
  - ➡ *Basic* guarantee.



For strong guarantee, only non-throwing move acceptable.

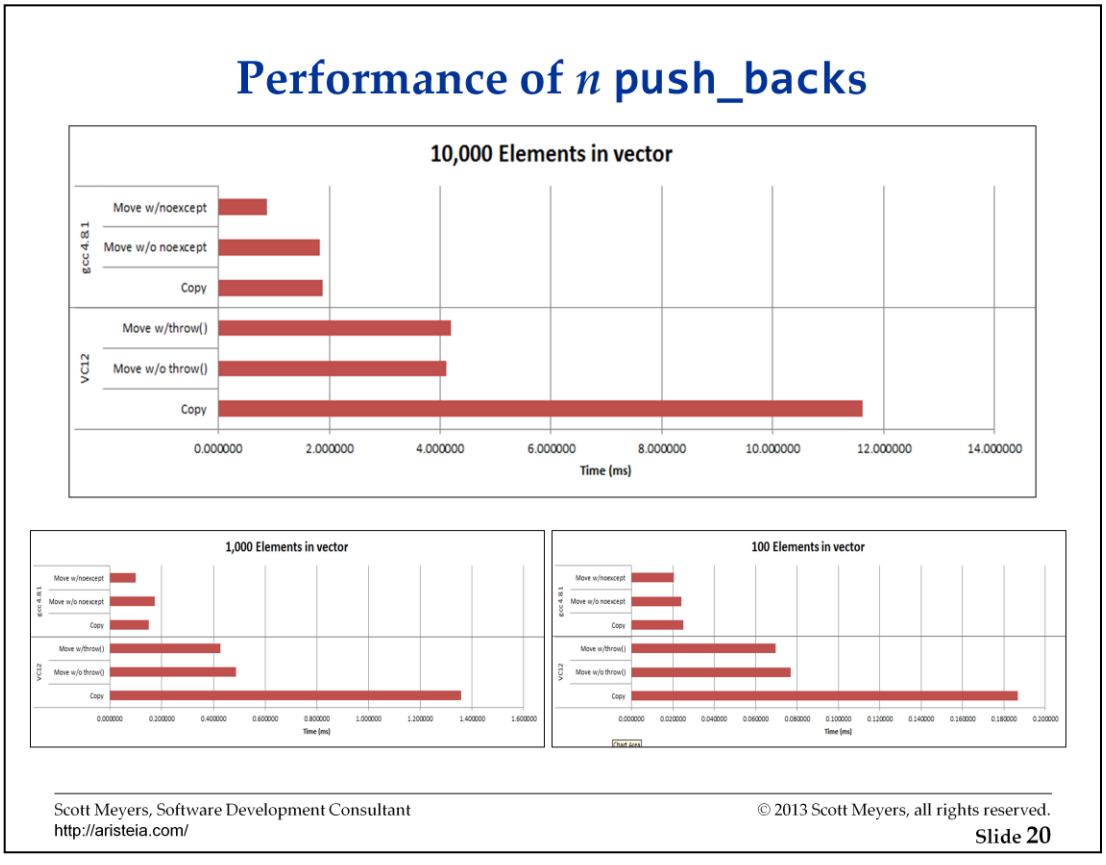
- `std::vector::push_back` uses **`std::move_if_noexcept`**.
  - ➡ "Cast to rvalue only if T's move known to not throw."

Copy vs. Move vs. Nonthrowing Move

Consider:

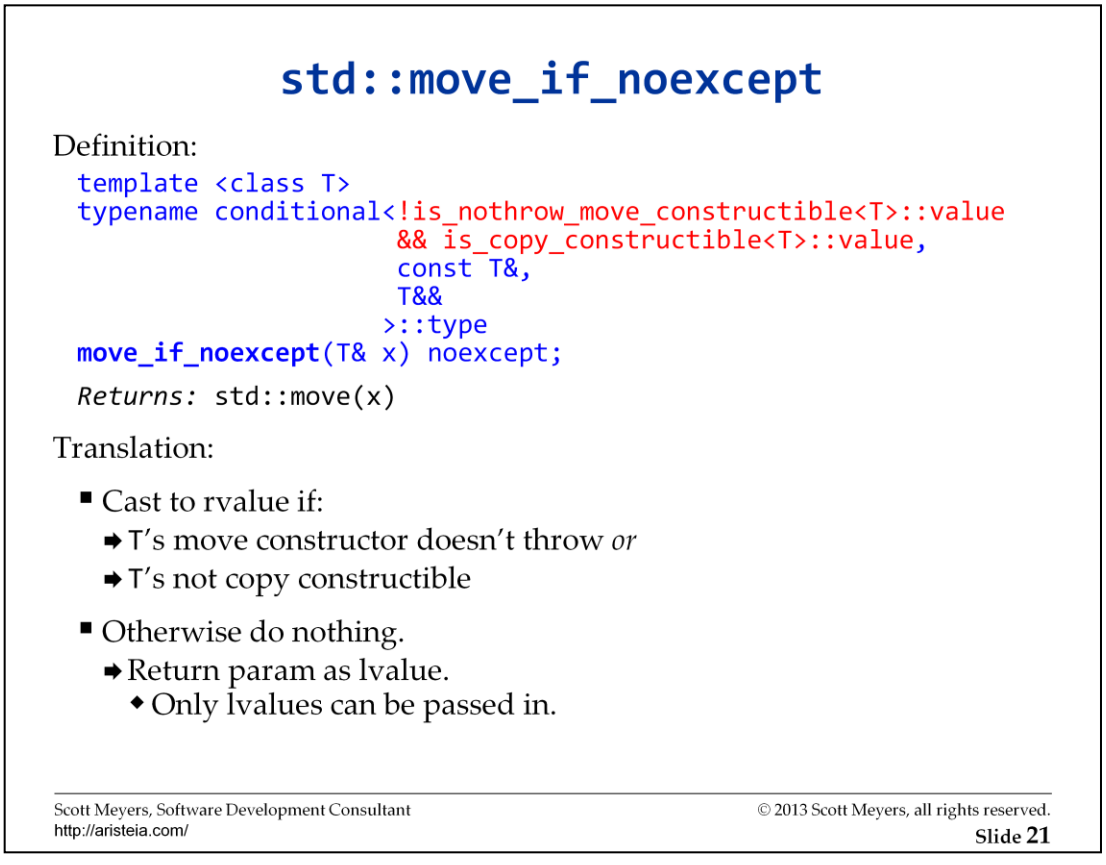
```
const std::string stringValue("This string has 29 characters");

class Widget {
private:
  std::string s;

public:
  Widget(): s(stringValue) {}
  …                                    // copy & move functions
};

std::vector<Widget> vw;
Widget w;

for (std::size_t i = 0; i < n; ++i) {  // append n copies
  vw.push_back(w);                     // of w to vw
}
```


Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.



All data gathered August 2013 on a Lenovo W510 laptop.

Likely reason that performance numbers for gcc are much better than for MSVC is that gcc uses reference-counting, which is not conforming under C++11.



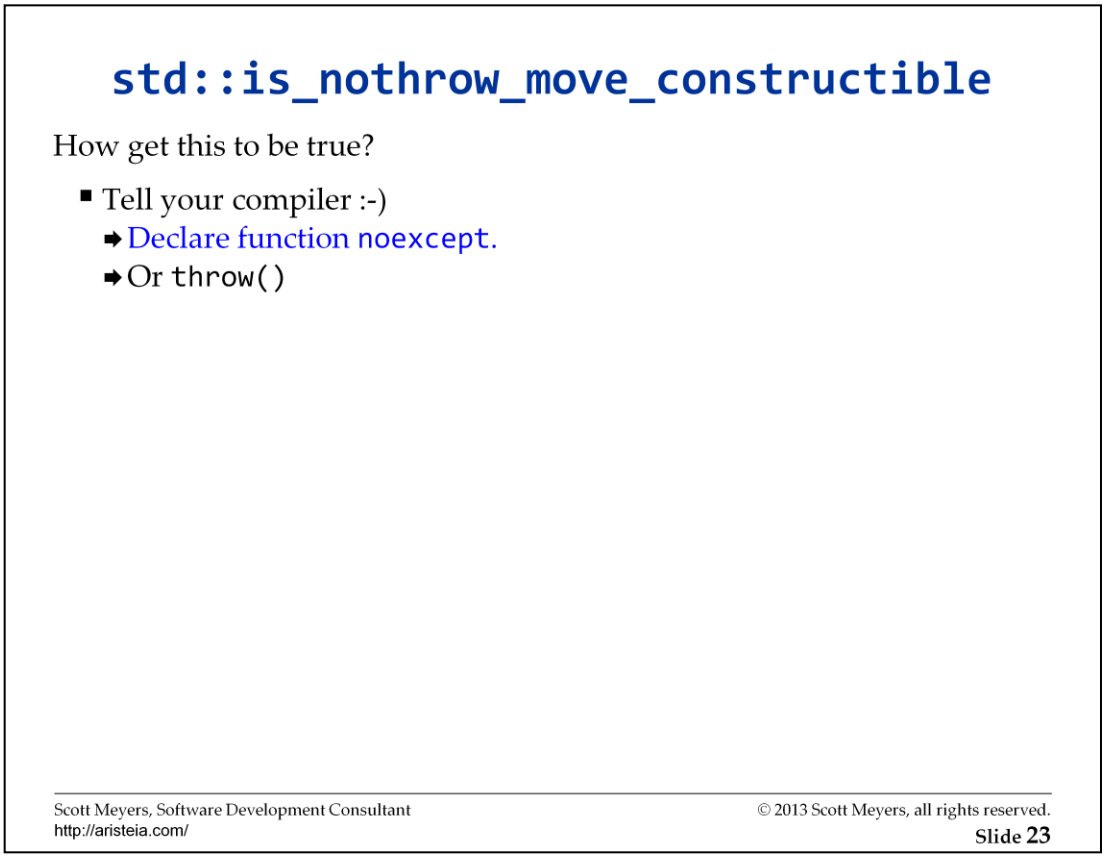
Definition is from §20.2.3 of the C++11 Standard.

`std::move_if_noexcept`

For move-only types (e.g., `std::unique_ptr`, `std::thread`):

- Cast *unconditionally* performed if copy construction not available.

`std::move_if_noexcept` ≢ "Move only if T's move known to not throw." :-)



VC12 (part of VS2013) generates correct values for `std::is_nothrow_move_constructibl<T>` if T's move constructor has a "`throw()`" specification.

Declare Functions `noexcept` *Whenever Possible*

`noexcept` part of a function's *interface*.

- Clients may depend on it, e.g., for exception-safety guarantees.
- Changing it may break client code.
- Don't use `noexcept` just because current implementation permits it.
  ➧ Implementation may change.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Declare Functions noexcept *Whenever Possible*

Most code is exception-neutral.

- Exceptions may freely flow through them.
- Such code eschews exception specifications.

```
void commonFunction( parameters );      // no exception spec
```

Move Functions Revisited

Not all move functions are `noexcept`.

- E.g., none of the standard containers.

```
template <class T, class Allocator = allocator<T> >  // from
class list {                                         // C++11
public:
  list(list&& x);                                    // no
  list<T,Allocator>&                                 // ex
    operator=(list<T,Allocator>&& x);                // specs.
  …
};
```

➡ In general, moving containers may require memory allocation.

Scott Meyers, Software Development Consultant
http://aristeia.com/

Slide 26

The move functions in `std::string` are declared `noexcept`, but (1) technically, `std::string` is not a container, and (2) Howard Hinnant has opined that this may be a defect in the standard.

Moving containers may require memory allocation, because empty containers may contain dynamically allocated memory. For example, `std::list` implementations may have a sentinel node present, even if there are no elements in the list. When move-constructing a `std::list`, a new sentinel node must be allocated, either for the new object or to make up for the sentinel node that was moved from the old object. Other containers may desire dynamically allocated memory for other reasons.

Move Functions Revisited

When `noexcept` correct for a class's move functions,

- Declare them `noexcept`.
- If copy functions not `noexcept`, encourage clients to review calls.
  - ➡ Some could-throw copies may become can't-throw moves.
    - ◆ Exception-safety guarantees maybe strengthenable.

Conditionally noexcept Functions

`noexcept` actually takes a compile-time boolean expression:

```
void f() noexcept;          // shorthand for below
void f() noexcept(true);    // same meaning as above

void f();                   // shorthand for below
void f() noexcept(false);   // same meaning as above
```

- `noexcept(false)` ≡ function not *guaranteed* not to throw.

Conditionally noexcept Functions

`noexcept` also an *operator*.

- Returns whether an expression guaranteed not to throw.
- Evaluated during compilation.
- Enables *conditionally* `noexcept` functions:

```
template<typename T>
void f(T&& param) noexcept(noexcept(*param));  // f noexcept
                                               // iff *param is
```

Conditionally noexcept Functions

Common for `swap` in standard library. Examples:

```
template <class T, size_t N>
void swap(T (&a)[N], T (&b)[N]) noexcept(noexcept(swap(*a, *b)));

template <class T1, class T2>
struct pair {
  …
  void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                              noexcept(swap(second, p.second)));
  …
};

template <class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y)
  noexcept(noexcept(x.swap(y)));
```

Scott Meyers, Software Development Consultant
http://aristeia.com/

Slide 30

All code on this slide copied out of the C++11 spec.

swap and noexcept

- Whether `swap` `noexcept` often depends on other `swap`s.
- Whether `swap` callers `noexcept` dependent on whether `swap` is.
- Optimization of `swap` and callers dependent on `noexcept` status.
- `swap` frequently used (e.g., in algorithms, copy assignment, etc.).

Implication:

- Declare `swap` `noexcept` whenever possible.
  - ➧ Noteworthy case of the general rule.

Guideline

Declare functions `noexcept` whenever possible.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Make `std::threads` Unjoinable on All Paths

A `std::thread` object may be *joinable*:

- Represent an underlying (i.e., OS) thread of execution (TOE).

Unjoinable `std::threads` represent no underlying TOE.

- Default-constructed `std::threads`.
- `std::threads` that have been `detached` or moved from.
- `std::threads` whose TOE has been `joined`.

Scott Meyers, Software Development Consultant
http://aristeia.com/

Slide 33

TOE is my abbreviation. It's not present in the standard.

The standard uses the term *joinable*, but not *unjoinable*.

Destroying Joinable Threads ⇒ Termination

`std::thread` dtor calls `std::terminate` if it's joinable:

```
{
  std::thread t(funcToRun);     // t is joinable
  …                             // assume t remains joinable
}                               // std::terminate called
```

- Different from Boost.Threads before Boost 1.50.
  - ➡ There, `t.detach` is called.

Implication of C++11/14 Behavior

Local `std::threads` must become unjoinable on all paths from block:

- Become unjoinable:
  - Be `joined` or
  - Be `detached` or
  - Be otherwise made unjoinable (e.g., moved from)
- All paths:
  - `continue`, `break`, `goto`, `return`
  - Flow off end
  - Exception

Unjoinable on *All Paths*

All paths ⇒ RAII classes.

- None for `std::thread` in standard library!
  - ➧ From the Standard:

    Either implicitly detaching or joining a `joinable()` thread in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is raised.
- Easy to write your own.

Scott Meyers, Software Development Consultant
http://aristeia.com/

Slide 38

Quote is from § 30.3.1.3.

RAII Class for `std::thread`

```
class ThreadRAII {
public:
  typedef void (std::thread::*RAIIAction)();     // dtor action

  ThreadRAII(std::thread&& thread, RAIIAction a)
  : t(std::move(thread)), action(a) {}

  ~ThreadRAII()
  { if (t.joinable()) (t.*action)(); }

  std::thread& get() { return t; }

private:
  RAIIAction action;
  std::thread t;
};

ThreadRAII t1(std::thread(doThisWork),           // join on
              &std::thread::join);               // destruction

ThreadRAII t2(std::thread(doThatWork),           // detach on
              &std::thread::detach);             // destruction
```

Scott Meyers, Software Development Consultant
http://aristeia.com/

Slide 39

The `joinable()` test in the destructor is necessary, because both `join` and `detach` have `joinable()` as a precondition. Client code could set up a `ThreadRAII` object to do a `detach` on destruction, then use `get` to perform a premature `join`. In that case, the test in the destructor for `joinable()` is needed to avoid having an exception thrown.

No race can arise due to testing `t.joinable` and then later calling `t.*action`, because `std::thread` objects can't asynchronously become unjoinable; only calls to `std::thread` member functions can change an object's joinable status. If another thread is executing a `std::thread` member function while this thread is executing the destructor, that's a race that leads to UB.

Alternative Approach

Consider `std::async` instead of explicit `std::thread`s.

- No exception-on-destruction issue.
- *Different* quirky behavior:
  - ➡ Dtor of *last future referring to* `std::async`*-derived shared state* blocks until asynchronously running thread completes.
    - ◆ May be modified in C++14…

Guideline

Make `std::threads` unjoinable on all paths.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Summary

- Understand `std::move` and `std::forward`.
- Declare functions `noexcept` whenever possible.
- Make `std::threads` unjoinable on all paths.

Further Information

C++11:

- *Programming Languages — C++*, Document 14882:2011, ISO/IEC, 2011-09-01.
  - ➡ ISO product page: http://tinyurl.com/5soe87 (CHF238 ≡ ~$245).
  - ➡ ANSI product page: http://tinyurl.com/8m6vb5m ($30).
  - ➡ First post-Standard draft (N3337): http://tinyurl.com/6wkboer (free).
    - ◆ Essentially identical to the standard.
- *C++11*, Wikipedia.
- *Overview of the New C++ (C++11/14)*, Scott Meyers, `http://www.artima.com/shop/overview_of_the_new_cpp`.
  - ➡ Annotated training materials (analogous to these).
- *The C++ Standard Library, Second Edition*, Nicolai M. Josuttis, Addison-Wesley, 2012.

Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Further Information

C++14:

- *Programming Languages — C++*, Document N3690, ISO/IEC, 2013-05-15.
  - ➧ Committee Draft (CD).
- "The view from C++ Standard meeting April 2013," Michael Wong, *C/C++ Cafe* (blog) :
  - ➧ Part 1, 25 April 2013. Discusses core language.
  - ➧ Part 2, 26 April 2013. Discusses standard library.
  - ➧ Part 3, 29 April 2013. Discusses concurrency and TSes.
- *Overview of the New C++ (C++11/14)*, Scott Meyers, `http://www.artima.com/shop/overview_of_the_new_cpp`.
  - ➧ Annotated training materials (analogous to these).

Further Information

`std::move` and `std::forward`:

- "On the Superfluousness of `std::move`," Scott Meyers, *The View from Aristeia* (blog), 11 November 2012.

Further Information

`noexcept`:

- "Exception Specifications in C++ 2011," Dietmar Kühl, *Overload*, June 2011.
- "Using noexcept," Andrzej Krzemieński, *Andrzej's C++ blog*, 10 June 2011.
- "How on earth did noexcept get through the standards process?," *Usenet newsgroup comp.lang.c++.moderated*, discussion initiated 14 March 2011.
  - ➧ Bottom of thread discusses `noexcept` and optimization.
- "Summary of C++0x Feature Availability," *Usenet newsgroup comp.std.c++*, discussion initiated 7 March 2010.
  - ➧ Includes discussion of `noexcept` and optimization.

Further Information

Cost of move operations:

- "std::string, SSO, and Move Semantics," Scott Meyers, *The View from Aristeia*, 10 April 2012.
- "Small String Optimization and Move Operations," John Ahlgren, *John Ahlgren*, 30 March 2012.

Further Information

The Concurrency API:

- *The C++ Standard Library, Second Edition*, Nicolai M. Josuttis, Addison-Wesley, 2012.
- *C++ Concurrency in Action*, Anthony Williams, Manning, 2012.
- "(Not) using std::thread," Andrzej Krzemieński, *Andrzej's C++ blog*, 14 November 2012.
- "N2802: A plea to reconsider detach-on-destruction for thread objects," Hans-J. Boehm, 4 December 2008.
- "std::futures from std::async aren't special!," Scott Meyers, *The View from Aristeia*, 20 March 2013.
- "async and ~future," Herb Sutter, C++ Standardization Committee Document N3451, 23 September 2012.

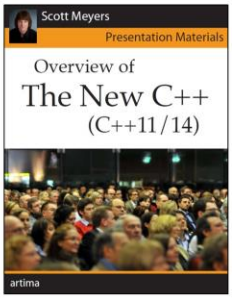
Scott Meyers, Software Development Consultant
http://aristeia.com/
© 2013 Scott Meyers, all rights reserved.

Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- Commercial use: http://aristeia.com/Licensing/licensing.html
- Personal use: http://aristeia.com/Licensing/personalUse.html

Courses currently available for personal use include:



About Scott Meyers

Scott offers training and consulting services on the design and implementation of C++ software systems. His web site,

http://aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog